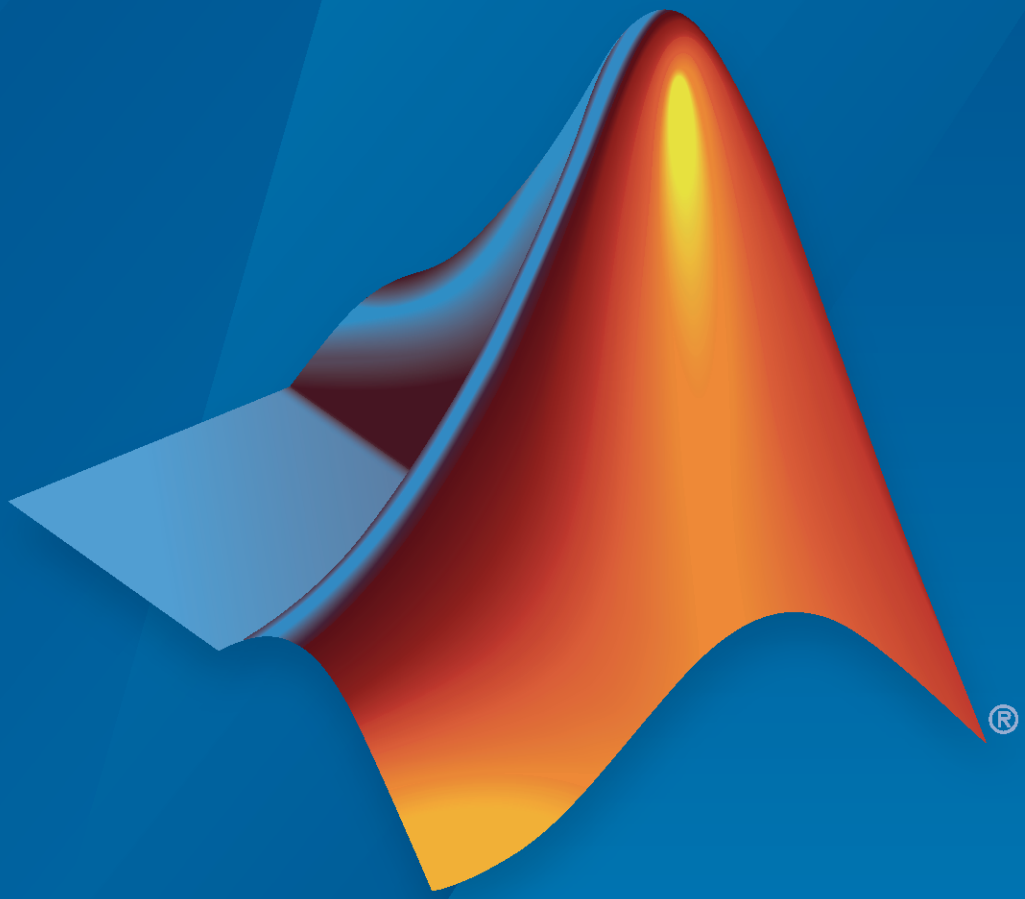
Text Analytics Toolbox™

User's Guide



MATLAB®

R2020b

MathWorks®

How to Contact MathWorks

Latest news: `www.mathworks.com`

Sales and services: `www.mathworks.com/sales_and_services`

User community: `www.mathworks.com/matlabcentral`

Technical support: `www.mathworks.com/support/contact_us`

Phone: 508-647-7000

The MathWorks, Inc.
1 Apple Hill Drive
Natick, MA 01760-2098

Text Analytics Toolbox™ User's Guide

Revision History

March 2018	Online Only	New for Version 1.1 (Release 2018a)
September 2018	Online Only	Revised for Version 1.2 (Release 2018b)
March 2019	Online Only	Revised for Version 1.3 (Release 2019a)
September 2019	Online Only	Revised for Version 1.4 (Release 2019b)
March 2020	Online Only	Revised for Version 1.5 (Release 2020a)
September 2020	Online Only	Revised for Version 1.6 (Release 2020b)

Contents

1 Text Data Preparation

2 Modeling and Prediction

Text Data Preparation

Extract Text Data from Files

This example shows how to extract the text data from text, HTML, Microsoft® Word, PDF, CSV, and Microsoft Excel® files and import it into MATLAB® for analysis.

Usually, the easiest way to import text data into MATLAB is to use the `extractFileText` function. This function extracts the text data from text, PDF, HTML, and Microsoft Word files. To import text from CSV and Microsoft Excel files, use `readtable`. To extract text from HTML code, use `extractHTMLText`. To read data from PDF forms, use `readPDFFormData`.

Text File

Extract the text from `sonnets.txt` using `extractFileText`. The file `sonnets.txt` contains Shakespeare's sonnets in plain text.

```
filename = "sonnets.txt";
str = extractFileText(filename);
```

View the first sonnet by extracting the text between the two titles "`I`" and "`II`".

```
start = " I" + newline;
fin = " II";
sonnet1 = extractBetween(str,start,fin)
```

```
sonnet1 =
    "
       From fairest creatures we desire increase,
       That thereby beauty's rose might never die,
       But as the riper should by time decease,
       His tender heir might bear his memory:
       But thou, contracted to thine own bright eyes,
       Feed'st thy light's flame with self-substantial fuel,
       Making a famine where abundance lies,
       Thy self thy foe, to thy sweet self too cruel:
       Thou that art now the world's fresh ornament,
       And only herald to the gaudy spring,
       Within thine own bud buriest thy content,
       And tender churl mak'st waste in niggarding:
         Pity the world, or else this glutton be,
         To eat the world's due, by the grave and thee.

      "
```

For text files containing multiple documents seperated by newline characters, use the `readlines` function.

```
filename = "multilineSonnets.txt";
str = readlines(filename)
```

```
str = 3×1 string
    "From fairest creatures we desire increase, That thereby beauty's rose might never die, But a
    "When forty winters shall besiege thy brow, And dig deep trenches in thy beauty's field, Thy
    "Look in thy glass and tell the face thou viewest Now is the time that face should form anoth
```

Microsoft Word Document

Extract the text from `sonnets.docx` using `extractFileText`. The file `exampleSonnets.docx` contains Shakespeare's sonnets in a Microsoft Word document.

```
filename = "exampleSonnets.docx";
str = extractFileText(filename);
```

View the second sonnet by extracting the text between the two titles "`II`" and "`III`".

```
start = " II" + newline;
fin = " III";
sonnet2 = extractBetween(str,start,fin)
```

```
sonnet2 =
    "
       When forty winters shall besiege thy brow,

       And dig deep trenches in thy beauty's field,

       Thy youth's proud livery so gazed on now,

       Will be a tatter'd weed of small worth held:

       Then being asked, where all thy beauty lies,

       Where all the treasure of thy lusty days;

       To say, within thine own deep sunken eyes,

       Were an all-eating shame, and thriftless praise.

       How much more praise deserv'd thy beauty's use,

       If thou couldst answer 'This fair child of mine

       Shall sum my count, and make my old excuse,'

       Proving his beauty by succession thine!

         This were to be new made when thou art old,

         And see thy blood warm when thou feel'st it cold.

      "
```

The example Microsoft Word document uses two newline characters between each line. To replace these characters with a single newline character, use the `replace` function.

```
sonnet2 = replace(sonnet2,[newline newline],newline)
```

```
sonnet2 =
    "
       When forty winters shall besiege thy brow,
       And dig deep trenches in thy beauty's field,
       Thy youth's proud livery so gazed on now,
       Will be a tatter'd weed of small worth held:
```

```
       Then being asked, where all thy beauty lies,
       Where all the treasure of thy lusty days;
       To say, within thine own deep sunken eyes,
       Were an all-eating shame, and thriftless praise.
       How much more praise deserv'd thy beauty's use,
       If thou couldst answer 'This fair child of mine
       Shall sum my count, and make my old excuse,'
       Proving his beauty by succession thine!
         This were to be new made when thou art old,
         And see thy blood warm when thou feel'st it cold.
     "
```

PDF Files

Extract text from PDF documents and data from PDF forms.

PDF Document

Extract the text from `sonnets.pdf` using `extractFileText`. The file `exampleSonnets.pdf` contains Shakespeare's sonnets in a PDF.

```
filename = "exampleSonnets.pdf";
str = extractFileText(filename);
```

View the third sonnet by extracting the text between the two titles "`III`" and "`IV`". This PDF has a space before each newline character.

```
start = " III " + newline;
fin = "IV";
sonnet3 = extractBetween(str,start,fin)
```

```
sonnet3 =
    "
       Look in thy glass and tell the face thou viewest
       Now is the time that face should form another;
       Whose fresh repair if now thou not renewest,
       Thou dost beguile the world, unbless some mother.
       For where is she so fair whose unear'd womb
       Disdains the tillage of thy husbandry?
       Or who is he so fond will be the tomb,
       Of his self-love to stop posterity?
       Thou art thy mother's glass and she in thee
       Calls back the lovely April of her prime;
       So thou through windows of thine age shalt see,
       Despite of wrinkles this thy golden time.
         But if thou live, remember'd not to be,
         Die single and thine image dies with thee.


       "
```

PDF Form

To read text data from PDF forms, use `readPDFFormData`. The function returns a struct containing the data from the PDF form fields.

```
filename = "weatherReportForm1.pdf";
data = readPDFFormData(filename)

data = struct with fields:
          event_type: "Thunderstorm Wind"
    event_narrative: "Large tree down between Plantersville and Nettleton."
```

HTML

Extract text from HTML files, HTML code, and the web.

HTML File

To extract text data from a saved HTML file, use `extractFileText`.

```
filename = "exampleSonnets.html";
str = extractFileText(filename);
```

View the forth sonnet by extracting the text between the two titles `"IV"` and `"V"`.

```
start = newline + "IV" + newline;
fin = newline + "V" + newline;
sonnet4 = extractBetween(str,start,fin)

sonnet4 =
    "
     Unthrifty loveliness, why dost thou spend
      Upon thy self thy beauty's legacy?
      Nature's bequest gives nothing, but doth lend,
      And being frank she lends to those are free:
      Then, beauteous niggard, why dost thou abuse
      The bounteous largess given thee to give?
      Profitless usurer, why dost thou use
      So great a sum of sums, yet canst not live?
      For having traffic with thy self alone,
      Thou of thy self thy sweet self dost deceive:
      Then how when nature calls thee to be gone,
      What acceptable audit canst thou leave?
      Thy unused beauty must be tombed with thee,
      Which, used, lives th' executor to be.
     "
```

HTML Code

To extract text data from a string containing HTML code, use `extractHTMLText`.

```
code = "<html><body><h1>THE SONNETS</h1><p>by William Shakespeare</p></body></html>";
str = extractHTMLText(code)

str =
    "THE SONNETS

     by William Shakespeare"
```

From the Web

To extract text data from a web page, first read the HTML code using `webread`, and then use `extractHTMLText`.

```
url = "https://www.mathworks.com/help/textanalytics";
code = webread(url);
str = extractHTMLText(code)
```

```
str =
    'Text Analytics Toolbox™ provides algorithms and visualizations for preprocessing, analyzing

     Text Analytics Toolbox includes tools for processing raw text from sources such as equipment

     Using machine learning techniques such as LSA, LDA, and word embeddings, you can find cluste
```

Parse HTML Code

To find particular elements of HTML code, parse the code using `htmlTree` and use `findElement`. Parse the HTML code and find all the hyperlinks. The hyperlinks are nodes with element name `"A"`.

```
tree = htmlTree(code);
selector = "A";
subtrees = findElement(tree,selector);
```

View the first 10 subtrees and extract the text using `extractHTMLText`.

```
subtrees(1:10)
```

```
ans =
  10×1 htmlTree:

    <A class="skip_link sr-only" href="#content_container" onclick="skipLinkFocus()">Skip to cont
    <A class="svg_link navbar-brand" href="https://www.mathworks.com?s_tid=gn_logo"><IMG alt="Mat
    <A href="https://www.mathworks.com/products.html?s_tid=gn_ps">Products</A>
    <A href="https://www.mathworks.com/solutions.html?s_tid=gn_sol">Solutions</A>
    <A href="https://www.mathworks.com/academia.html?s_tid=gn_acad">Academia</A>
    <A href="https://www.mathworks.com/support.html?s_tid=gn_supp">Support</A>
    <A href="https://www.mathworks.com/matlabcentral/?s_tid=gn_mlc">Community</A>
    <A href="https://www.mathworks.com/company/events.html?s_tid=gn_ev">Events</A>
    <A href="https://www.mathworks.com/company/aboutus/contact_us.html?s_tid=gn_cntus">Contact Us
    <A href="https://www.mathworks.com/products/get-matlab.html?s_tid=gn_getml">Get MATLAB</A>
```

```
str = extractHTMLText(subtrees);
```

View the extracted text of the first 10 hyperlinks.

```
str(1:10)
```

```
ans = 10×1 string
    "Skip to content"
    ""
    "Products"
    "Solutions"
    "Academia"
    "Support"
    "Community"
```

```
    "Events"
    "Contact Us"
    "Get MATLAB"
```

To get the link targets, use `getAttributes` and specify the attribute `"href"` (hyperlink reference). Get the link targets of the first 10 subtrees.

```
attr = "href";
str = getAttribute(subtrees(1:10),attr)
```

```
str = 10×1 string
    "#content_container"
    "https://www.mathworks.com?s_tid=gn_logo"
    "https://www.mathworks.com/products.html?s_tid=gn_ps"
    "https://www.mathworks.com/solutions.html?s_tid=gn_sol"
    "https://www.mathworks.com/academia.html?s_tid=gn_acad"
    "https://www.mathworks.com/support.html?s_tid=gn_supp"
    "https://www.mathworks.com/matlabcentral/?s_tid=gn_mlc"
    "https://www.mathworks.com/company/events.html?s_tid=gn_ev"
    "https://www.mathworks.com/company/aboutus/contact_us.html?s_tid=gn_cntus"
    "https://www.mathworks.com/products/get-matlab.html?s_tid=gn_getml"
```

CSV and Microsoft Excel Files

To extract text data from CSV and Microsoft Excel files, use `readtable` and extract the text data from the table that it returns.

Extract the table data from `factoryReposts.csv` using the `readtable` function and view the first few rows of the table.

```
T = readtable('factoryReports.csv','TextType','string');
head(T)
```

```
ans=8×5 table
                                Description                                      Category
    ____________________________________________________________________    ____________________

    "Items are occasionally getting stuck in the scanner spools."           "Mechanical Failure"
    "Loud rattling and banging sounds are coming from assembler pistons."   "Mechanical Failure"
    "There are cuts to the power when starting the plant."                  "Electronic Failure"
    "Fried capacitors in the assembler."                                    "Electronic Failure"
    "Mixer tripped the fuses."                                              "Electronic Failure"
    "Burst pipe in the constructing agent is spraying coolant."             "Leak"
    "A fuse is blown in the mixer."                                         "Electronic Failure"
    "Things continue to tumble off of the belt."                            "Mechanical Failure"
```

Extract the text data from the `event_narrative` column and view the first few strings.

```
str = T.Description;
str(1:10)
```

```
ans = 10×1 string
    "Items are occasionally getting stuck in the scanner spools."
    "Loud rattling and banging sounds are coming from assembler pistons."
    "There are cuts to the power when starting the plant."
    "Fried capacitors in the assembler."
```

```
    "Mixer tripped the fuses."
    "Burst pipe in the constructing agent is spraying coolant."
    "A fuse is blown in the mixer."
    "Things continue to tumble off of the belt."
    "Falling items from the conveyor belt."
    "The scanner reel is split, it will soon begin to curve."
```

Extract Text from Multiple Files

If your text data is contained in multiple files in a folder, then you can import the text data into MATLAB using a file datastore.

Create a file datastore for the example sonnet text files. The example files are named "`exampleSonnetN.txt`", where `N` is the number of the sonnet. Specify the file name using the wildcard "*" to find all file names of this structure. To specify the read function to be `extractFileText`, input this function to `fileDatastore` using a function handle.

```
location = fullfile(matlabroot,"examples","textanalytics","data","exampleSonnet*.txt");
fds = fileDatastore(location,'ReadFcn',@extractFileText)
```

```
fds =
  FileDatastore with properties:

                      Files: {
                             ' ...\matlab\examples\textanalytics\data\exampleSonnet1.txt';
                             ' ...\matlab\examples\textanalytics\data\exampleSonnet2.txt';
                             ' ...\matlab\examples\textanalytics\data\exampleSonnet3.txt'
                              ... and 2 more
                             }
                    Folders: {
                             ' ...\matlab\examples\textanalytics\data'
                             }
                UniformRead: 0
                   ReadMode: 'file'
                  BlockSize: Inf
                 PreviewFcn: @extractFileText
      SupportedOutputFormats: ["txt"    "csv"    "xlsx"    "xls"    "parquet"    "parq"    "png"
                     ReadFcn: @extractFileText
    AlternateFileSystemRoots: {}
```

Loop over the files in the datastore and read each text file.

```
str = [];
while hasdata(fds)
    textData = read(fds);
    str = [str; textData];
end
```

View the extracted text.

```
str
```

```
str = 5×1 string
    "  From fairest creatures we desire increase,↵  That thereby beauty's rose might never die,↵
    "  When forty winters shall besiege thy brow,↵  And dig deep trenches in thy beauty's field,
    "  Look in thy glass and tell the face thou viewest↵  Now is the time that face should form a
```

```
"  Unthrifty loveliness, why dost thou spend↵  Upon thy self thy beauty's legacy?↵  Nature's
"from fairest creatures we desire increase that thereby beautys rose might never die but as
```

See Also

`extractFileText` | `extractHTMLText` | `readPDFFormData` | `tokenizedDocument`

Related Examples

- “Prepare Text Data for Analysis” on page 1-10
- “Create Simple Text Model for Classification” on page 2-2
- “Visualize Text Data Using Word Clouds” on page 3-2
- “Analyze Text Data Containing Emojis” on page 2-32
- “Analyze Text Data Using Topic Models” on page 2-13
- “Analyze Text Data Using Multiword Phrases” on page 2-7
- “Classify Text Data Using Deep Learning” on page 2-65
- “Train a Sentiment Classifier” on page 2-51

Prepare Text Data for Analysis

This example shows how to create a function which cleans and preprocesses text data for analysis.

Text data can be large and can contain lots of noise which negatively affects statistical analysis. For example, text data can contain the following:

- Variations in case, for example "new" and "New"
- Variations in word forms, for example "walk" and "walking"
- Words which add noise, for example stop words such as "the" and "of"
- Punctuation and special characters
- HTML and XML tags

These word clouds illustrate word frequency analysis applied to some raw text data from factory reports, and a preprocessed version of the same text data.

Load and Extract Text Data

Load the example data. The file `factoryReports.csv` contains factory reports, including a text description and categorical labels for each event.

```
filename = "factoryReports.csv";
data = readtable(filename,'TextType','string');
```

Extract the text data from the field `Description`, and the label data from the field `Category`.

```
textData = data.Description;
labels = data.Category;
textData(1:10)
```

```
ans = 10×1 string
    "Items are occasionally getting stuck in the scanner spools."
    "Loud rattling and banging sounds are coming from assembler pistons."
    "There are cuts to the power when starting the plant."
    "Fried capacitors in the assembler."
    "Mixer tripped the fuses."
    "Burst pipe in the constructing agent is spraying coolant."
    "A fuse is blown in the mixer."
    "Things continue to tumble off of the belt."
    "Falling items from the conveyor belt."
    "The scanner reel is split, it will soon begin to curve."
```

Create Tokenized Documents

Create an array of tokenized documents.

```
cleanedDocuments = tokenizedDocument(textData);
cleanedDocuments(1:10)
```

```
ans =
  10×1 tokenizedDocument:

    10 tokens: Items are occasionally getting stuck in the scanner spools .
    11 tokens: Loud rattling and banging sounds are coming from assembler pistons .
    11 tokens: There are cuts to the power when starting the plant .
     6 tokens: Fried capacitors in the assembler .
     5 tokens: Mixer tripped the fuses .
    10 tokens: Burst pipe in the constructing agent is spraying coolant .
     8 tokens: A fuse is blown in the mixer .
     9 tokens: Things continue to tumble off of the belt .
     7 tokens: Falling items from the conveyor belt .
    13 tokens: The scanner reel is split , it will soon begin to curve .
```

To improve lemmatization, add part of speech details to the documents using `addPartOfSpeechDetails`. Use the `addPartOfSpeech` function before removing stop words and lemmatizing.

```
cleanedDocuments = addPartOfSpeechDetails(cleanedDocuments);
```

Words like "a", "and", "to", and "the" (known as stop words) can add noise to data. Remove a list of stop words using the `removeStopWords` function. Use the `removeStopWords` function before using the `normalizeWords` function.

```
cleanedDocuments = removeStopWords(cleanedDocuments);
cleanedDocuments(1:10)
```

```
ans =
  10×1 tokenizedDocument:

    7 tokens: Items occasionally getting stuck scanner spools .
    8 tokens: Loud rattling banging sounds coming assembler pistons .
    5 tokens: cuts power starting plant .
    4 tokens: Fried capacitors assembler .
```

```
    4 tokens: Mixer tripped fuses .
    7 tokens: Burst pipe constructing agent spraying coolant .
    4 tokens: fuse blown mixer .
    6 tokens: Things continue tumble off belt .
    5 tokens: Falling items conveyor belt .
    8 tokens: scanner reel split , soon begin curve .
```

Lemmatize the words using `normalizeWords`.

```
cleanedDocuments = normalizeWords(cleanedDocuments,'Style','lemma');
cleanedDocuments(1:10)
```

```
ans =
  10×1 tokenizedDocument:

    7 tokens: items occasionally get stuck scanner spool .
    8 tokens: loud rattle bang sound come assembler piston .
    5 tokens: cut power start plant .
    4 tokens: fry capacitor assembler .
    4 tokens: mixer trip fuse .
    7 tokens: burst pipe constructing agent spray coolant .
    4 tokens: fuse blow mixer .
    6 tokens: thing continue tumble off belt .
    5 tokens: fall item conveyor belt .
    8 tokens: scanner reel split , soon begin curve .
```

Erase the punctuation from the documents.

```
cleanedDocuments = erasePunctuation(cleanedDocuments);
cleanedDocuments(1:10)
```

```
ans =
  10×1 tokenizedDocument:

    6 tokens: items occasionally get stuck scanner spool
    7 tokens: loud rattle bang sound come assembler piston
    4 tokens: cut power start plant
    3 tokens: fry capacitor assembler
    3 tokens: mixer trip fuse
    6 tokens: burst pipe constructing agent spray coolant
    3 tokens: fuse blow mixer
    5 tokens: thing continue tumble off belt
    4 tokens: fall item conveyor belt
    6 tokens: scanner reel split soon begin curve
```

Remove words with 2 or fewer characters, and words with 15 or greater characters.

```
cleanedDocuments = removeShortWords(cleanedDocuments,2);
cleanedDocuments = removeLongWords(cleanedDocuments,15);
cleanedDocuments(1:10)
```

```
ans =
  10×1 tokenizedDocument:

    6 tokens: items occasionally get stuck scanner spool
    7 tokens: loud rattle bang sound come assembler piston
```

```
    4 tokens: cut power start plant
    3 tokens: fry capacitor assembler
    3 tokens: mixer trip fuse
    6 tokens: burst pipe constructing agent spray coolant
    3 tokens: fuse blow mixer
    5 tokens: thing continue tumble off belt
    4 tokens: fall item conveyor belt
    6 tokens: scanner reel split soon begin curve
```

Create Bag-of-Words Model

Create a bag-of-words model.

```
cleanedBag = bagOfWords(cleanedDocuments)
```

```
cleanedBag =
  bagOfWords with properties:

          Counts: [480×352 double]
      Vocabulary: [1×352 string]
        NumWords: 352
    NumDocuments: 480
```

Remove words that do not appear more than two times in the bag-of-words model.

```
cleanedBag = removeInfrequentWords(cleanedBag,2)
```

```
cleanedBag =
  bagOfWords with properties:

          Counts: [480×163 double]
      Vocabulary: [1×163 string]
        NumWords: 163
    NumDocuments: 480
```

Some preprocessing steps such as `removeInfrequentWords` leaves empty documents in the bag-of-words model. To ensure that no empty documents remain in the bag-of-words model after preprocessing, use `removeEmptyDocuments` as the last step.

Remove empty documents from the bag-of-words model and the corresponding labels from `labels`.

```
[cleanedBag,idx] = removeEmptyDocuments(cleanedBag);
labels(idx) = [];
cleanedBag
```

```
cleanedBag =
  bagOfWords with properties:

          Counts: [480×163 double]
      Vocabulary: [1×163 string]
        NumWords: 163
    NumDocuments: 480
```

Create a Preprocessing Function

It can be useful to create a function which performs preprocessing so you can prepare different collections of text data in the same way. For example, you can use a function so that you can preprocess new data using the same steps as the training data.

Create a function which tokenizes and preprocesses the text data so it can be used for analysis. The function `preprocessText`, performs the following steps:

1. Tokenize the text using `tokenizedDocument`.
2. Remove a list of stop words (such as "and", "of", and "the") using `removeStopWords`.
3. Lemmatize the words using `normalizeWords`.
4. Erase punctuation using `erasePunctuation`.
5. Remove words with 2 or fewer characters using `removeShortWords`.
6. Remove words with 15 or more characters using `removeLongWords`.

Use the example preprocessing function `preprocessText` to prepare the text data.

```
newText = "The sorting machine is making lots of loud noises.";
newDocuments = preprocessText(newText)
```

```
newDocuments =
  tokenizedDocument:

   6 tokens: sorting machine make lot loud noise
```

Compare with Raw Data

Compare the preprocessed data with the raw data.

```
rawDocuments = tokenizedDocument(textData);
rawBag = bagOfWords(rawDocuments)
```

```
rawBag =
  bagOfWords with properties:

          Counts: [480×555 double]
      Vocabulary: [1×555 string]
        NumWords: 555
    NumDocuments: 480
```

Calculate the reduction in data.

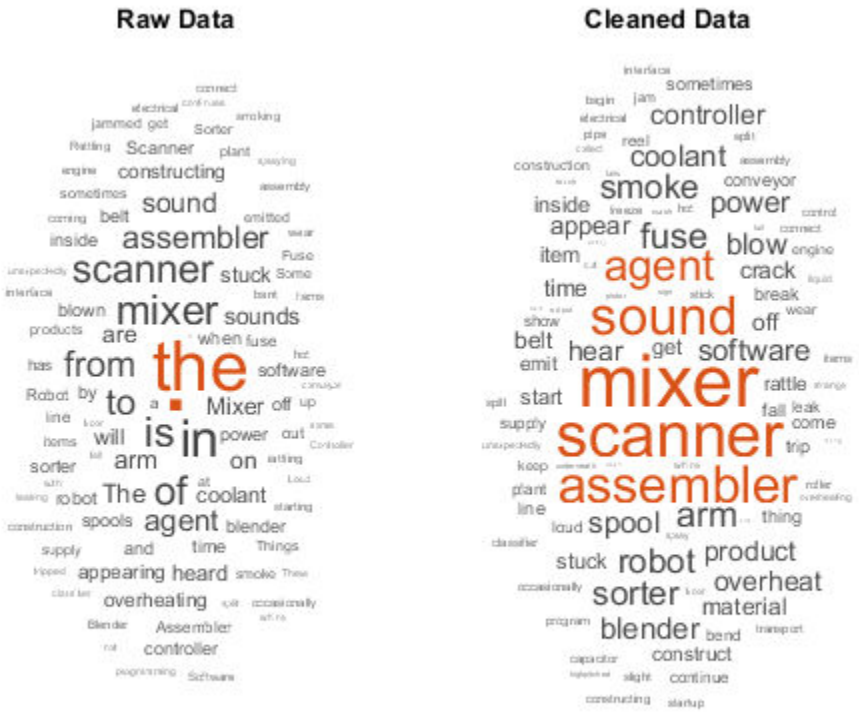
```
numWordsCleaned = cleanedBag.NumWords;
numWordsRaw = rawBag.NumWords;
reduction = 1 - numWordsCleaned/numWordsRaw
```

```
reduction = 0.7063
```

Compare the raw data and the cleaned data by visualizing the two bag-of-words models using word clouds.

```
figure
subplot(1,2,1)
```

```
wordcloud(rawBag);
title("Raw Data")
subplot(1,2,2)
wordcloud(cleanedBag);
title("Cleaned Data")
```



Preprocessing Function

The function `preprocessText`, performs the following steps in order:

1. Tokenize the text using `tokenizedDocument`.
2. Remove a list of stop words (such as "and", "of", and "the") using `removeStopWords`.
3. Lemmatize the words using `normalizeWords`.
4. Erase punctuation using `erasePunctuation`.
5. Remove words with 2 or fewer characters using `removeShortWords`.
6. Remove words with 15 or more characters using `removeLongWords`.

```
function documents = preprocessText(textData)

% Tokenize the text.
documents = tokenizedDocument(textData);

% Remove a list of stop words then lemmatize the words. To improve
% lemmatization, first use addPartOfSpeechDetails.
documents = addPartOfSpeechDetails(documents);
```

```
documents = removeStopWords(documents);
documents = normalizeWords(documents,'Style','lemma');

% Erase punctuation.
documents = erasePunctuation(documents);

% Remove words with 2 or fewer characters, and words with 15 or more
% characters.
documents = removeShortWords(documents,2);
documents = removeLongWords(documents,15);

end
```

See Also

`addPartOfSpeechDetails` | `bagOfWords` | `erasePunctuation` | `normalizeWords` | `removeEmptyDocuments` | `removeInfrequentWords` | `removeLongWords` | `removeShortWords` | `removeStopWords` | `tokenizedDocument` | `wordcloud`

Related Examples

- “Extract Text Data from Files” on page 1-2
- “Create Simple Text Model for Classification” on page 2-2
- “Visualize Text Data Using Word Clouds” on page 3-2
- “Analyze Text Data Containing Emojis” on page 2-32
- “Analyze Text Data Using Topic Models” on page 2-13
- “Analyze Text Data Using Multiword Phrases” on page 2-7
- “Classify Text Data Using Deep Learning” on page 2-65
- “Train a Sentiment Classifier” on page 2-51

Parse HTML and Extract Text Content

This example shows how to parse HTML code and extract the text content from particular elements.

Parse HTML Code

Read HTML code from the URL `https://www.mathworks.com/help/textanalytics` using `webread`.

```
url = "https://www.mathworks.com/help/textanalytics";
code = webread(url);
```

Parse the HTML code using `htmlTree`.

```
tree = htmlTree(code);
```

View the HTML element name of the tree.

```
tree.Name
```

```
ans =
"HTML"
```

View the child elements of the tree. The children are subtrees of `tree`.

```
tree.Children
```

```
ans =
  4×1 htmlTree:

    " "
    <HEAD><TITLE>Text Analytics Toolbox Documentation</TITLE><META charset="utf-8"/><META content
    " "
    <BODY id="responsive_offcanvas"><!-- Mobile TopNav: Start --><DIV class="header visible-xs vi
```

Extract Text from HTML Tree

To extract text directly from the HTML tree, use `extractHTMLText`.

```
str = extractHTMLText(tree)
```

```
str =
    "Text Analytics Toolbox™ provides algorithms and visualizations for preprocessing, analyzing

     Text Analytics Toolbox includes tools for processing raw text from sources such as equipment

     Using machine learning techniques such as LSA, LDA, and word embeddings, you can find cluste
```

Find HTML Elements

To find particular elements of an HTML tree, use `findElement`. Find all the hyperlinks in the HTML tree. In HTML, hyperlinks use the "A" tag.

```
selector = "A";
subtrees = findElement(tree,selector);
```

View the first few subtrees.

```
subtrees(1:20)
```

```
ans =
  20×1 htmlTree:

    <A class="svg_link navbar-brand" href="https://www.mathworks.com?s_tid=gn_logo"><IMG alt="Ma
    <A class="mwa-nav_login" href="https://www.mathworks.com/login?uri=http://www.mathworks.com/
    <A href="https://www.mathworks.com/products.html?s_tid=gn_ps">Products</A>
    <A href="https://www.mathworks.com/solutions.html?s_tid=gn_sol">Solutions</A>
    <A href="https://www.mathworks.com/academia.html?s_tid=gn_acad">Academia</A>
    <A href="https://www.mathworks.com/support.html?s_tid=gn_supp">Support</A>
    <A href="https://www.mathworks.com/matlabcentral/?s_tid=gn_mlc">Community</A>
    <A href="https://www.mathworks.com/company/events.html?s_tid=gn_ev">Events</A>
    <A href="https://www.mathworks.com/company/aboutus/contact_us.html?s_tid=gn_cntus">Contact Us
    <A href="https://www.mathworks.com/store?s_cid=store_top_nav&amp;s_tid=gn_store">How to Buy</
    <A href="https://www.mathworks.com/company/aboutus/contact_us.html?s_tid=gn_cntus">Contact Us
    <A href="https://www.mathworks.com/store?s_cid=store_top_nav&amp;s_tid=gn_store">How to Buy</
    <A class="mwa-nav_login" href="https://www.mathworks.com/login?uri=http://www.mathworks.com/
    <A class="svg_link pull-left" href="https://www.mathworks.com?s_tid=gn_logo"><IMG alt="MathWo
    <A href="https://www.mathworks.com/products.html?s_tid=gn_ps">Products</A>
    <A href="https://www.mathworks.com/solutions.html?s_tid=gn_sol">Solutions</A>
    <A href="https://www.mathworks.com/academia.html?s_tid=gn_acad">Academia</A>
    <A href="https://www.mathworks.com/support.html?s_tid=gn_supp">Support</A>
    <A href="https://www.mathworks.com/matlabcentral/?s_tid=gn_mlc">Community</A>
    <A href="https://www.mathworks.com/company/events.html?s_tid=gn_ev">Events</A>
```

Create a word cloud from the text of the hyperlinks.

```
str = extractHTMLText(subtrees);
figure
wordcloud(str);
title("Hyperlinks")
```

Get HTML Attributes

Get the class attributes from the paragraph elements in the HTML tree.

```
subtrees = findElement(tree,'p');
attr = "class";
str = getAttribute(subtrees,attr)
```

```
str = 21×1 string array
    <missing>
    <missing>
    "add_margin_5"
    <missing>
    <missing>
    <missing>
    <missing>
    <missing>
    "category_desc"
    "category_desc"
    "category_desc"
    "category_desc"
    <missing>
    <missing>
    <missing>
    "text-center"
    <missing>
    <missing>
    <missing>
```

```
    "copyright"
    <missing>
```

Create a word cloud from the text contained in paragraph elements with class `"category_desc"`.

```
subtrees = findElement(tree,'p.category_desc');
str = extractHTMLText(subtrees);
figure
wordcloud(str);
```

See Also

`extractHTMLText` | `findElement` | `getAttribute` | `htmlTree` | `tokenizedDocument`

Related Examples

- “Prepare Text Data for Analysis” on page 1-10
- “Create Simple Text Model for Classification” on page 2-2
- “Visualize Text Data Using Word Clouds” on page 3-2
- “Analyze Text Data Using Topic Models” on page 2-13
- “Analyze Text Data Using Multiword Phrases” on page 2-7
- “Classify Text Data Using Deep Learning” on page 2-65
- “Train a Sentiment Classifier” on page 2-51

Correct Spelling in Documents

This example shows how to correct spelling in documents using Hunspell.

Load Text Data

Create an array of tokenized documents.

```
str = [
    "Use MATLAB to correct spelling of words."
    "Correctly spelled worrds are important for lemmatization."
    "Text Analytics Toolbox providesfunctions for spelling correction."];
documents = tokenizedDocument(str)
```

```
documents =
  3x1 tokenizedDocument:

    8 tokens: Use MATLAB to correct spelling of words .
    8 tokens: Correctly spelled worrds are important for lemmatization .
    8 tokens: Text Analytics Toolbox providesfunctions for spelling correction .
```

Correct Spelling

Correct the spelling of the documents using the `correctSpelling` function.

```
updatedDocuments = correctSpelling(documents)
```

```
updatedDocuments =
  3x1 tokenizedDocument:

    9 tokens: Use MAT LAB to correct spelling of words .
    8 tokens: Correctly spelled words are important for solemnization .
    9 tokens: Text Analytic Toolbox provides functions for spelling correction .
```

Notice that:

- The input word "MATLAB" has been split into the two words "MAT" and "LAB".
- The input word "worrds" has been changed to "words".
- The input word "lemmatization" has been changed to "solemnization".
- The input word "Analytics" has been changed to "Analytic".
- The input word "providesfunctions" has been split into the two words "provides" and "functions".

Specify Custom Words

To prevent the software from updating particular words, you can provide a list of known words using the `'KnownWords'` option of the `correctSpelling` function.

Correct the spelling of the documents again and specify the words "MATLAB", "Analytics", and "lemmatization" as known words.

```
updatedDocuments = correctSpelling(documents,'KnownWords',["MATLAB" "Analytics" "lemmatization"])
```

```
updatedDocuments =
  3x1 tokenizedDocument:
```

```
8 tokens: Use MATLAB to correct spelling of words .
8 tokens: Correctly spelled words are important for lemmatization .
9 tokens: Text Analytics Toolbox provides functions for spelling correction .
```

Notice here that the words "MATLAB", "Analytics", and "lemmatization" remain unchanged.

See Also

`correctSpelling` | `tokenizedDocument`

More About

- “Create Extension Dictionary for Spelling Correction” on page 1-23
- “Create Custom Spelling Correction Function Using Edit Distance Searchers” on page 1-27
- “Prepare Text Data for Analysis” on page 1-10
- “Create Simple Text Model for Classification” on page 2-2
- “Analyze Text Data Using Topic Models” on page 2-13

Create Extension Dictionary for Spelling Correction

This example shows how to create a Hunspell extension dictionary for spelling correction.

When using the `correctSpelling` function, the function may update some correctly spelled words. To provide a list of known words, you can use the "'KnownWords'" option directly with a string array of known words. Alternatively, you can specify a Hunspell extension dictionary (also known as a *personal dictionary*) that specifies lists of known words, forbidden words, and words alongside affix rules.

Specify Known Words

Create an array of tokenized documents.

```
str = [
    "Use MATLAB to correct spelling of words."
    "Correctly spelled worrds are important for lemmatizing."
    "Text Analytics Toolbox providesfunctions for spelling correction."];
documents = tokenizedDocument(str);
```

Correct the spelling of the documents using the `correctSpelling` function.

```
updatedDocuments = correctSpelling(documents)
```

```
updatedDocuments =
  3x1 tokenizedDocument:

    9 tokens: Use MAT LAB to correct spelling of words .
    8 tokens: Correctly spelled words are important for legitimatizing .
    9 tokens: Text Analytic Toolbox provides functions for spelling correction .
```

The function has corrected the spelling of the words "worrds" and "providesfunctions", though it has also updated some correctly spelled words:

- The input word "MATLAB" has been split into the two words "MAT" and "LAB".
- The input word "lemmatizing" has been changed to "legitimatizing".
- The input word "Analytics" has been changed to "Analytic".

To create a Hunspell extension dictionary containing a list of known words, create a `.dic` file containing these words with one word per line. Create an extension dictionary with name `knownWords.dic` file containing the words "MATLAB", "lemmatization", and "Analytics".

```
MATLAB
Analytics
lemmatizing
```

Correct the spelling of the documents again and specify the extension dictionary `knownWords.dic`.

```
updatedDocuments = correctSpelling(documents,'ExtensionDictionary','knownWords.dic')
```

```
updatedDocuments =
  3x1 tokenizedDocument:

    8 tokens: Use MATLAB to correct spelling of words .
    8 tokens: Correctly spelled words are important for lemmatizing .
```

```
    9 tokens: Text Analytics Toolbox provides functions for spelling correction .
```

Specify Affix Rules

When specifying multiple words with the same root word (for example, specifying the words "lemmatize", "lemmatizer", "lemmatized", and so on), it can be easier to indicate a set of affix rules. Instead of specifying the same word multiple times with different affixes, you can specify particular word to inherit a set of affix rules from.

For example, create an array of tokenized documents and use the `correctSpelling` function.

```
str = [
    "A lemmatizer reduces words to their dictionary forms."
    "To lemmatize words, use the normalizeWords function."
    "Before lemmatizing, add part of speech details to the text."
    "Display lemmatized words in a word cloud."];
documents = tokenizedDocument(str);
updatedDocuments = correctSpelling(documents)
```

```
updatedDocuments =
  4x1 tokenizedDocument:

     9 tokens: A legitimatize reduces words to their dictionary forms .
    10 tokens: To legitimatize words , use the normalize Words function .
    12 tokens: Before legitimatizing , add part of speech details to the text .
     8 tokens: Display legitimatized words in a word cloud .
```

Notice that the word "normalizeWords" and variants of "lemmatize" do not get updated correctly.

Create an extension dictionary with name `knownWordsWithAffixes.dic` file containing the words "normalizeWords" and "lemmatize". For the word "lemmatize", also specify to also include valid affixes of the word "equalize" using the `"/"` symbol.

```
normalizeWords
lemmatize/equalize
```

Correct the spelling of the documents again and specify the extension dictionary `knownWordsWithAffixes.dic`.

```
updatedDocuments = correctSpelling(documents,'ExtensionDictionary','knownWordsWithAffixes.dic')
```

```
updatedDocuments =
  4x1 tokenizedDocument:

     9 tokens: A lemmatizer reduces words to their dictionary forms .
     9 tokens: To lemmatize words , use the normalizeWords function .
    12 tokens: Before lemmatizing , add part of speech details to the text .
     8 tokens: Display lemmatized words in a word cloud .
```

Notice that the variants of "lemmatize" have not been changed. The default dictionary contains the word "equalize" and also recognizes the words "equalizer" and "equalized" via the "-r" and "-d" suffixes, respectively. By specifying the entry "lemmatize/equalize", the software recognizes the word "lemmatize" as well as other words by extension of the affixes corresponding to "equalize". For example, the words "lemmatizer" and "lemmatized".

Specify Forbidden Words

When using the `correctSpelling` function, the function may output undesirable words, even if a more desirable word is in the dictionary. For example, for the input word "MALTAB", the `correctSpelling` function may output the words "MALT AB" or the word "MALTA". To ensure that certain words to not appear in the output, you can specify forbidden words in the extension dictionary.

For example, create an array of tokenized documents and correct the spelling using the extension dictionary `knownWords.dic`. Note that this dictionary contains the word "MATLAB".

```
str = [
    "Analyze text data using MATLAB."
    "Use MALTAB for text analysis."];
documents = tokenizedDocument(str);
updatedDocuments = correctSpelling(documents,'ExtensionDictionary','knownWords.dic')
```

```
updatedDocuments =
  2x1 tokenizedDocument:

    6 tokens: Analyze text data using MATLAB .
    7 tokens: Use MALT AB for text analysis .
```

Even though the word "MATLAB" is in the dictionary or extension dictionary, the software may still choose other words as matches to incorrectly spelled words close to "MATLAB".

Create an extension dictionary with name `knownWordsWithForbiddenWords.dic` file containing the word "MATLAB" and also specify the forbidden words "malt" and "Malta" using the "`*`" symbol.

```
MATLAB
*malt
*Malta
```

Correct the spelling using the extension dictionary `knownWordsWithForbiddenWords.dic`.

```
updatedDocuments = correctSpelling(documents,'ExtensionDictionary','knownWordsWithForbiddenWords
```

```
updatedDocuments =
  2x1 tokenizedDocument:

    6 tokens: Analyze text data using MATLAB .
    6 tokens: Use MATLAB for text analysis .
```

Here, the word "MALTAB" is corrected to "MATLAB".

See Also

`correctSpelling` | `tokenizedDocument`

More About

- “Correct Spelling in Documents” on page 1-21
- “Create Custom Spelling Correction Function Using Edit Distance Searchers” on page 1-27
- “Prepare Text Data for Analysis” on page 1-10

- “Create Simple Text Model for Classification” on page 2-2
- “Analyze Text Data Using Topic Models” on page 2-13

Create Custom Spelling Correction Function Using Edit Distance Searchers

This example shows how to correct spelling using edit distance searchers and a vocabulary of known words.

Lemmatization with `normalizeWords` and `word2vec` requires correctly spelled words to work. To easily correct the spelling of words in text, use the `correctSpelling` function. To learn how to create a spelling correction function from scratch using edit distance searchers, use this example as a guide.

If you have misspelled words in a collection of text, then you can use edit distance searchers to find the nearest correctly spelled words to a given vocabulary. To correct the spelling of misspelled words in documents, replace them with the nearest neighbors in the vocabulary. Use edit distance searchers to find the nearest correctly spelled word to misspelled words according to an edit distance. For example, the number of adjacent grapheme swaps and grapheme insertions, deletions, and substitutions.

Load Data

Create a vocabulary of known words. Download and extract the Spell Checking Oriented Word Lists (SCOWL) from https://sourceforge.net/projects/wordlist/ into a folder in the currrent directory. Import the words from the downloaded data using the supporting function `scowlWordList`.

```
folderName = "scowl-2019.10.06";
maxSize = 60;
vocabulary = scowlWordList(folderName,'english',maxSize);
```

View the number of words in the vocabulary.

```
numWords = numel(vocabulary)
```

```
numWords = 98213
```

Create Simple Spelling Corrector

Using the imported vocabulary, create an edit distance searcher with a maximum distance of 2. For better results, allow for adjacent grapheme swaps by setting the `'SwapCost'` option to 1. For large vocabularies, this can take a few minutes.

```
maxDist = 2;
eds = editDistanceSearcher(vocabulary,maxDist,'SwapCost',1);
```

This edit distance searcher is case sensitive which means that changing the case of characters contributes to the edit distance. For example, the searcher can find the neighbor "testing" for the word "tseting" because it has edit distance 1 (one swap), but not of the word "TSeTiNG" because it has edit distance 6.

Correct Spelling

Correct the spelling of misspelled words in an array of tokenized documents by selecting the misspelled words and finding the nearest neighbors in the edit distance searcher.

Create a tokenized document object containing typos and spelling mistakes.

```
str = "An exmaple dccoument with typos and averyunusualword.";
document = tokenizedDocument(str)
```

```
document =
  tokenizedDocument:

   8 tokens: An exmaple dccoument with typos and averyunusualword .
```

Convert the documents to a string array of words using the string function.

```
words = string(document)
```

```
words = 1×8 string
    "An"    "exmaple"    "dccoument"    "with"    "typos"    "and"    "averyunusualword"    "."
```

Find the words that need correction. To ignore words that are correctly spelled, find the indices of the words already in the vocabulary. To ignore punctuation and complex tokens such as email addresses, find the indices of the words which do not have the token types "letters" or "other". Get the token details from the document using the `tokenDetails` function.

```
tdetails = tokenDetails(document);
idxVocabularyWords = ismember(tdetails.Token,eds.Vocabulary);

idxComplexTokens = ...
    tdetails.Type ~= "letters" & ...
    tdetails.Type ~= "other";

idxWordsToCheck = ...
    ~idxVocabularyWords & ...
    ~idxComplexTokens
```

```
idxWordsToCheck = 8×1 logical array

   1
   1
   1
   0
   0
   0
   1
   0
```

Find the numeric indices of the words and view the corresponding words.

```
idxWordsToCheck = find(idxWordsToCheck)
```

```
idxWordsToCheck = 4×1

     1
     2
     3
     7
```

```
wordsToCheck = words(idxWordsToCheck)
```

```
wordsToCheck = 1×4 string
    "An"    "exmaple"    "dccoument"    "averyunusualword"
```

Notice that the word "An" is flagged as a word to check. This word is flagged because the vocabulary does not contain the word "An" with an uppercase "A". A later section in the example shows how to create a case insensitive spelling corrector.

Find the nearest words and their distances using the `knnsearch` function with the edit distance searcher.

```
[idxNearestWords,d] = knnsearch(eds,wordsToCheck)
```

```
idxNearestWords = 4×1

         165
        1353
        1152
         NaN


d = 4×1

     1
     1
     2
   Inf
```

If any of the words are not found in the searcher, then the function returns index `NaN` with distance `Inf`. The word "averyunusualword" does not have a match within edit distance 2, so the function returns the index `NaN` for that word.

Find the indices of the words with positive finite edit distances.

```
idxMatches = ~isnan(idxNearestWords)
```

```
idxMatches = 4×1 logical array

   1
   1
   1
   0
```

Get the indices of the words with matches in the searcher and view the corresponding corrected words in the vocabulary.

```
idxCorrectedWords = idxNearestWords(idxMatches)
```

```
idxCorrectedWords = 3×1

         165
        1353
        1152


correctedWords = eds.Vocabulary(idxCorrectedWords)
```

```
correctedWords = 1×3 string
    "an"    "example"    "document"
```

Replace the misspelled words that have matches with the corrected words.

```
idxToCorrect = idxWordsToCheck(idxMatches);
words(idxToCorrect) = correctedWords
```

```
words = 1×8 string
    "an"    "example"    "document"    "with"    "typos"    "and"    "averyunusualword"    "."
```

To create a tokenized document of these words, use the `tokenizedDocument` function and set `'TokenizedMethod'` to `'none'`.

```
document = tokenizedDocument(words,'TokenizeMethod','none')
```

```
document =
  tokenizedDocument:

   8 tokens: an example document with typos and averyunusualword .
```

The next section shows how to correct the spelling of multiple documents at once by creating a custom spelling correction function and using `docfun`.

Create Spelling Correction Function

To correct the spelling in multiple documents at once, create a custom function using the code from the previous section and use this function with the `docfun` function.

Create a function that takes an edit distance searcher, a string array of words, and the corresponding table of token details as inputs and outputs the corrected words. The `correctSpelling` function, listed at the end of the example, corrects the spelling in a string array of words using the corresponding token details and an edit distance searcher.

To use this function with the `docfun` function, create a function handle that takes a string array of words and the corresponding table of token details as the inputs.

```
func = @(words,tdetails) correctSpelling(eds,words,tdetails);
```

Correct the spelling of an array of tokenized documents using `docfun` with the function handle `func`.

```
str = [
    "Here is some reallyu badly wrirten texct."
    "Some moree mitsakes here too."];
documents = tokenizedDocument(str);
updatedDocuments = docfun(func,documents)
```

```
updatedDocuments =
  2×1 tokenizedDocument:

    8 tokens: here is some really badly written text .
    6 tokens: come more mistakes here too .
```

Note that uppercase characters can get corrected to different lowercase characters. For example, the word "Some" can get corrected to "come". If multiple words in the edit distance searcher vocabulary

have the same edit distance to the input, then the function outputs the first result it found. For example, the words "come" and "some" both have edit distance 1 from the word "Some".

The next section shows how to create an spelling corrector that is case insensitive.

Create Case Insensitive Spelling Corrector

To prevent differences in case clashing with other substitutions, create an edit distance searcher with the vocabulary in lower case and convert the documents to lowercase before using the edit distance searcher.

Convert the vocabulary to lowercase. This operation can introduce duplicate words, remove them by taking the unique values only.

```
vocabularyLower = lower(vocabulary);
vocabularyLower = unique(vocabularyLower);
```

Create an edit distance searcher using the lowercase vocabulary using the same options as before. This can take a few minutes to run.

```
maxDist = 2;
eds = editDistanceSearcher(vocabularyLower,maxDist,'SwapCost',1);
```

Use the edit distance searcher to correct the spelling of the words in tokenized document. To use the case insensitive spelling corrector, convert the documents to lowercase.

```
documentsLower = lower(documents);
```

Correct the spelling using the new edit distance searcher using same steps as before.

```
func = @(words,tdetails) correctSpelling(eds,words,tdetails);
updatedDocuments = docfun(func,documentsLower)
```

```
updatedDocuments =
  2×1 tokenizedDocument:

    8 tokens: here is some really badly written text .
    6 tokens: some more mistakes here too .
```

Here, the word "Some" in the original text is converted to "some" before being input to the spelling corrector. The corresponding word "some" is unaffected by the searcher as the word some occurs in the vocabulary.

Spelling Correction Function

The `correctSpelling` function corrects the spelling in a string array of words using the corresponding token details and an edit distance searcher. You can use this function with `docfun` to correct the spelling of multiple documents at once.

```
function words = correctSpelling(eds,words,tdetails)

% Get indices of misspelled words ignoring complex tokens.
idxVocabularyWords = ismember(tdetails.Token,eds.Vocabulary);

idxComplexTokens = ...
    tdetails.Type ~= "letters" & ...
    tdetails.Type ~= "other";
```

```
idxWordsToCheck = ...
    ~idxVocabularyWords & ...
    ~idxComplexTokens;

% Convert to numeric indices.
idxWordsToCheck = find(idxWordsToCheck);

% Find nearest words.
wordsToCheck = words(idxWordsToCheck);
idxNearestWords = knnsearch(eds,wordsToCheck);

% Find words with matches.
idxMatches = ~isnan(idxNearestWords);

% Get corrected words.
idxCorrectedWords = idxNearestWords(idxMatches);
correctedWords = eds.Vocabulary(idxCorrectedWords);

% Correct words.
idxToCorrect = idxWordsToCheck(idxMatches);
words(idxToCorrect) = correctedWords;

end
```

See Also

`correctSpelling` | `docfun` | `editDistance` | `editDistanceSearcher` | `knnsearch` | `tokenDetails` | `tokenizedDocument`

More About

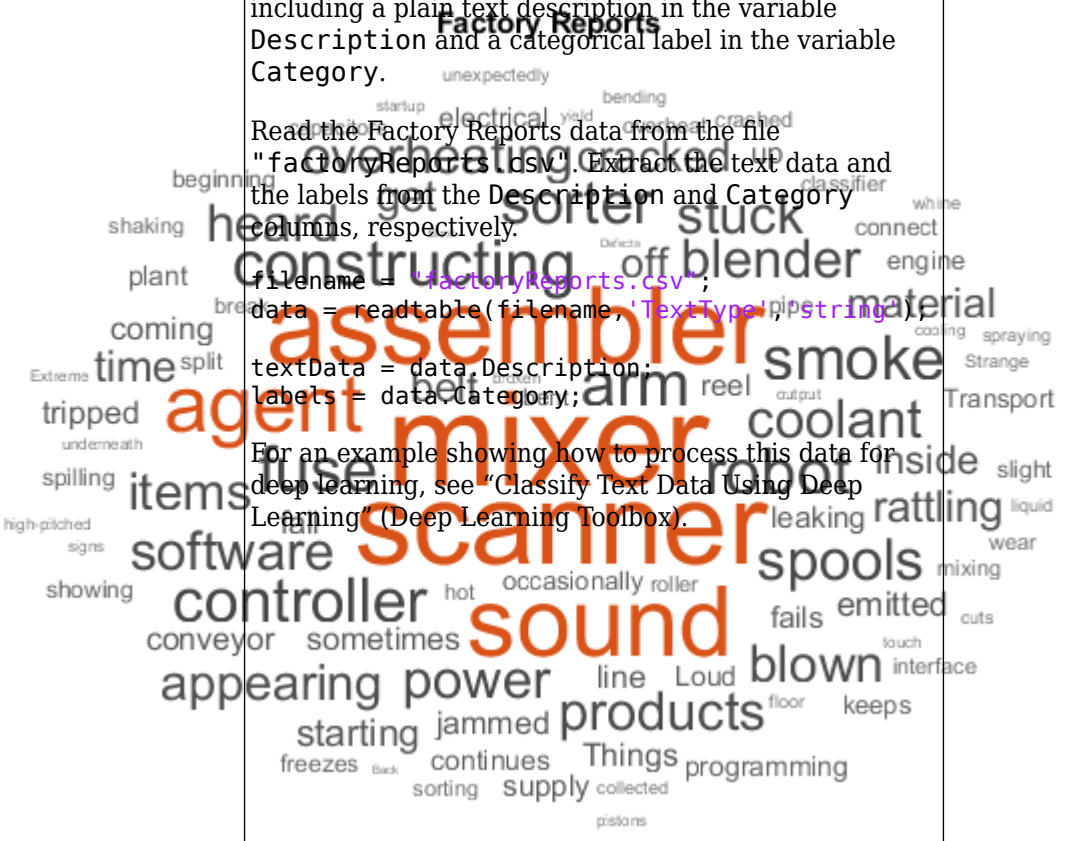
- “Correct Spelling in Documents” on page 1-21
- “Create Extension Dictionary for Spelling Correction” on page 1-23
- “Prepare Text Data for Analysis” on page 1-10
- “Create Simple Text Model for Classification” on page 2-2

Data Sets for Text Analytics

This page provides a list of different data sets that you can use to get started with text analytics applications.

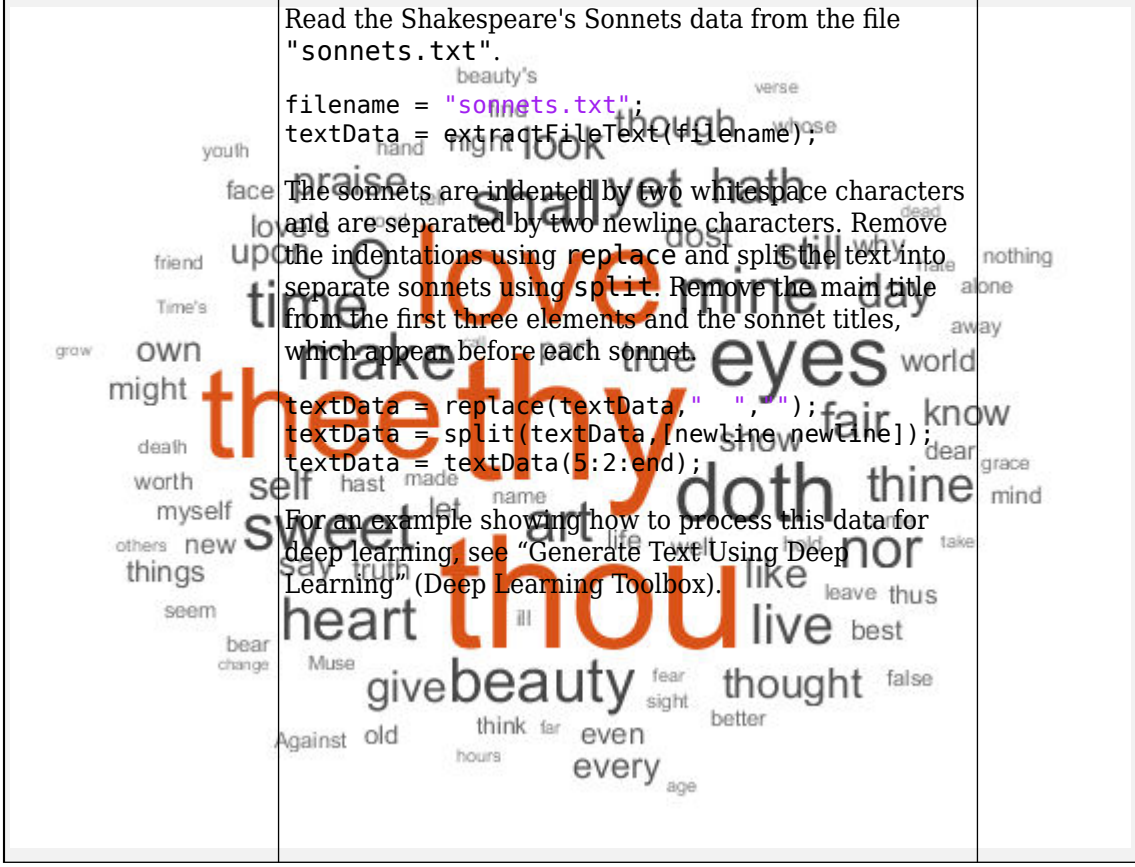
Data Set	Description	Task

Factory Reports	The Factory Reports data set is a table containing approximately 500 reports with various attributes including a plain text description in the variable `Description` and a categorical label in the variable `Category`.
	
	Read the Factory Reports data from the file `"factoryReports.csv"`. Extract the text data and the labels from the `Description` and `Category` columns, respectively.
	
	`filename = "factoryReports.csv";`
	`data = readtable(filename,'TextType','string');`
	
	`textData = data.Description;`
	`labels = data.Category;`
	
	For an example showing how to process this data for deep learning, see "Classify Text Data Using Deep Learning" (Deep Learning Toolbox).
		Text classification, topic modeling



Data Set	Description	Task

Shakespeare's Sonnets	The file `sonnets.txt` contains all of Shakespeare's sonnets in a single text file.
	
	Read the Shakespeare's Sonnets data from the file `"sonnets.txt"`.
	
	`filename = "sonnets.txt";`
	`textData = extractFileText(filename);`
	
	The sonnets are indented by two whitespace characters and are separated by two newline characters. Remove the indentations using `replace` and split the text into separate sonnets using `split`. Remove the main title from the first three elements and the sonnet titles, which appear before each sonnet.
	
	`textData = replace(textData,"  ","");`
	`textData = split(textData,[newline newline]);`
	`textData = textData(5:2:end);`
	
	For an example showing how to process this data for deep learning, see "Generate Text Using Deep Learning" (Deep Learning Toolbox).
		Topic modeling, text generation



Data Set	Description	Task

ArXiv Metadata	The ArXiv API allows you to access the metadata of scientific e-prints submitted to https://arxiv.org including the abstract and subject areas. For more information, see https://arxiv.org/help/api.
	
	Import a set of abstracts and category labels from math papers using the arXiV API.
	
	`url = "https://export.arxiv.org/oai2?verb=ListRecords" + ... "&set=math" + ... "&metadataPrefix=arXiv";`
	`options = weboptions('Timeout',160);`
	`code = webread(url,options);`
	
	For an example showing how to parse the returned XML code and import more records, see "Multilabel Text Classification Using Deep Learning" on page 2-94.
		Text classification, topic modeling

Data Set	Description	Task

Books from Project Gutenberg	You can download many books from Project Gutenberg. For example, download the text from Alice's Adventures in Wonderland by Lewis Carroll from https://www.gutenberg.org/files/11/11-h/11-h.htm using the `webread` function.
	
	`url = "https://www.gutenberg.org/files/11/11-h/11-h.htm";`
	`code = webread(url);`
	
	The HTML code contains the relevant text inside `<p>` (paragraph) elements. Extract the relevant text by parsing the HTML code using the `htmlTree` function and then finding all the elements with the element name `"p"`.
	
	`tree = htmlTree(code);`
	`selector = "p";`
	`subtrees = findElement(tree,selector);`
	
	Extract the text data from the HTML subtrees using the `extractHTMLText` function and remove the empty elements.
	
	`textData = extractHTMLText(subtrees);`
	`textData(textData == "") = [];`
	
	For an example showing how to process this data for deep learning, see "Word-By-Word Text Generation Using Deep Learning" on page 2-142.
		Topic modeling, text generation

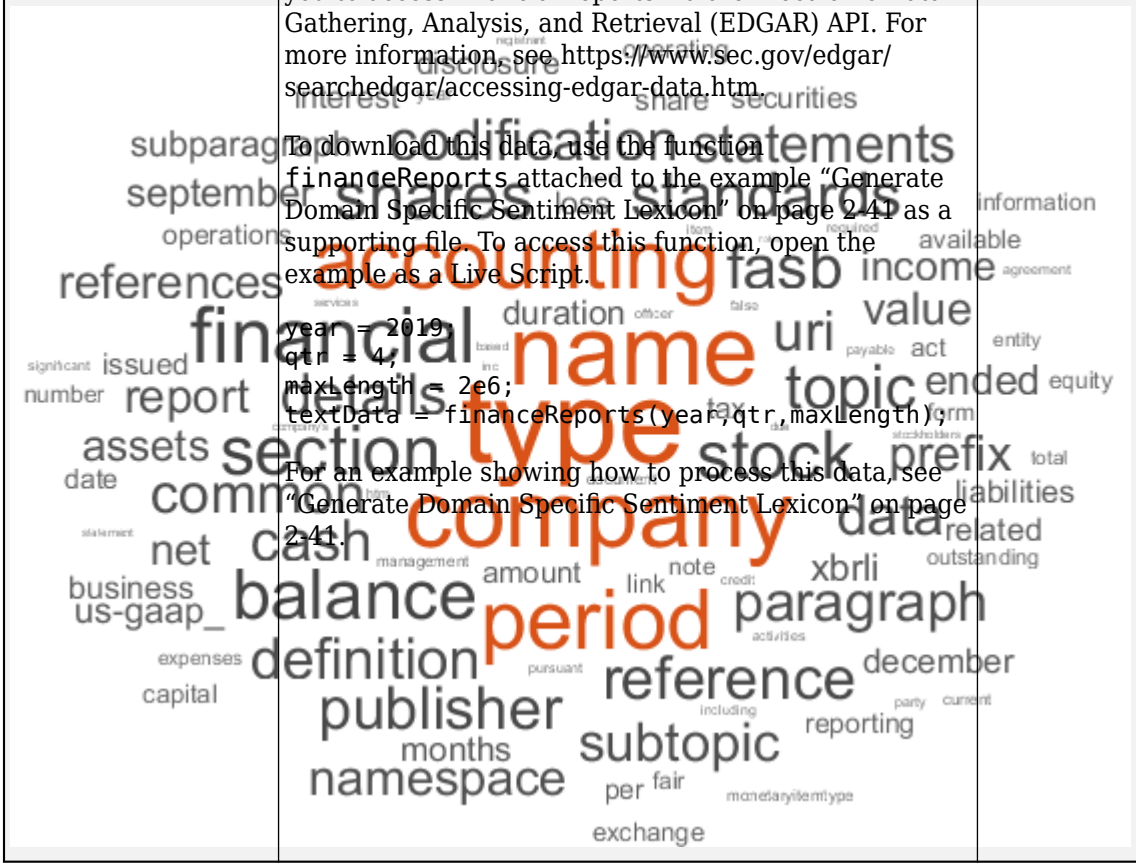
Data Set	Description	Task

Weekend updates	The file `weekendUpdates.xlsx` contains example social media status updates containing the hashtags "#weekend" and "#vacation".
	
	Extract the text data from the file `weekendUpdates.xlsx` using the `readtable` function and extract the text data from the variable `TextData`.
	
	`filename = "weekendUpdates.xlsx";`
	`tbl = readtable(filename,'TextType','string');`
	`textData = tbl.TextData;`
	
	For an example showing how to process this data, see "Analyze Sentiment in Text" on page 2-38.
		Sentiment analysis

Roman Numerals	The CSV file `"romanNumerals.csv"` contains the decimal numbers 1–1000 in the first column and the corresponding Roman numerals in the second column.
	
	Load the decimal-Roman numeral pairs from the CSV file `"romanNumerals.csv"`.
	
	`filename = fullfile("romanNumerals.csv");`
	`options = detectImportOptions(filename, ...`
	`    'TextType','string', ...`
	`    'ReadVariableNames',false);`
	`options.VariableNames = ["Source" "Target"];`
	`options.VariableTypes = ["string" "string"];`
	
	`data = readtable(filename,options);`
	
	For an example showing how to process this data for deep learning, see "Sequence-to-Sequence Translation Using Attention" on page 2-114.
		Sequence-to-sequence translation

Data Set	Description	Task

Finance Reports	The Securities and Exchange Commission (SEC) allows you to access financial reports via the Electronic Data Gathering, Analysis, and Retrieval (EDGAR) API. For more information, see https://www.sec.gov/edgar/searchedgar/accessing-edgar-data.htm.
	
	To download this data, use the function `financeReports` attached to the example "Generate Domain Specific Sentiment Lexicon" on page 2-41 as a supporting file. To access this function, open the example as a Live Script.
	
	`year = 2019;`
	`qtr = 4;`
	`maxLength = 2e6;`
	`textData = financeReports(year,qtr,maxLength);`
	
	For an example showing how to process this data, see "Generate Domain Specific Sentiment Lexicon" on page 2-41.
		Sentiment analysis



See Also

More About

- "Extract Text Data from Files" on page 1-2
- "Parse HTML and Extract Text Content" on page 1-17
- "Prepare Text Data for Analysis" on page 1-10
- "Analyze Text Data Containing Emojis" on page 2-32
- "Create Simple Text Model for Classification" on page 2-2
- "Classify Text Data Using Deep Learning" on page 2-65
- "Analyze Text Data Using Topic Models" on page 2-13
- "Analyze Sentiment in Text" on page 2-38
- "Sequence-to-Sequence Translation Using Attention" on page 2-114
- "Generate Text Using Deep Learning" (Deep Learning Toolbox)

Modeling and Prediction

Create Simple Text Model for Classification

This example shows how to train a simple text classifier on word frequency counts using a bag-of-words model.

You can create a simple classification model which uses word frequency counts as predictors. This example trains a simple classification model to predict the category of factory reports using text descriptions.

Load and Extract Text Data

Load the example data. The file `factoryReports.csv` contains factory reports, including a text description and categorical labels for each report.

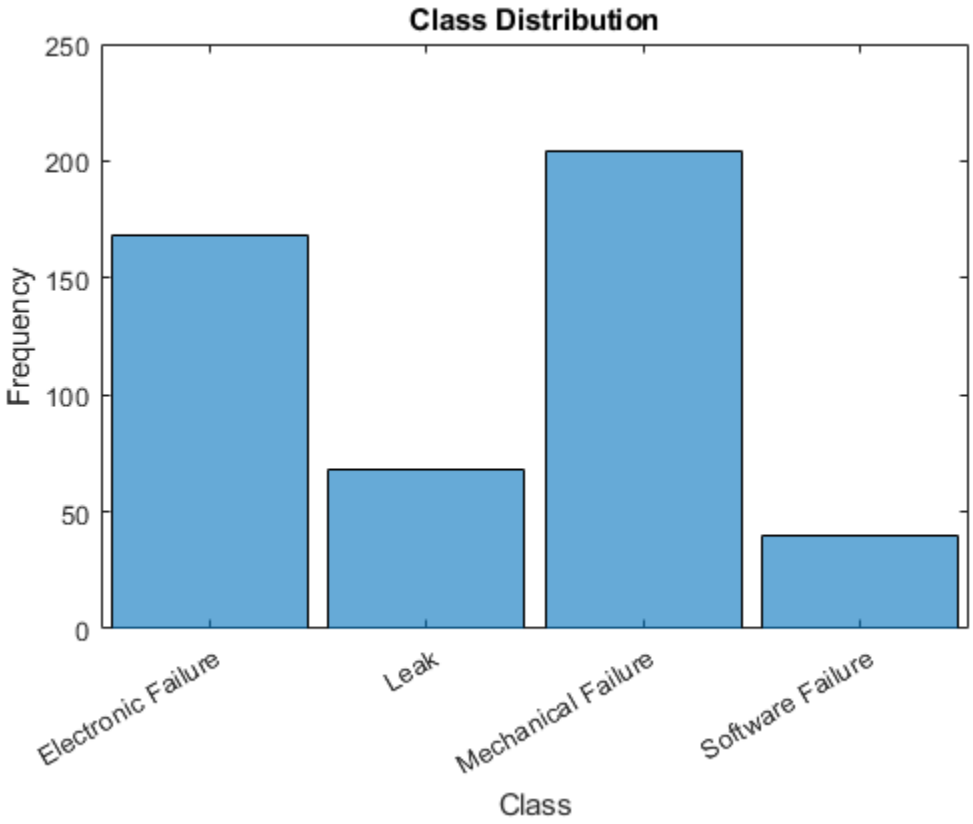
```
filename = "factoryReports.csv";
data = readtable(filename,'TextType','string');
head(data)
```

```
ans=8×5 table
                                Description                                 
    ____________________________________________________________________    

    "Items are occasionally getting stuck in the scanner spools."           
    "Loud rattling and banging sounds are coming from assembler pistons."   
    "There are cuts to the power when starting the plant."                  
    "Fried capacitors in the assembler."                                    
    "Mixer tripped the fuses."                                              
    "Burst pipe in the constructing agent is spraying coolant."             
    "A fuse is blown in the mixer."                                         
    "Things continue to tumble off of the belt."                            
```

```
                Category         
    ____________________    

    "Mechanical Failure"
    "Mechanical Failure"
    "Electronic Failure"
    "Electronic Failure"
    "Electronic Failure"
    "Leak"              
    "Electronic Failure"
    "Mechanical Failure"
```

Convert the labels in the `Category` column of the table to categorical and view the distribution of the classes in the data using a histogram.

```
data.Category = categorical(data.Category);
figure
histogram(data.Category)
xlabel("Class")
ylabel("Frequency")
title("Class Distribution")
```



Partition the data into a training partition and a held-out test set. Specify the holdout percentage to be 10%.

```
cvp = cvpartition(data.Category,'Holdout',0.1);
dataTrain = data(cvp.training,:);
dataTest = data(cvp.test,:);
```

Extract the text data and labels from the tables.

```
textDataTrain = dataTrain.Description;
textDataTest = dataTest.Description;
YTrain = dataTrain.Category;
YTest = dataTest.Category;
```

Prepare Text Data for Analysis

Create a function which tokenizes and preprocesses the text data so it can be used for analysis. The function `preprocessText`, performs the following steps in order:

1. Tokenize the text using `tokenizedDocument`.
2. Remove a list of stop words (such as "and", "of", and "the") using `removeStopWords`.
3. Lemmatize the words using `normalizeWords`.
4. Erase punctuation using `erasePunctuation`.
5. Remove words with 2 or fewer characters using `removeShortWords`.
6. Remove words with 15 or more characters using `removeLongWords`.

Use the example preprocessing function `preprocessText` to prepare the text data.

```
documents = preprocessText(textDataTrain);
documents(1:5)
```

```
ans =
  5×1 tokenizedDocument:

    6 tokens: items occasionally get stuck scanner spool
    7 tokens: loud rattle bang sound come assembler piston
    4 tokens: cut power start plant
    3 tokens: fry capacitor assembler
    3 tokens: mixer trip fuse
```

Create a bag-of-words model from the tokenized documents.

```
bag = bagOfWords(documents)
```

```
bag =
  bagOfWords with properties:

          Counts: [432×336 double]
      Vocabulary: [1×336 string]
        NumWords: 336
    NumDocuments: 432
```

Remove words from the bag-of-words model that do not appear more than two times in total. Remove any documents containing no words from the bag-of-words model, and remove the corresponding entries in labels.

```
bag = removeInfrequentWords(bag,2);
[bag,idx] = removeEmptyDocuments(bag);
YTrain(idx) = [];
bag
```

```
bag =
  bagOfWords with properties:

          Counts: [432×155 double]
      Vocabulary: [1×155 string]
        NumWords: 155
    NumDocuments: 432
```

Train Supervised Classifier

Train a supervised classification model using the word frequency counts from the bag-of-words model and the labels.

Train a multiclass linear classification model using `fitcecoc`. Specify the `Counts` property of the bag-of-words model to be the predictors, and the event type labels to be the response. Specify the learners to be linear. These learners support sparse data input.

```
XTrain = bag.Counts;
mdl = fitcecoc(XTrain,YTrain,'Learners','linear')
```

```
mdl =
  CompactClassificationECOC
```

```
      ResponseName: 'Y'
        ClassNames: [Electronic Failure    Leak    Mechanical Failure    Software Failure]
    ScoreTransform: 'none'
    BinaryLearners: {6×1 cell}
      CodingMatrix: [4×6 double]


  Properties, Methods
```

For a better fit, you can try specifying different parameters of the linear learners. For more information on linear classification learner templates, see `templateLinear`.

Test Classifier

Predict the labels of the test data using the trained model and calculate the classification accuracy. The classification accuracy is the proportion of the labels that the model predicts correctly.

Preprocess the test data using the same preprocessing steps as the training data. Encode the resulting test documents as a matrix of word frequency counts according to the bag-of-words model.

```
documentsTest = preprocessText(textDataTest);
XTest = encode(bag,documentsTest);
```

Predict the labels of the test data using the trained model and calculate the classification accuracy.

```
YPred = predict(mdl,XTest);
acc = sum(YPred == YTest)/numel(YTest)
```

```
acc = 0.8542
```

Predict Using New Data

Classify the event type of new factory reports. Create a string array containing the new factory reports.

```
str = [
    "Coolant is pooling underneath sorter."
    "Sorter blows fuses at start up."
    "There are some very loud rattling sounds coming from the assembler."];
documentsNew = preprocessText(str);
XNew = encode(bag,documentsNew);
labelsNew = predict(mdl,XNew)
```

```
labelsNew = 3×1 categorical
     Leak
     Electronic Failure
     Mechanical Failure
```

Example Preprocessing Function

The function `preprocessText`, performs the following steps in order:

1. Tokenize the text using `tokenizedDocument`.
2. Remove a list of stop words (such as "and", "of", and "the") using `removeStopWords`.
3. Lemmatize the words using `normalizeWords`.

4 Erase punctuation using `erasePunctuation`.

5 Remove words with 2 or fewer characters using `removeShortWords`.

6 Remove words with 15 or more characters using `removeLongWords`.

```
function documents = preprocessText(textData)

% Tokenize the text.
documents = tokenizedDocument(textData);

% Remove a list of stop words then lemmatize the words. To improve
% lemmatization, first use addPartOfSpeechDetails.
documents = addPartOfSpeechDetails(documents);
documents = removeStopWords(documents);
documents = normalizeWords(documents,'Style','lemma');

% Erase punctuation.
documents = erasePunctuation(documents);

% Remove words with 2 or fewer characters, and words with 15 or more
% characters.
documents = removeShortWords(documents,2);
documents = removeLongWords(documents,15);

end
```

See Also

`addPartOfSpeechDetails` | `bagOfWords` | `encode` | `erasePunctuation` | `normalizeWords` | `removeLongWords` | `removeShortWords` | `removeStopWords` | `tokenizedDocument` | `wordcloud`

Related Examples

- “Analyze Text Data Using Topic Models” on page 2-13
- “Analyze Text Data Using Multiword Phrases” on page 2-7
- “Analyze Text Data Containing Emojis” on page 2-32
- “Train a Sentiment Classifier” on page 2-51
- “Classify Text Data Using Deep Learning” on page 2-65
- “Generate Text Using Deep Learning” (Deep Learning Toolbox)

Analyze Text Data Using Multiword Phrases

This example shows how to analyze text using n-gram frequency counts.

An n-gram is a tuple of n consecutive words. For example, a bigram (the case when $n = 2$) is a pair of consecutive words such as "heavy rainfall". A unigram (the case when $n = 1$) is a single word. A bag-of-n-grams model records the number of times that different n-grams appear in document collections.

Using a bag-of-n-grams model, you can retain more information on word ordering in the original text data. For example, a bag-of-n-grams model is better suited for capturing short phrases which appear in the text, such as "heavy rainfall" and "thunderstorm winds".

To create a bag-of-n-grams model, use `bagOfNgrams`. You can input `bagOfNgrams` objects into other Text Analytics Toolbox functions such as `wordcloud` and `fitlda`.

Load and Extract Text Data

Load the example data. The file `factoryReports.csv` contains factory reports, including a text description and categorical labels for each event. Remove the rows with empty reports.

```
filename = "factoryReports.csv";
data = readtable(filename,'TextType','String');
```

Extract the text data from the table and view the first few reports.

```
textData = data.Description;
textData(1:5)
```

```
ans = 5×1 string
    "Items are occasionally getting stuck in the scanner spools."
    "Loud rattling and banging sounds are coming from assembler pistons."
    "There are cuts to the power when starting the plant."
    "Fried capacitors in the assembler."
    "Mixer tripped the fuses."
```

Prepare Text Data for Analysis

Create a function which tokenizes and preprocesses the text data so it can be used for analysis. The function `preprocessText` listed at the end of the example, performs the following steps:

1. Convert the text data to lowercase using `lower`.
2. Tokenize the text using `tokenizedDocument`.
3. Erase punctuation using `erasePunctuation`.
4. Remove a list of stop words (such as "and", "of", and "the") using `removeStopWords`.
5. Remove words with 2 or fewer characters using `removeShortWords`.
6. Remove words with 15 or more characters using `removeLongWords`.
7. Lemmatize the words using `normalizeWords`.

Use the example preprocessing function `preprocessTest` to prepare the text data.

```
documents = preprocessText(textData);
documents(1:5)
```

```
ans =
  5×1 tokenizedDocument:

    6 tokens: item occasionally get stuck scanner spool
    7 tokens: loud rattle bang sound come assembler piston
    4 tokens: cut power start plant
    3 tokens: fry capacitor assembler
    3 tokens: mixer trip fuse
```

Create Word Cloud of Bigrams

Create a word cloud of bigrams by first creating a bag-of-n-grams model using `bagOfNgrams`, and then inputting the model to `wordcloud`.

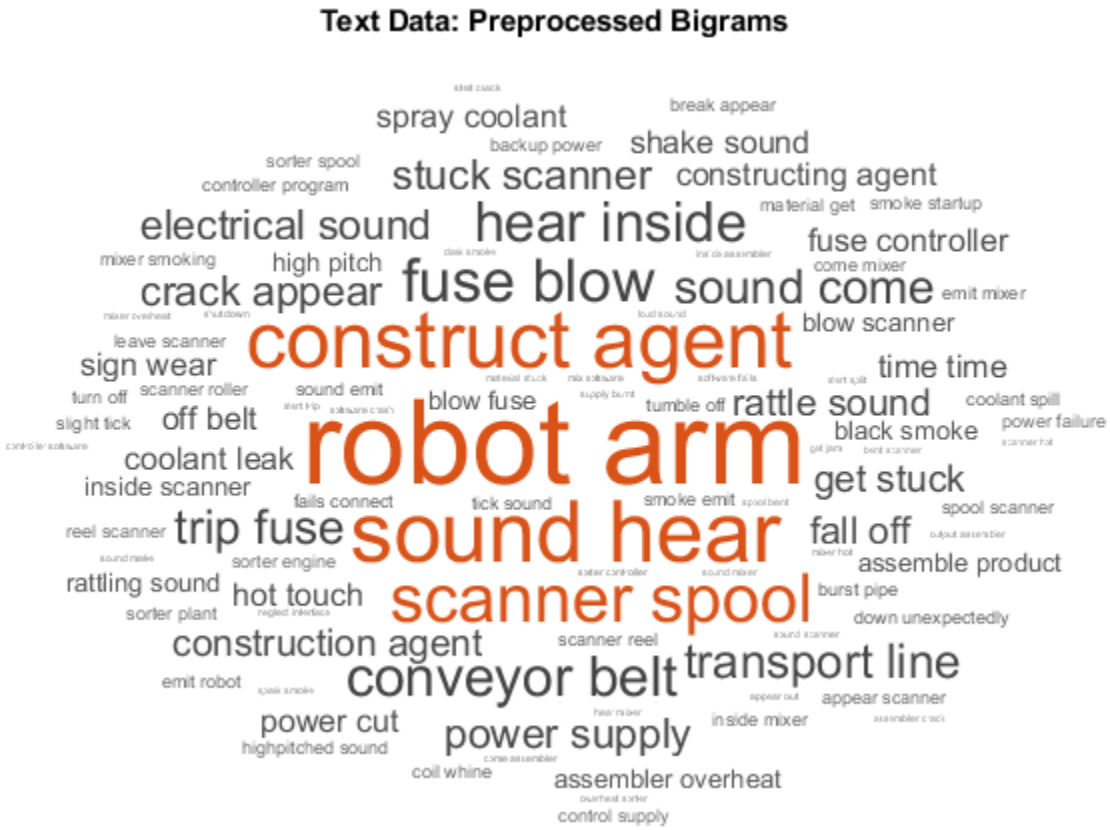
To count the n-grams of length 2 (bigrams), use `bagOfNgrams` with the default options.

```
bag = bagOfNgrams(documents)
```

```
bag =
  bagOfNgrams with properties:

          Counts: [480×941 double]
      Vocabulary: [1×351 string]
          Ngrams: [941×2 string]
    NgramLengths: 2
       NumNgrams: 941
    NumDocuments: 480
```

Visualize the bag-of-n-grams model using a word cloud.

```
figure
wordcloud(bag);
title("Text Data: Preprocessed Bigrams")
```



Fit Topic Model to Bag-of-N-Grams

A Latent Dirichlet Allocation (LDA) model is a topic model which discovers underlying topics in a collection of documents and infers the word probabilities in topics.

Create an LDA topic model with 10 topics using `fitlda`. The function fits an LDA model by treating the n-grams as single words.

```
mdl = fitlda(bag,10,'Verbose',0);
```

Visualize the first four topics as word clouds.

```
figure
for i = 1:4
    subplot(2,2,i)
    wordcloud(mdl,i);
    title("LDA Topic " + i)
end
```

The word clouds highlight commonly co-occurring bigrams in the LDA topics. The function plots the bigrams with sizes according to their probabilities for the specified LDA topics.

Analyze Text Using Longer Phrases

To analyze text using longer phrases, specify the `'NGramLengths'` option in `bagOfNgrams` to be a larger value.

When working with longer phrases, it can be useful to keep stop words in the model. For example, to detect the phrase "is not happy", keep the stop words "is" and "not" in the model.

Preprocess the text. Erase the punctuation using `erasePunctuation`, and tokenize using `tokenizedDocument`.

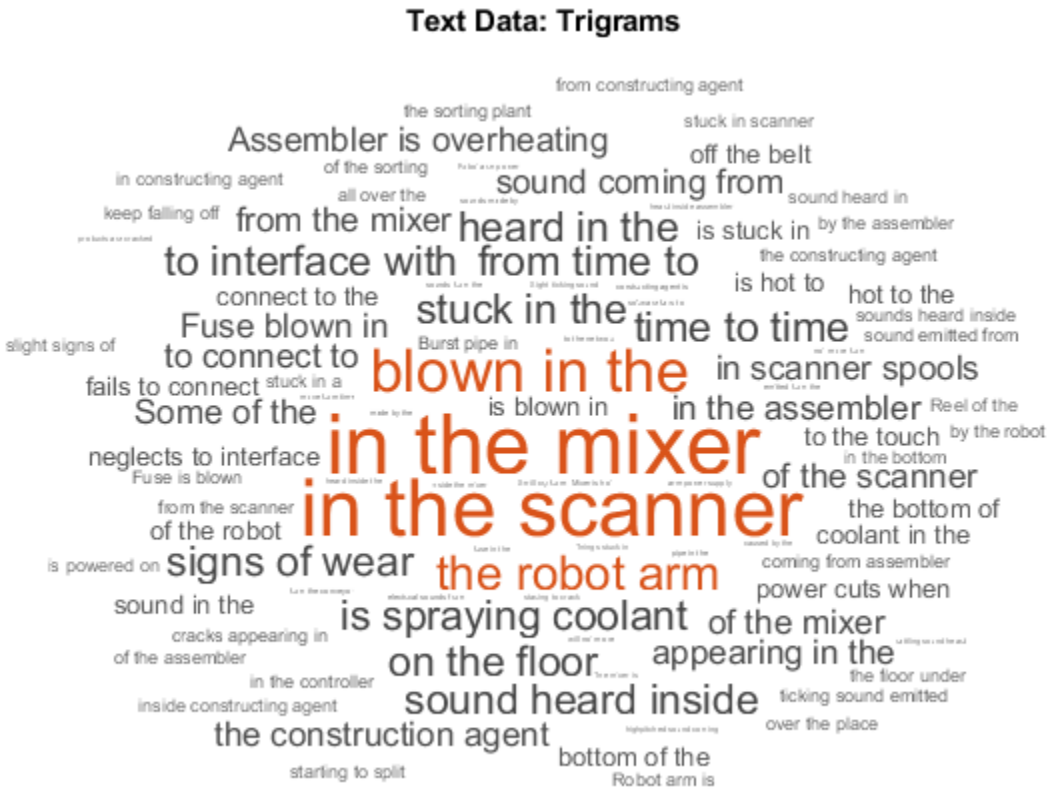
```
cleanTextData = erasePunctuation(textData);
documents = tokenizedDocument(cleanTextData);
```

To count the n-grams of length 3 (trigrams), use `bagOfNgrams` and specify `'NGramLengths'` to be 3.

```
bag = bagOfNgrams(documents,'NGramLengths',3);
```

Visualize the bag-of-n-grams model using a word cloud. The word cloud of trigrams better shows the context of the individual words.

```
figure
wordcloud(bag);
title("Text Data: Trigrams")
```



View the top 10 trigrams and their frequency counts using `topkngrams`.

```
tbl = topkngrams(bag,10)
```

```
tbl=10×3 table
                    Ngram                     Count    NgramLength
    ______________________________________    _____    ___________

    "in"       "the"          "mixer"          14           3
    "in"       "the"          "scanner"        13           3
    "blown"    "in"           "the"             9           3
    "the"      "robot"        "arm"             7           3
    "stuck"    "in"           "the"             6           3
    "is"       "spraying"     "coolant"         6           3
    "from"     "time"         "to"              6           3
    "time"     "to"           "time"            6           3
    "heard"    "in"           "the"             6           3
    "on"       "the"          "floor"           6           3
```

Example Preprocessing Function

The function `preprocessText` performs the following steps in order:

1. Convert the text data to lowercase using `lower`.
2. Tokenize the text using `tokenizedDocument`.

3 Erase punctuation using `erasePunctuation`.

4 Remove a list of stop words (such as "and", "of", and "the") using `removeStopWords`.

5 Remove words with 2 or fewer characters using `removeShortWords`.

6 Remove words with 15 or more characters using `removeLongWords`.

7 Lemmatize the words using `normalizeWords`.

```
function documents = preprocessText(textData)

% Convert the text data to lowercase.
cleanTextData = lower(textData);

% Tokenize the text.
documents = tokenizedDocument(cleanTextData);

% Erase punctuation.
documents = erasePunctuation(documents);

% Remove a list of stop words.
documents = removeStopWords(documents);

% Remove words with 2 or fewer characters, and words with 15 or greater
% characters.
documents = removeShortWords(documents,2);
documents = removeLongWords(documents,15);

% Lemmatize the words.
documents = addPartOfSpeechDetails(documents);
documents = normalizeWords(documents,'Style','lemma');

end
```

See Also

`addPartOfSpeechDetails` | `bagOfNgrams` | `bagOfWords` | `erasePunctuation` | `fitlda` | `ldaModel` | `normalizeWords` | `removeLongWords` | `removeShortWords` | `removeStopWords` | `tokenizedDocument` | `topkngrams` | `wordcloud`

Related Examples

- “Create Simple Text Model for Classification” on page 2-2
- “Analyze Text Data Containing Emojis” on page 2-32
- “Analyze Text Data Using Topic Models” on page 2-13
- “Train a Sentiment Classifier” on page 2-51
- “Classify Text Data Using Deep Learning” on page 2-65
- “Generate Text Using Deep Learning” (Deep Learning Toolbox)

Analyze Text Data Using Topic Models

This example shows how to use the Latent Dirichlet Allocation (LDA) topic model to analyze text data.

A Latent Dirichlet Allocation (LDA) model is a topic model which discovers underlying topics in a collection of documents and infers the word probabilities in topics.

Load and Extract Text Data

Load the example data. The file `factoryReports.csv` contains factory reports, including a text description and categorical labels for each event.

```
data = readtable("factoryReports.csv",'TextType','string');
head(data)
```

```
ans=8×5 table
                                Description                                 
    ____________________________________________________________________    

    "Items are occasionally getting stuck in the scanner spools."           
    "Loud rattling and banging sounds are coming from assembler pistons."   
    "There are cuts to the power when starting the plant."                  
    "Fried capacitors in the assembler."                                    
    "Mixer tripped the fuses."                                              
    "Burst pipe in the constructing agent is spraying coolant."             
    "A fuse is blown in the mixer."                                         
    "Things continue to tumble off of the belt."                            
```

```
                  Category          
    ____________________________    

    "Mechanical Failure"
    "Mechanical Failure"
    "Electronic Failure"
    "Electronic Failure"
    "Electronic Failure"
    "Leak"
    "Electronic Failure"
    "Mechanical Failure"
```

Extract the text data from the field `Description`.

```
textData = data.Description;
textData(1:10)
```

```
ans = 10×1 string
    "Items are occasionally getting stuck in the scanner spools."
    "Loud rattling and banging sounds are coming from assembler pistons."
    "There are cuts to the power when starting the plant."
    "Fried capacitors in the assembler."
    "Mixer tripped the fuses."
    "Burst pipe in the constructing agent is spraying coolant."
    "A fuse is blown in the mixer."
    "Things continue to tumble off of the belt."
    "Falling items from the conveyor belt."
    "The scanner reel is split, it will soon begin to curve."
```

Prepare Text Data for Analysis

Create a function which tokenizes and preprocesses the text data so it can be used for analysis. The function `preprocessText`, listed at the end of the example, performs the following steps in order:

1. Tokenize the text using `tokenizedDocument`.
2. Lemmatize the words using `normalizeWords`.
3. Erase punctuation using `erasePunctuation`.
4. Remove a list of stop words (such as "and", "of", and "the") using `removeStopWords`.

5 Remove words with 2 or fewer characters using `removeShortWords`.

6 Remove words with 15 or more characters using `removeLongWords`.

Use the preprocessing function `preprocessText` to prepare the text data.

```
documents = preprocessText(textData);
documents(1:5)
```

```
ans =
  5×1 tokenizedDocument:

    6 tokens: items occasionally get stuck scanner spool
    7 tokens: loud rattle bang sound come assembler piston
    4 tokens: cut power start plant
    3 tokens: fry capacitor assembler
    3 tokens: mixer trip fuse
```

Create a bag-of-words model from the tokenized documents.

```
bag = bagOfWords(documents)
```

```
bag =
  bagOfWords with properties:

          Counts: [480×351 double]
      Vocabulary: [1×351 string]
        NumWords: 351
    NumDocuments: 480
```

Remove words from the bag-of-words model that have do not appear more than two times in total. Remove any documents containing no words from the bag-of-words model.

```
bag = removeInfrequentWords(bag,2);
bag = removeEmptyDocuments(bag)
```

```
bag =
  bagOfWords with properties:

          Counts: [480×162 double]
      Vocabulary: [1×162 string]
        NumWords: 162
    NumDocuments: 480
```

Fit LDA Model

Fit an LDA model with 7 topics. For an example showing how to choose the number of topics, see “Choose Number of Topics for LDA Model” on page 2-19. To suppress verbose output, set `'Verbose'` to 0.

```
numTopics = 7;
mdl = fitlda(bag,numTopics,'Verbose',0);
```

If you have a large dataset, then the stochastic approximate variational Bayes solver is usually better suited as it can fit a good model in fewer passes of the data. The default solver for `fitlda` (collapsed Gibbs sampling) can be more accurate at the cost of taking longer to run. To use stochastic

approximate variational Bayes, set the `'Solver'` option to `'savb'`. For an example showing how to compare LDA solvers, see “Compare LDA Solvers” on page 2-23.

Visualize Topics Using Word Clouds

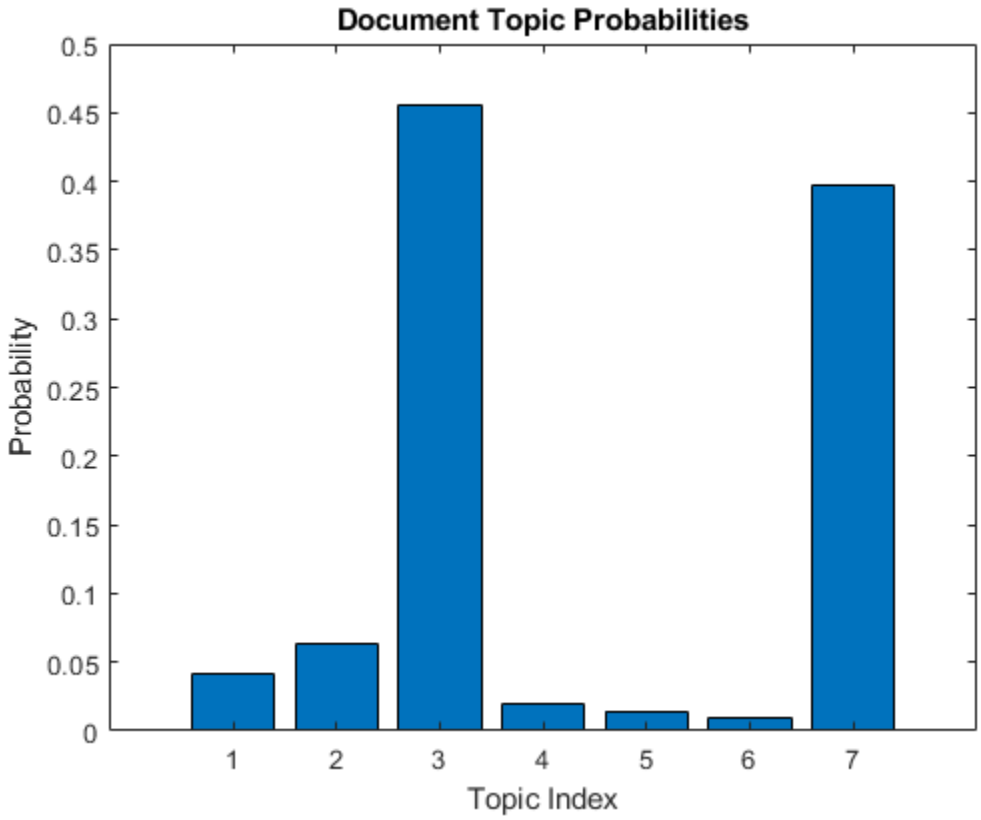
You can use word clouds to view the words with the highest probabilities in each topic. Visualize the first four topics using word clouds.

```
figure;
for topicIdx = 1:4
    subplot(2,2,topicIdx)
    wordcloud(mdl,topicIdx);
    title("Topic " + topicIdx)
end
```

View Mixtures of Topics in Documents

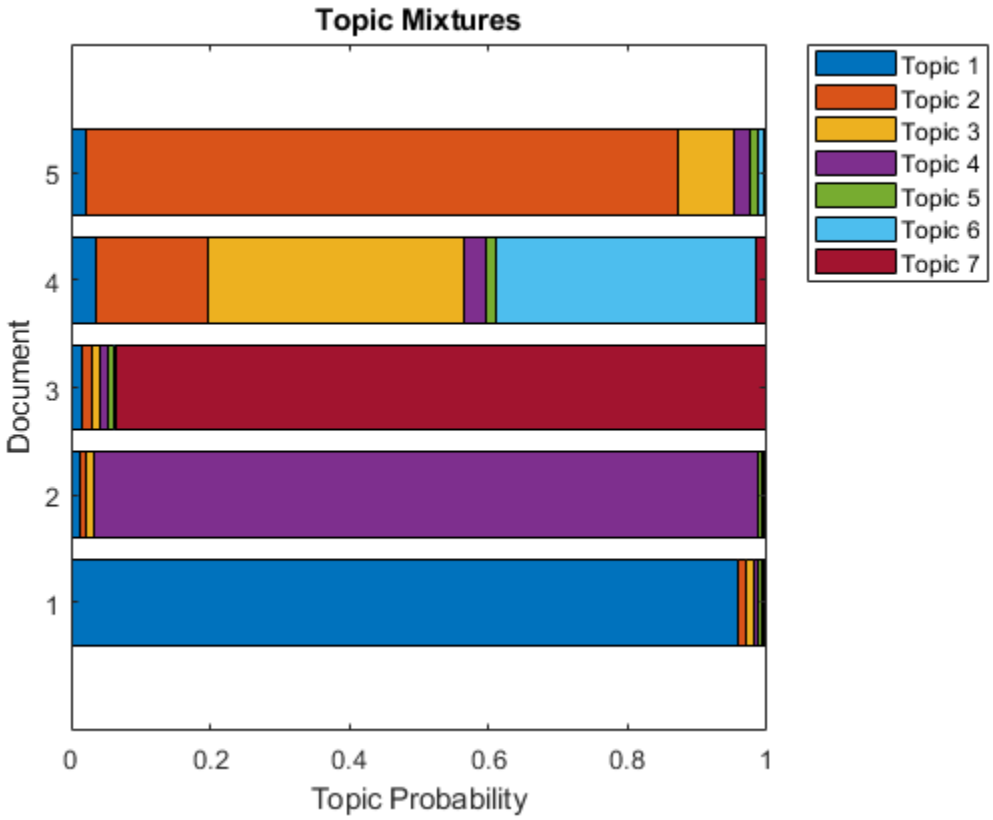
Use `transform` to transform the documents into vectors of topic probabilities.

```
newDocument = tokenizedDocument("Coolant is pooling underneath sorter.");
topicMixture = transform(mdl,newDocument);

figure
bar(topicMixture)
xlabel("Topic Index")
ylabel("Probability")
title("Document Topic Probabilities")
```



Visualize multiple topic mixtures using stacked bar charts. Visualize the topic mixtures of the first 5 input documents.

```
figure
topicMixtures = transform(mdl,documents(1:5));
barh(topicMixtures(1:5,:),'stacked')
xlim([0 1])
title("Topic Mixtures")
xlabel("Topic Probability")
ylabel("Document")
legend("Topic " + string(1:numTopics),'Location','northeastoutside')
```



Preprocessing Function

The function `preprocessText`, performs the following steps in order:

1. Tokenize the text using `tokenizedDocument`.
2. Lemmatize the words using `normalizeWords`.
3. Erase punctuation using `erasePunctuation`.
4. Remove a list of stop words (such as "and", "of", and "the") using `removeStopWords`.
5. Remove words with 2 or fewer characters using `removeShortWords`.
6. Remove words with 15 or more characters using `removeLongWords`.

```
function documents = preprocessText(textData)

% Tokenize the text.
documents = tokenizedDocument(textData);

% Lemmatize the words.
documents = addPartOfSpeechDetails(documents);
documents = normalizeWords(documents,'Style','lemma');

% Erase punctuation.
documents = erasePunctuation(documents);

% Remove a list of stop words.
documents = removeStopWords(documents);
```

```
% Remove words with 2 or fewer characters, and words with 15 or greater
% characters.
documents = removeShortWords(documents,2);
documents = removeLongWords(documents,15);

end
```

See Also

`addPartOfSpeechDetails` | `bagOfWords` | `fitlda` | `ldaModel` | `removeEmptyDocuments` | `removeInfrequentWords` | `removeStopWords` | `tokenizedDocument` | `transform` | `wordcloud`

Related Examples

- “Create Simple Text Model for Classification” on page 2-2
- “Analyze Text Data Containing Emojis” on page 2-32
- “Analyze Text Data Using Multiword Phrases” on page 2-7
- “Train a Sentiment Classifier” on page 2-51
- “Classify Text Data Using Deep Learning” on page 2-65
- “Generate Text Using Deep Learning” (Deep Learning Toolbox)

Choose Number of Topics for LDA Model

This example shows how to decide on a suitable number of topics for a latent Dirichlet allocation (LDA) model.

To decide on a suitable number of topics, you can compare the goodness-of-fit of LDA models fit with varying numbers of topics. You can evaluate the goodness-of-fit of an LDA model by calculating the perplexity of a held-out set of documents. The perplexity indicates how well the model describes a set of documents. A lower perplexity suggests a better fit.

Extract and Preprocess Text Data

Load the example data. The file `factoryReports.csv` contains factory reports, including a text description and categorical labels for each event. Extract the text data from the field `Description`.

```
filename = "factoryReports.csv";
data = readtable(filename,'TextType','string');
textData = data.Description;
```

Tokenize and preprocess the text data using the function `preprocessText` which is listed at the end of this example.

```
documents = preprocessText(textData);
documents(1:5)
```

```
ans =
  5×1 tokenizedDocument:

    6 tokens: item occasionally get stuck scanner spool
    7 tokens: loud rattle bang sound come assembler piston
    4 tokens: cut power start plant
    3 tokens: fry capacitor assembler
    3 tokens: mixer trip fuse
```

Set aside 10% of the documents at random for validation.

```
numDocuments = numel(documents);
cvp = cvpartition(numDocuments,'HoldOut',0.1);
documentsTrain = documents(cvp.training);
documentsValidation = documents(cvp.test);
```

Create a bag-of-words model from the training documents. Remove the words that do not appear more than two times in total. Remove any documents containing no words.

```
bag = bagOfWords(documentsTrain);
bag = removeInfrequentWords(bag,2);
bag = removeEmptyDocuments(bag);
```

Choose Number of Topics

The goal is to choose a number of topics that minimize the perplexity compared to other numbers of topics. This is not the only consideration: models fit with larger numbers of topics may take longer to converge. To see the effects of the tradeoff, calculate both goodness-of-fit and the fitting time. If the optimal number of topics is high, then you might want to choose a lower value to speed up the fitting process.

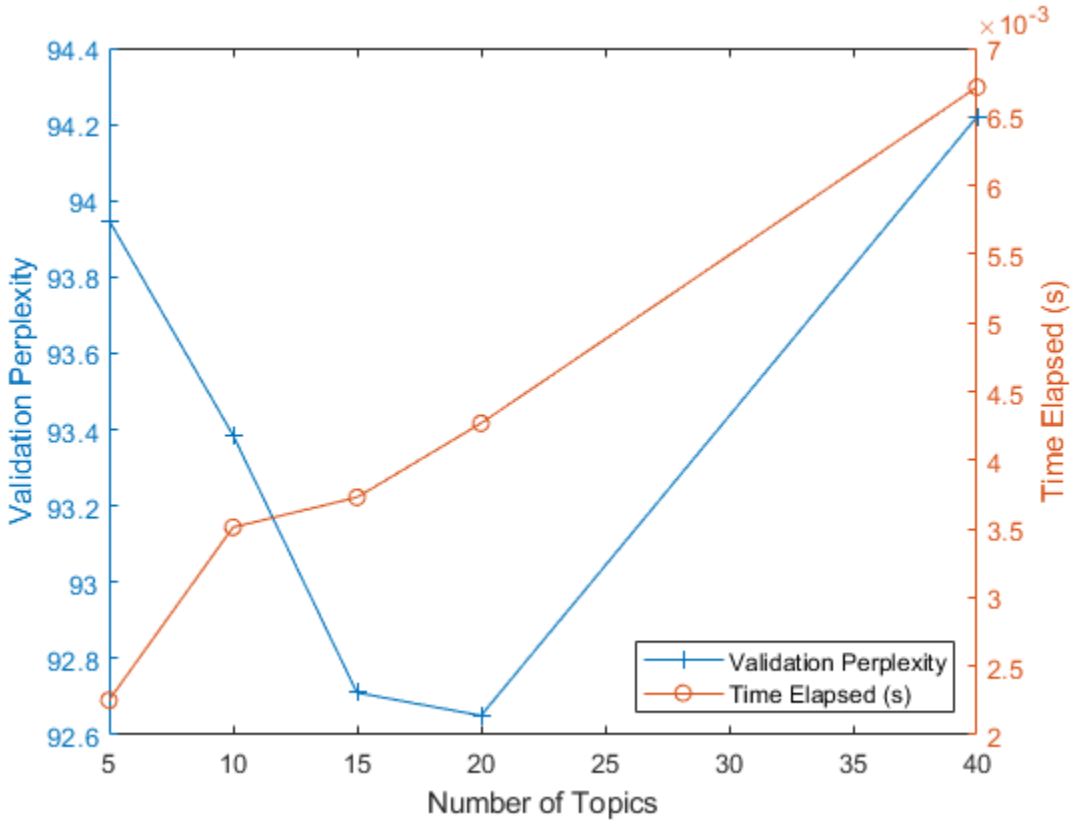
Fit some LDA models for a range of values for the number of topics. Compare the fitting time and the perplexity of each model on the held-out set of test documents. The perplexity is the second output to the `logp` function. To obtain the second output without assigning the first output to anything, use the ~ symbol. The fitting time is the `TimeSinceStart` value for the last iteration. This value is in the `History` struct of the `FitInfo` property of the LDA model.

For a quicker fit, specify `'Solver'` to be `'savb'`. To suppress verbose output, set `'Verbose'` to `0`. This may take a few minutes to run.

```
numTopicsRange = [5 10 15 20 40];
for i = 1:numel(numTopicsRange)
    numTopics = numTopicsRange(i);

    mdl = fitlda(bag,numTopics, ...
        'Solver','savb', ...
        'Verbose',0);

    [~,validationPerplexity(i)] = logp(mdl,documentsValidation);
    timeElapsed(i) = mdl.FitInfo.History.TimeSinceStart(end);
end
```

Show the perplexity and elapsed time for each number of topics in a plot. Plot the perplexity on the left axis and the time elapsed on the right axis.

```
figure
yyaxis left
plot(numTopicsRange,validationPerplexity,'+-')
ylabel("Validation Perplexity")

yyaxis right
plot(numTopicsRange,timeElapsed,'o-')
ylabel("Time Elapsed (s)")

legend(["Validation Perplexity" "Time Elapsed (s)"],'Location','southeast')
xlabel("Number of Topics")
```



The plot suggests that fitting a model with 10–20 topics may be a good choice. The perplexity is low compared with the models with different numbers of topics. With this solver, the elapsed time for this many topics is also reasonable. With different solvers, you may find that increasing the number of topics can lead to a better fit, but fitting the model takes longer to converge.

Example Preprocessing Function

The function `preprocessText`, performs the following steps in order:

1. Convert the text data to lowercase using `lower`.
2. Tokenize the text using `tokenizedDocument`.
3. Erase punctuation using `erasePunctuation`.
4. Remove a list of stop words (such as "and", "of", and "the") using `removeStopWords`.
5. Remove words with 2 or fewer characters using `removeShortWords`.
6. Remove words with 15 or more characters using `removeLongWords`.
7. Lemmatize the words using `normalizeWords`.

```
function documents = preprocessText(textData)

% Convert the text data to lowercase.
cleanTextData = lower(textData);

% Tokenize the text.
documents = tokenizedDocument(cleanTextData);
```

```
% Erase punctuation.
documents = erasePunctuation(documents);

% Remove a list of stop words.
documents = removeStopWords(documents);

% Remove words with 2 or fewer characters, and words with 15 or greater
% characters.
documents = removeShortWords(documents,2);
documents = removeLongWords(documents,15);

% Lemmatize the words.
documents = addPartOfSpeechDetails(documents);
documents = normalizeWords(documents,'Style','lemma');
end
```

See Also

`addPartOfSpeechDetails` | `bagOfWords` | `bagOfWords` | `erasePunctuation` | `fitlda` | `ldaModel` | `logp` | `normalizeWords` | `removeEmptyDocuments` | `removeInfrequentWords` | `removeLongWords` | `removeShortWords` | `removeStopWords` | `tokenizedDocument`

Related Examples

- “Analyze Text Data Using Topic Models” on page 2-13
- “Compare LDA Solvers” on page 2-23

Compare LDA Solvers

This example shows how to compare latent Dirichlet allocation (LDA) solvers by comparing the goodness of fit and the time taken to fit the model.

Import Text Data

Import a set of abstracts and category labels from math papers using the arXiV API. Specify the number of records to import using the `importSize` variable. Note that the arXiV API is rate limited to querying 1000 articles at a time and requires waiting between requests.

```
importSize = 50000;
```

Import the first set of records.

```
url = "https://export.arxiv.org/oai2?verb=ListRecords" + ...
    "&set=math" + ...
    "&metadataPrefix=arXiv";
options = weboptions('Timeout',160);
code = webread(url,options);
```

Parse the returned XML content and create an array of `htmlTree` objects containing the record information.

```
tree = htmlTree(code);
subtrees = findElement(tree,"record");
numel(subtrees)
```

Iteratively import more chunks of records until the required amount is reached, or there are no more records. To continue importing records from where you left of, use the `resumptionToken` attribute from the previous result. To adhere to the rate limits imposed by the arXiV API, add a delay of 20 seconds before each query using the `pause` function.

```
while numel(subtrees) < importSize
    subtreeResumption = findElement(tree,"resumptionToken");

    if isempty(subtreeResumption)
        break
    end

    resumptionToken = extractHTMLText(subtreeResumption);

    url = "https://export.arxiv.org/oai2?verb=ListRecords" + ...
        "&resumptionToken=" + resumptionToken;

    pause(20)
    code = webread(url,options);

    tree = htmlTree(code);

    subtrees = [subtrees; findElement(tree,"record")];
end
```

Extract and Preprocess Text Data

Extract the abstracts and labels from the parsed HTML trees.

Find the `"<abstract>"` elements using the `findElement` function.

```
subtreesAbstract = htmlTree("");
for i = 1:numel(subtrees)
    subtreesAbstract(i) = findElement(subtrees(i),"abstract");
end
```

Extract the text data from the subtrees containing the abstracts using the `extractHTMLText` function.

```
textData = extractHTMLText(subtreesAbstract);
```

Set aside 30% of the documents at random for validation.

```
numDocuments = numel(textData);
cvp = cvpartition(numDocuments,'HoldOut',0.1);
textDataTrain = textData(training(cvp));
textDataValidation = textData(test(cvp));
```

Tokenize and preprocess the text data using the function `preprocessText` which is listed at the end of this example.

```
documentsTrain = preprocessText(textDataTrain);
documentsValidation = preprocessText(textDataValidation);
```

Create a bag-of-words model from the training documents. Remove the words that do not appear more than two times in total. Remove any documents containing no words.

```
bag = bagOfWords(documentsTrain);
bag = removeInfrequentWords(bag,2);
bag = removeEmptyDocuments(bag);
```

For the validation data, create a bag-of-words model from the validation documents. You do not need to remove any words from the validaiton data because any words that do not appear in the fitted LDA models are automatically ignored.

```
validationData = bagOfWords(documentsValidation);
```

Fit and Compare Models

For each of the LDA solvers, fit a model with 40 topics. To distinguish the solvers when plotting the results on the same axes, specify different line properties for each solver.

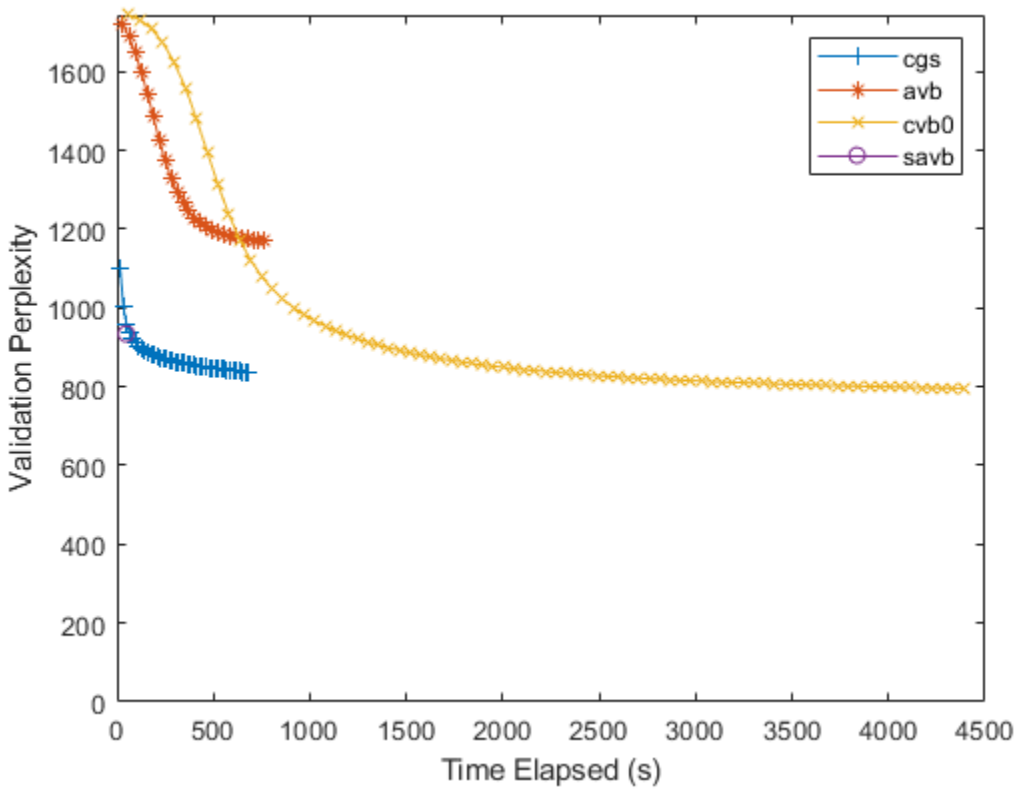
```
numTopics = 40;
solvers = ["cgs" "avb" "cvb0" "savb"];
lineSpecs = ["+-" "*-" "x-" "o-"];
```

Fit an LDA model using each solver. For each solver, specify the initial topic concentration 1, to validate the model once per data pass, and to not fit the topic concentration parameter. Using the data in the `FitInfo` property of the fitted LDA models, plot the validation perplexity and the time elapsed.

The stochastic solver, by default, uses a mini-batch size of 1000 and validates the model every 10 iterations. For this solver, to validate the model once per data pass, set the validation frequency to `ceil(numObservations/1000)`, where `numObservations` is the number of documents in the training data. For the other solvers, set the validation frequency to 1.

For the iterations that the stochastic solver does not evaluate the validation perplexity, the stochastic solver reports `NaN` in the `FitInfo` property. To plot the validation perplexity, remove the NaNs from the reported values.

```
numObservations = bag.NumDocuments;

figure
for i = 1:numel(solvers)
    solver = solvers(i);
    lineSpec = lineSpecs(i);

    if solver == "savb"
        numIterationsPerDataPass = ceil(numObservations/1000);
    else
        numIterationsPerDataPass = 1;
    end

    mdl = fitlda(bag,numTopics, ...
        'Solver',solver, ...
        'InitialTopicConcentration',1, ...
        'FitTopicConcentration',false, ...
        'ValidationData',validationData, ...
        'ValidationFrequency',numIterationsPerDataPass, ...
        'Verbose',0);

    history = mdl.FitInfo.History;

    timeElapsed = history.TimeSinceStart;

    validationPerplexity = history.ValidationPerplexity;

    % Remove NaNs.
    idx = isnan(validationPerplexity);
    timeElapsed(idx) = [];
    validationPerplexity(idx) = [];

    plot(timeElapsed,validationPerplexity,lineSpec)
    hold on
end

hold off
xlabel("Time Elapsed (s)")
ylabel("Validation Perplexity")
ylim([0 inf])
legend(solvers)
```



For the stochastic solver, there is only one data point. This is because this solver passes through input data once. To specify more data passes, use the `'DataPassLimit'` option. For the batch solvers (`"cgs"`, `"avb"`, and `"cvb0"`), to specify the number of iterations used to fit the models, use the `'IterationLimit'` option.

A lower validation perplexity suggests a better fit. Usually, the solvers `"savb"` and `"cgs"` converge quickly to a good fit. The solver `"cvb0"` might converge to a better fit, but it can take much longer to converge.

For the `FitInfo` property, the `fitlda` function estimates the validation perplexity from the document probabilities at the maximum likelihood estimates of the per-document topic probabilities. This is usually quicker to compute, but can be less accurate than other methods. Alternatively, calculate the validation perplexity using the `logp` function. This function calculates more accurate values but can take longer to run. For an example showing how to compute the perplexity using `logp`, see “Calculate Document Log-Probabilities from Word Count Matrix”.

Preprocessing Function

The function `preprocessText` performs the following steps:

1. Tokenize the text using `tokenizedDocument`.
2. Lemmatize the words using `normalizeWords`.
3. Erase punctuation using `erasePunctuation`.
4. Remove a list of stop words (such as "and", "of", and "the") using `removeStopWords`.

5 Remove words with 2 or fewer characters using `removeShortWords`.

6 Remove words with 15 or more characters using `removeLongWords`.

```
function documents = preprocessText(textData)

% Tokenize the text.
documents = tokenizedDocument(textData);

% Lemmatize the words.
documents = addPartOfSpeechDetails(documents);
documents = normalizeWords(documents,'Style','lemma');

% Erase punctuation.
documents = erasePunctuation(documents);

% Remove a list of stop words.
documents = removeStopWords(documents);

% Remove words with 2 or fewer characters, and words with 15 or greater
% characters.
documents = removeShortWords(documents,2);
documents = removeLongWords(documents,15);

end
```

See Also

`addPartOfSpeechDetails` | `bagOfWords` | `erasePunctuation` | `fitlda` | `ldaModel` | `logp` | `normalizeWords` | `removeEmptyDocuments` | `removeInfrequentWords` | `removeLongWords` | `removeShortWords` | `removeStopWords` | `tokenizedDocument` | `wordcloud`

Related Examples

- “Analyze Text Data Using Topic Models” on page 2-13
- “Choose Number of Topics for LDA Model” on page 2-19

Create Co-occurrence Network

This example shows how to create a co-occurrence network using a bag-of-words model.

Given a corpus of documents, a co-occurrence network is an undirected graph, with nodes corresponding to unique words in a vocabulary and edges corresponding to the frequency of words co-occurring in a document. Use co-occurrence networks to visualize and extract information of the relationships between words in a corpus of documents. For example, you can use a co-occurrence network to discover which words commonly appear with a specified word.

Import Text Data

Extract the text data in the file `weekendUpdates.xlsx` using `readtable`. The file `weekendUpdates.xlsx` contains status updates containing the hashtags `"#weekend"` and `"#vacation"`. Read the data using the `readtable` function and extract the text data from the `TextData` column.

```
filename = "weekendUpdates.xlsx";
tbl = readtable(filename,'TextType','string');
textData = tbl.TextData;
```

View the first few observations.

```
textData(1:5)
```

```
ans = 5x1 string
    "Happy anniversary! ❤ Next stop: Paris! ✈ #vacation"
    "Haha, BBQ on the beach, engage smug mode! □□□□❤ □□#vacation"
    "getting ready for Saturday night □□#yum #weekend □□"
    "Say it with me - I NEED A #VACATION!!! ☹"
    "□□Chilling □□at home for the first time in ages…This is the life! □□#weekend"
```

Preprocess Text Data

Tokenize the text, convert it to lowercase, and remove the stop words.

```
documents = tokenizedDocument(textData);

documents = lower(documents);
documents = removeStopWords(documents);
```

Create a matrix of word counts using a bag-of-words model.

```
bag = bagOfWords(documents);
counts = bag.Counts;
```

To compute the word co-occurrences, multiply the word-count matrix by its transpose.

```
cooccurrence = counts.'*counts;
```

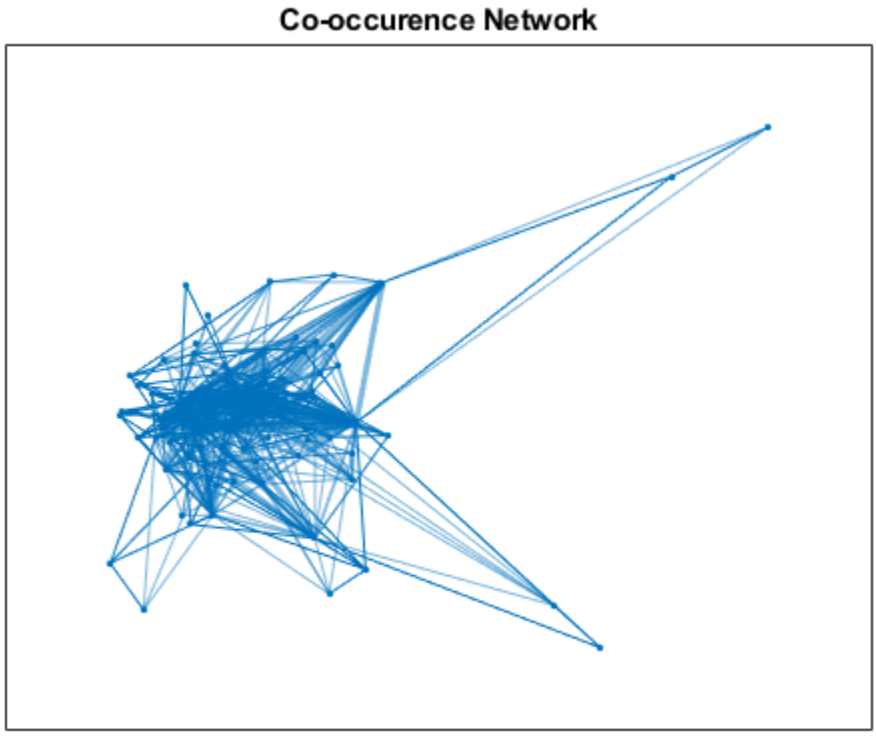
Convert the co-occurrence matrix to a network using the `graph` function.

```
G = graph(cooccurrence,bag.Vocabulary,'omitselfloops');
```

Visualize the network using the `plot` function. Set the line thickness to a multiple of the edge weight.

```
LWidths = 5*G.Edges.Weight/max(G.Edges.Weight);
```

```
plot(G,'LineWidth',LWidths)
title("Co-occurence Network")
```



Find neighbors of the word "great" using the `neighbors` function.

```
word = "great"
```

```
word =
"great"
```

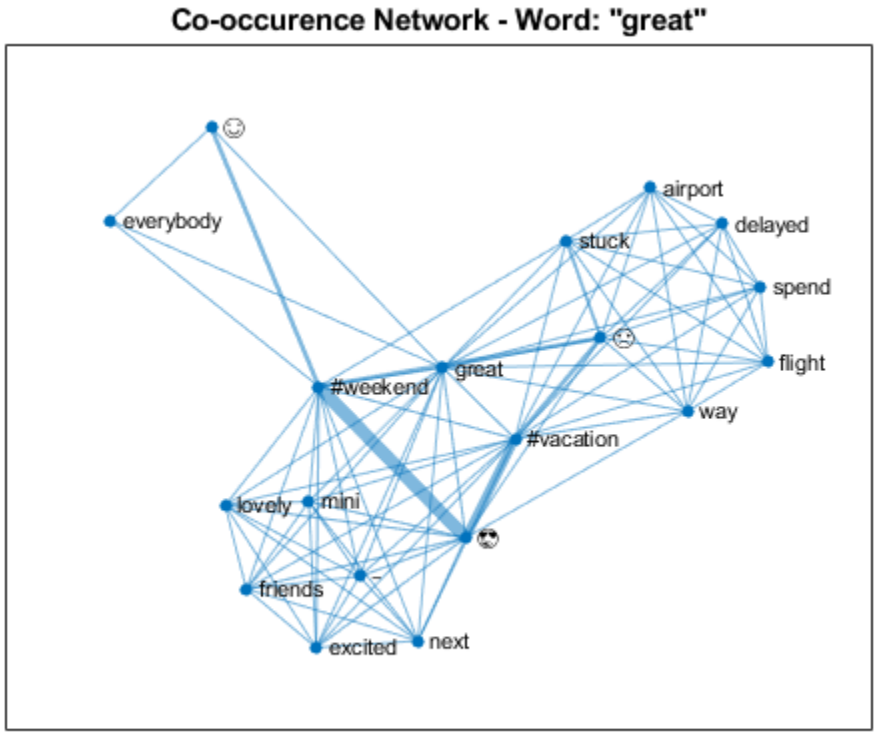
```
idx = find(bag.Vocabulary == word);
nbrs = neighbors(G,idx);
bag.Vocabulary(nbrs)'
```

```
ans = 18x1 string
    "next"
    "#vacation"
    "□□"
    "#weekend"
    "☹"
    "excited"
    "flight"
    "delayed"
    "stuck"
    "airport"
    "way"
    "spend"
    "□□"
```

```
    "lovely"
    "friends"
    "-"
    "mini"
    "everybody"
```

Visualize the co-occurrences of the word "great" by extracting a subgraph of this word and its neighbors.

```
H = subgraph(G,[idx; nbrs]);

LWidths = 5*H.Edges.Weight/max(H.Edges.Weight);
plot(H,'LineWidth',LWidths)
title("Co-occurence Network - Word: """ + word + """");
```



For more information about graphs and network analysis, see “Graph and Network Algorithms”.

See Also

`bagOfWords` | `graph` | `removeStopWords` | `tokenizedDocument`

Related Examples

- “Analyze Text Data Using Topic Models” on page 2-13
- “Analyze Text Data Using Multiword Phrases” on page 2-7

- “Analyze Text Data Containing Emojis” on page 2-32

Analyze Text Data Containing Emojis

This example shows how to analyze text data containing emojis.

Emojis are pictorial symbols that appear inline in text. When writing text on mobile devices such as smartphones and tablets, people use emojis to keep the text short and convey emotion and feelings.

You also can use emojis to analyze text data. For example, use them to identify relevant strings of text or to visualize the sentiment or emotion of the text.

When working with text data, emojis can behave unpredictably. Depending on your system fonts, your system might not display some emojis correctly. Therefore, if an emoji is not displayed correctly, then the data is not necessarily missing. Your system might be unable to display the emoji in the current font.

Composing Emojis

In most cases, you can read emojis from a file (for example, by using `extractFileText`, `extractHTMLText`, or `readtable`) or by copying and pasting them directly into MATLAB®. Otherwise, you must compose the emoji using Unicode UTF16 code units.

Some emojis consist of multiple Unicode UTF16 code units. For example, the "smiling face with sunglasses" emoji (😎 with code point U+1F60E) is a single glyph but comprises two UTF16 code units `"D83D"` and `"DE0E"`. Create a string containing this emoji using the `compose` function, and specify the two code units with the prefix `"\x"`.

```
emoji = compose("\xD83D\xDE0E")
```

```
emoji =
"😎"
```

First get the Unicode UTF16 code units of an emoji. Use `char` to get the numeric representation of the emoji, and then use `dec2hex` to get the corresponding hex value.

```
codeUnits = dec2hex(char(emoji))
```

```
codeUnits = 2x4 char array
    'D83D'
    'DE0E'
```

Reconstruct the composition string using the `strjoin` function with the empty delimiter `""`.

```
formatSpec = strjoin("\x" + codeUnits,"")
```

```
formatSpec =
"\xD83D\xDE0E"
```

```
emoji = compose(formatSpec)
```

```
emoji =
"😎"
```

Import Text Data

Extract the text data in the file `weekendUpdates.xlsx` using `readtable`. The file `weekendUpdates.xlsx` contains status updates containing the hashtags `"#weekend"` and `"#vacation"`.

```
filename = "weekendUpdates.xlsx";
tbl = readtable(filename,'TextType','string');
head(tbl)
```

```
ans=8×2 table
    ID                                        TextData
    __    ____________________________________________________________________________________

    1     "Happy anniversary! ❤ Next stop: Paris! ✈ #vacation"
    2     "Haha, BBQ on the beach, engage smug mode! □□□□❤ □□#vacation"
    3     "getting ready for Saturday night □□#yum #weekend □□"
    4     "Say it with me - I NEED A #VACATION!!! ☹"
    5     "□□Chilling □□at home for the first time in ages…This is the life! □□#weekend"
    6     "My last #weekend before the exam □□□□."
    7     "can’t believe my #vacation is over □□so unfair"
    8     "Can’t wait for tennis this #weekend □□□□□□"
```

Extract the text data from the field `TextData` and view the first few status updates.

```
textData = tbl.TextData;
textData(1:5)
```

```
ans = 5x1 string
    "Happy anniversary! ❤ Next stop: Paris! ✈ #vacation"
    "Haha, BBQ on the beach, engage smug mode! □□□□❤ □□#vacation"
    "getting ready for Saturday night □□#yum #weekend □□"
    "Say it with me - I NEED A #VACATION!!! ☹"
    "□□Chilling □□at home for the first time in ages…This is the life! □□#weekend"
```

Visualize the text data in a word cloud.

```
figure
wordcloud(textData);
```

Filter Text Data by Emoji

Identify the status updates containing a particular emoji using the `contains` function. Find the indices of the documents containing the "smiling face with sunglasses" emoji (😎 with code U+1F60E). This emoji comprises the two Unicode UTF16 code units `"D83D"` and `"DE0E"`.

```
emoji = compose("\xD83D\xDE0E");
idx = contains(textData,emoji);
textDataSunglasses = textData(idx);
textDataSunglasses(1:5)
```

```
ans = 5x1 string
    "Haha, BBQ on the beach, engage smug mode! 😎😎😎❤ 😎#vacation"
    "getting ready for Saturday night 😎#yum #weekend 😍"
    "😎Chilling 😎at home for the first time in ages…This is the life! 😎#weekend"
    "😎Check the out-of-office crew, we are officially ON #VACATION!! 😎"
    "Who needs a #vacation when the weather is this good ☀ 😎"
```

Visualize the extracted text data in a word cloud.

```
figure
wordcloud(textDataSunglasses);
```

Extract and Visualize Emojis

Visualize all the emojis in text data using a word cloud.

Extract the emojis. First tokenize the text using `tokenizedDocument`, and then view the first few documents.

```
documents = tokenizedDocument(textData);
documents(1:5)
```

```
ans =
  5x1 tokenizedDocument:

    11 tokens: Happy anniversary ! ❤ Next stop : Paris ! ✈ #vacation
    16 tokens: Haha , BBQ on the beach , engage smug mode ! □□□□❤ □□#vacation
     9 tokens: getting ready for Saturday night □□#yum #weekend □□
    13 tokens: Say it with me - I NEED A #VACATION ! ! ! ☹
    19 tokens: □□Chilling □□at home for the first time in ages … This is the life ! □□#weekend
```

The `tokenizedDocument` function automatically detects emoji and assigns the token type `"emoji"`. View the first few token details of the documents using the `tokenDetails` function.

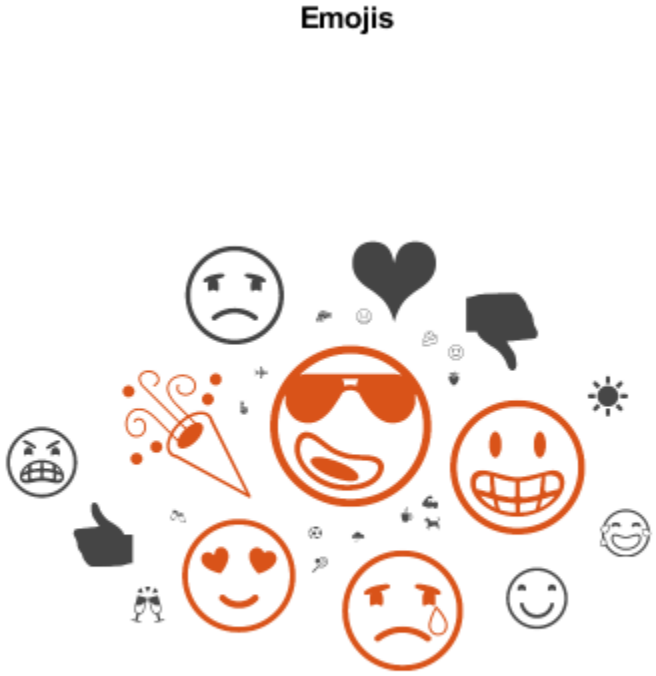
```
tdetails = tokenDetails(documents);
head(tdetails)
```

```
ans=8×5 table
        Token        DocumentNumber    LineNumber       Type        Language
```

```
_____________    ______________    __________    ___________    ________

"Happy"                1                1         letters          en
"anniversary"          1                1         letters          en
"!"                    1                1         punctuation      en
"❤"                    1                1         emoji            en
"Next"                 1                1         letters          en
"stop"                 1                1         letters          en
":"                    1                1         punctuation      en
"Paris"                1                1         letters          en
```

Visualize the emojis in a word cloud by extracting the tokens with token type `"emoji"` and inputting them into the `wordcloud` function.

```
idx = tdetails.Type == "emoji";
tokens = tdetails.Token(idx);
figure
wordcloud(tokens);
title("Emojis")
```



See Also

`tokenDetails` | `tokenizedDocument` | `wordcloud`

Related Examples

- “Analyze Text Data Using Topic Models” on page 2-13
- “Analyze Text Data Using Multiword Phrases” on page 2-7
- “Train a Sentiment Classifier” on page 2-51
- “Classify Text Data Using Deep Learning” on page 2-65
- “Generate Text Using Deep Learning” (Deep Learning Toolbox)

Analyze Sentiment in Text

This example shows how to use the Valence Aware Dictionary and sEntiment Reasoner (VADER) algorithm for sentiment analysis.

The VADER algorithm uses a list of annotated words (the sentiment lexicon), where each word has a corresponding sentiment score. The VADER algorithm also utilizes word lists that modify the scores of proceeding words in the text:

- Boosters – words or n-grams that boost the sentiment of proceeding tokens. For example, words like "absolutely" and "amazingly".
- Dampeners – words or n-grams that dampen the sentiment of proceeding tokens. For example, words like "hardly" and "somewhat".
- Negations – words that negate the sentiment of proceeding tokens. For example, words like "not" and "isn't".

To evaluate sentiment in text, use the `vaderSentimentScores` function.

Load Data

Extract the text data in the file `weekendUpdates.xlsx` using `readtable`. The file `weekendUpdates.xlsx` contains status updates containing the hashtags `"#weekend"` and `"#vacation"`.

```
filename = "weekendUpdates.xlsx";
tbl = readtable(filename,'TextType','string');
head(tbl)
```

```
ans=8×2 table
    ID                                        TextData
    __    ____________________________________________________________________________________

    1     "Happy anniversary! ❤ Next stop: Paris! ✈ #vacation"
    2     "Haha, BBQ on the beach, engage smug mode! □□□□❤ □□#vacation"
    3     "getting ready for Saturday night □□#yum #weekend □□"
    4     "Say it with me - I NEED A #VACATION!!! ☹"
    5     "□□Chilling □□at home for the first time in ages…This is the life! □□#weekend"
    6     "My last #weekend before the exam □□□□."
    7     "can’t believe my #vacation is over □□so unfair"
    8     "Can’t wait for tennis this #weekend □□□□□□"
```

Create an array of tokenized documents from the text data and view the first few documents.

```
str = tbl.TextData;
documents = tokenizedDocument(str);
documents(1:5)
```

```
ans =
  5x1 tokenizedDocument:

    11 tokens: Happy anniversary ! ❤ Next stop : Paris ! ✈ #vacation
    16 tokens: Haha , BBQ on the beach , engage smug mode ! □□□□❤ □□#vacation
     9 tokens: getting ready for Saturday night □□#yum #weekend □□
    13 tokens: Say it with me - I NEED A #VACATION ! ! ! ☹
```

```
    19 tokens: □□Chilling □□at home for the first time in ages … This is the life ! □□#weekend
```

Evaluate Sentiment

Evaluate the sentiment of the tokenized documents using the `vaderSentimentLexicon` function. Scores close to 1 indicate positive sentiment, scores close to -1 indicate negative sentiment, and scores close to 0 indicate neutral sentiment.

```
compoundScores = vaderSentimentScores(documents);
```

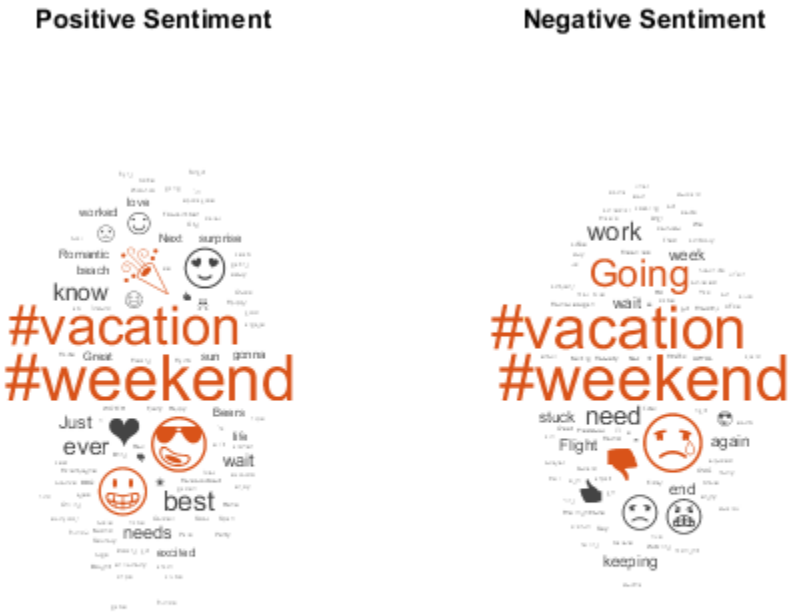
View the scores of the first few documents.

```
compoundScores(1:5)
```

```
ans = 5×1

    0.4738
    0.9348
    0.6705
   -0.5067
    0.7345
```

Visualize the text with positive and negative sentiment in word clouds.

```
idx = compoundScores > 0;
strPositive = str(idx);
strNegative = str(~idx);

figure
subplot(1,2,1)
wordcloud(strPositive);
title("Positive Sentiment")

subplot(1,2,2)
wordcloud(strNegative);
title("Negative Sentiment")
```



See Also

`ratioSentimentScores` | `tokenizedDocument` | `vaderSentimentScores`

More About

- “Generate Domain Specific Sentiment Lexicon” on page 2-41
- “Train a Sentiment Classifier” on page 2-51
- “Prepare Text Data for Analysis” on page 1-10
- “Analyze Text Data Containing Emojis” on page 2-32
- “Create Simple Text Model for Classification” on page 2-2
- “Analyze Text Data Using Topic Models” on page 2-13
- “Analyze Text Data Using Multiword Phrases” on page 2-7

Generate Domain Specific Sentiment Lexicon

This example shows how to generate a lexicon for sentiment analysis using 10-K and 10-Q financial reports.

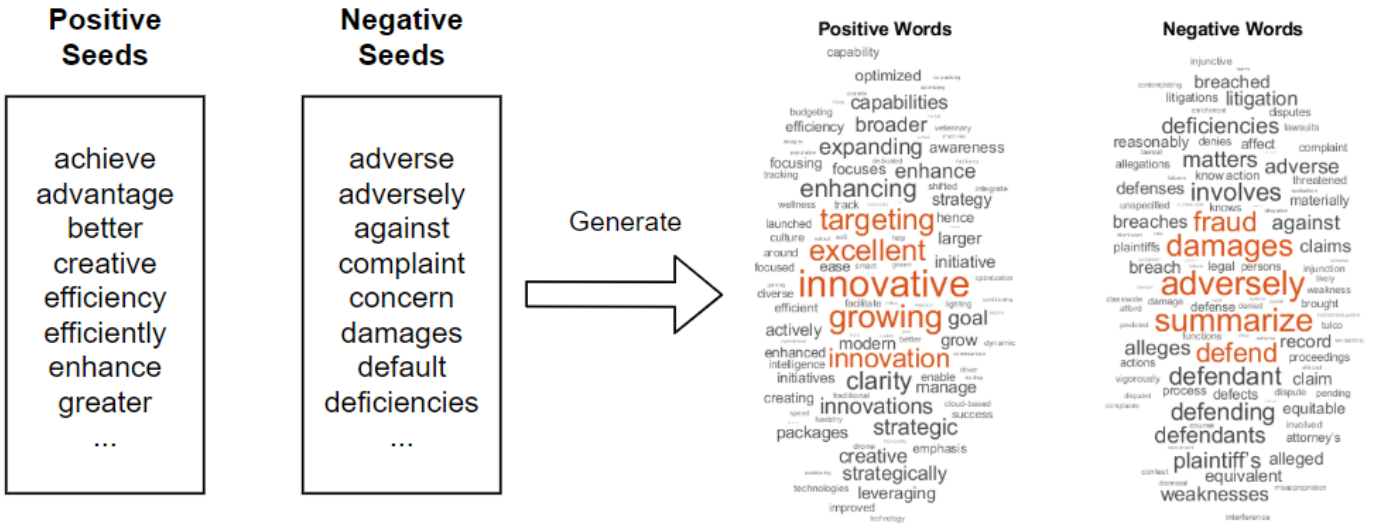
Sentiment analysis allows you to automatically summarize the sentiment in a given piece of text. For example, assign the pieces of text "This company is showing strong growth." and "This other company is accused of misleading consumers." with positive and negative sentiment, respectively. Also, for example, to assign the text "This company is showing *extremely* strong growth." a stronger sentiment score than the text "This company is showing strong growth."

Sentiment analysis algorithms such as VADER rely on annotated lists of words called sentiment lexicons. For example, VADER uses a sentiment lexicon with words annotated with a sentiment score ranging from -1 to 1, where scores close to 1 indicate strong positive sentiment, scores close to -1 indicate strong negative sentiment, and scores close to zero indicate neutral sentiment.

To analyze the sentiment of text using the VADER algorithm, use the `vaderSentimentScores` function. If the sentiment lexicon used by the `vaderSentimentScores` function does not suit the data you are analyzing, for example, if you have a domain-specific data set like medical or engineering data, then you can generate your own custom sentiment lexicon using a small set of seed words.

This example shows how to generate a sentiment lexicon given a collection of seed words using a graph-based approach based on [1 on page 2-0]:

- Train a word embedding that models the similarity between words using the training data.
- Create a simplified graph representing the embedding with nodes corresponding to words and edges weighted by similarity.
- To determine words with strong polarity, identify the words connected to multiple seed words through short but heavily weighted paths.



Load Data

Download the 10-K and 10-Q financial reports data from Securities and Exchange Commission (SEC) via the Electronic Data Gathering, Analysis, and Retrieval (EDGAR) API [2 on page 2-0] using the

`financeReports` helper function attached to this example as a supporting file. To access this file, open this example as a Live Script. The `financeReports` function downloads 10-K and 10-Q reports for the specified year, quarter, and maximum character length.

Download a set of reports from the fourth quarter 2019 with fewer than 2 million characters. Depending on the sizes of the reports, this can take some time to run. If you have Parallel Computing Toolbox™, then the function processes the reports in parallel.

```
year = 2019;
qtr = 4;
maxLength = 2e6;
textData = financeReports(year,qtr,maxLength);
```

```
Downloading 10-K and 10-Q reports...
Done.
Elapsed time is 605.480268 seconds.
```

Define sets of positive and negative seed words to use with this data.

```
seedsPositive = ["achieve" "advantage" "better" "creative" "efficiency" ...
    "efficiently" "enhance" "greater" "improved" "improving" ...
    "innovation" "innovations" "innovative" "opportunities" "profitable" ...
    "profitably" "strength" "strengthen" "strong" "success"]';

seedsNegative = ["adverse" "adversely" "against" "complaint" "concern" ...
    "damages" "default" "deficiencies" "disclosed" "failure" ...
    "fraud" "impairment" "litigation" "losses" "misleading" ...
    "omit" "restated" "restructuring" "termination" "weaknesses"]';
```

Prepare Text Data

Create a function names `preprocessText` that prepares the text data for analysis. The `preprocessText` function, listed at the end of the example performs the following steps:

- Erase any URLs.
- Tokenize the text.
- Remove tokens containing digits.
- Convert the text to lower case.
- Remove any words with two or fewer characters.
- Remove any stop words.

Preprocess the text using the `preprocessText` function. Depending on the size of the text data, this can take some time to run.

```
documents = preprocessText(textData);
```

Visualize the preprocessed text data in a word cloud.

```
figure
wordcloud(documents);
```

Train Word Embedding

Word embeddings map words in a vocabulary to numeric vectors. These embeddings can capture semantic details of the words so that similar words have similar vectors.

Train a word embedding that models the similarity between words using the training data. Specify a context window of size 25 and discard words that appear fewer than 20 times. Depending on the size of the text data, this can take some time to run.

```
emb = trainWordEmbedding(documents,'Window',25,'MinCount',20);
```

```
Computing Vocabulary. Word count in millions: 10.
Vocabulary count: 12648.
Training: 100% Loss: 0.713101 Remaining time: 0 hours 0 minutes.
```

Create Word Graph

Create a simplified graph representing the embedding with nodes corresponding to words and edges weighted by similarity.

Create a weighted graph with nodes corresponding to words in the vocabulary, edges denoting whether the words are within a neigborhood of 7 of each other, and weights corresponding to the cosine distance between the corresponding word vectors in the embedding.

For each word in the vocabulary, find the nearest 7 words and their cosine distances.

```
numNeighbors = 7;
vocabulary = emb.Vocabulary;
```

```
wordVectors = word2vec(emb,vocabulary);

[nearestWords,dist] = vec2word(emb,wordVectors,numNeighbors);
```

To create the graph, use the `graph` function and specify pairwise source and target nodes, and specify their edge weights.

Define the source and target nodes.

```
sourceNodes = repelem(vocabulary,numNeighbors);
targetNodes = reshape(nearestWords,1,[]);
```

Calculate the edge weights.

```
edgeWeights = reshape(dist,1,[]);
```

Create a graph connecting each word with its neigbors with edge weights corresponding to the similarity scores.

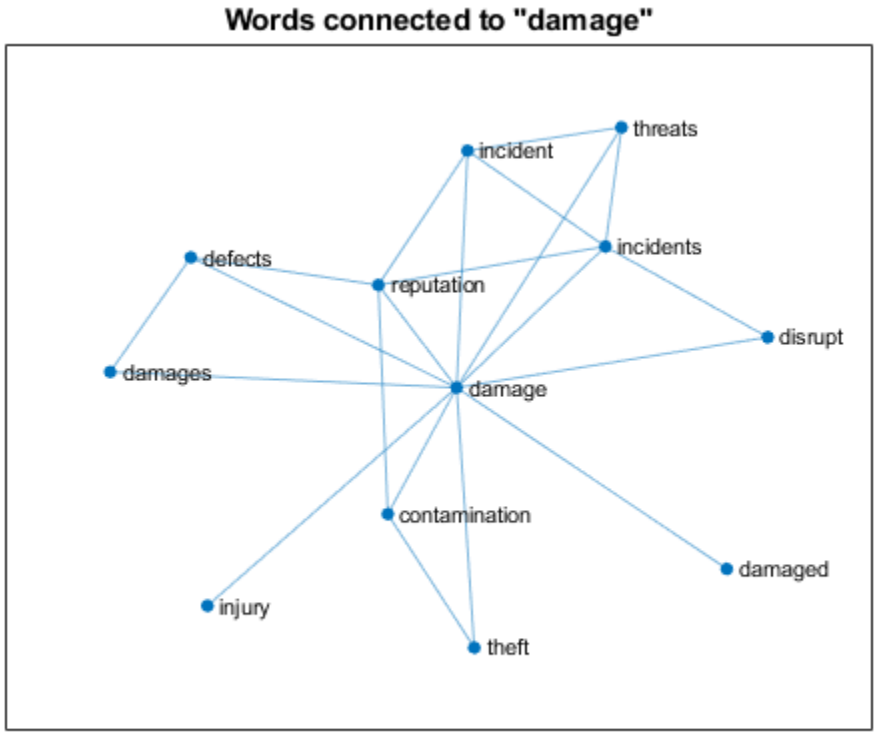
```
wordGraph = graph(sourceNodes,targetNodes,edgeWeights,vocabulary);
```

Remove the repeated edges using the `simplify` function.

```
wordGraph = simplify(wordGraph);
```

Visualize the section of the word graph connected to the word "damage".

```
word = "damage";
idx = findnode(wordGraph,word);
nbrs = neighbors(wordGraph,idx);
wordSubgraph = subgraph(wordGraph,[idx; nbrs]);
figure
plot(wordSubgraph)
title("Words connected to """ + word + """")
```



Generate Sentiment Scores

To determine words with strong polarity, identify the words connected to multiple seed words through short but heavily weighted paths.

Initialize an array of sentiment scores corresponding to each word in the vocabulary.

```
sentimentScores = zeros([1 numel(vocabulary)]);
```

Iteratively traverse the graph and update the sentiment scores.

Traverse the graph at different depths. For each depth, calculate the positive and negative polarity of the words by using the positive and negative seeds to propagate sentiment to the rest of the graph.

For each depth:

- Calculate the positive and negative polarity scores.
- Account for the difference in overall mass of positive and negative flow in the graph.
- For each node-word, normalize the difference of its two scores.

After running the algorithm, if a phrase has a higher positive than negative polarity score, then its final polarity will be positive, and negative otherwise.

Specify a maximum path length of 4.

```
maxPathLength = 4;
```

Iteratively traverse the graph and calculate the sum of the sentiment scores.

```
for depth = 1:maxPathLength

    % Calculate polarity scores.
    polarityPositive = polarityScores(seedsPositive,vocabulary,wordGraph,depth);
    polarityNegative = polarityScores(seedsNegative,vocabulary,wordGraph,depth);

    % Account for difference in overall mass of positive and negative flow
    % in the graph.
    b = sum(polarityPositive) / sum(polarityNegative);

    % Calculate new sentiment scores.
    sentimentScoresNew = polarityPositive - b * polarityNegative;
    sentimentScoresNew = normalize(sentimentScoresNew,'range',[-1,1]);

    % Add scores to sum.
    sentimentScores = sentimentScores + sentimentScoresNew;
end
```

Normalize the sentiment scores by the number of iterations.

```
sentimentScores = sentimentScores / maxPathLength;
```

Create a table containing the vocabulary and the corresponding sentiment scores.

```
tbl = table;
tbl.Token = vocabulary';
tbl.SentimentScore = sentimentScores';
```

To remove tokens with neutral sentiment from the lexicon, remove the tokens with sentiment score that have absolute value less than a threshold of 0.1.

```
thr = 0.1;
idx = abs(tbl.SentimentScore) < thr;
tbl(idx,:) = [];
```

Sort the table rows by descending sentiment score and view the first few rows.

```
tbl = sortrows(tbl,'SentimentScore','descend');
head(tbl)
```

```
ans=8×2 table
        Token        SentimentScore
    _____________    ______________

    "innovative"              1
    "efficiency"        0.91852
    "strong"            0.82362
    "efficiently"       0.81475
    "creative"          0.74264
    "enhance"           0.73791
    "innovations"       0.72985
    "improved"          0.71476
```

You can use this table as a custom sentiment lexicon for the `vaderSentimentScores` function.

Visualize the sentiment lexicon in word clouds. Display tokens with a positive score in one word cloud and tokens with negative scores in another. Display the words with sizes given by the absolute value their corresponding sentiment score.

```
figure
subplot(1,2,1);
idx = tbl.SentimentScore > 0;
tblPositive = tbl(idx,:);
wordcloud(tblTopPositive,'Token','SentimentScore')
title('Positive Words')

subplot(1,2,2);
idx = tbl.SentimentScore < 0;
tblNegative = tbl(idx,:);
tblNegative.SentimentScore = abs(tblNegative.SentimentScore);
wordcloud(tblTopNegative,'Token','SentimentScore')
title('Negative Words')
```

Export the table to a CSV file.

```
filename = "financeSentimentLexicon.csv";
writetable(tbl,filename)
```

Analyze Sentiment in Text

To analyze the sentiment in for previously unseen text data, preprocess the text using the same preprocessing steps and use the `vaderSentimentScores` function.

Create a string array containing the text data and preprocess it using the `preprocessText` function.

```
textDataNew = [
    "This company is showing extremely strong growth."
    "This other company is accused of misleading consumers."];
documentsNew = preprocessText(textDataNew);
```

Evaluate the sentiment using the `vaderSentimentScores` function. Specify the sentiment lexicon created in this example using the `'SentimentLexicon'` option.

```
compoundScores = vaderSentimentScores(documentsNew,'SentimentLexicon',tbl)
```

```
compoundScores = 2×1

    0.2834
   -0.1273
```

Positive and negative scores indicate positive and negative sentiment, respectively. The magnitude of the value corresponds to the strength of the sentiment.

Supporting Functions

Text Preprocessing Function

The `preprocessText` function performs the following steps:

- Erase any URLs.
- Tokenize the text.
- Remove tokens containing digits.
- Convert the text to lower case.
- Remove any words with two or fewer characters.
- Remove any stop words.

```
function documents = preprocessText(textData)

% Erase URLS.
textData = eraseURLs(textData);

% Tokenize.
documents = tokenizedDocument(textData);

% Remove tokens containing digits.
pat = textBoundary + wildcardPattern + digitsPattern + wildcardPattern + textBoundary;
documents = replace(documents,pat,"");

% Convert to lowercase.
documents = lower(documents);

% Remove short words.
documents = removeShortWords(documents,2);

% Remove stop words.
documents = removeStopWords(documents);

end
```

Polarity Scores Function

The `polarityScores` function returns a vector of polarity scores given a set of seed words, vocabulary, graph, and a specified depth. The function computes the sum over the maximum weighted path from every seed word to each node in the vocabulary. A high polarity score indicates phrases connected to multiple seed words via both short and strongly weighted paths.

The function performs the following steps:

- Initialize the scores of the seeds with ones and otherwise zeros.
- Loop over the seeds. For each seed, iteratively traverse the graph at different depth levels. For the first iteration, set the search space to the immediate neighbors of the seed.
- For each depth level, loop over the nodes in the search space and identify its neighbors in the graph.
- Loop over its neighbors and update the corresponding scores. The updated score is the maximum value of the current score for the seed and neighbor, and the score for the seed and search node weighted by the corresponding graph edge.
- At the end of the search for the depth level, append the neighbors to the search space. This increases the depth of the search for the next iteration.

The output polarity is the sum of the scores connected to the input seeds.

```
function polarity = polarityScores(seeds,vocabulary,wordGraph,depth)

% Initialize scores.
vocabularySize = numel(vocabulary);
scores = zeros(vocabularySize);
idx = ismember(vocabulary,seeds);
scores(idx,idx) = eye(numel(seeds));

% Loop over seeds.
for i = 1:numel(seeds)

    % Initialize search space.
    seed = seeds(i);
    idxSeed = vocabulary == seed;
    searchSpace = find(idxSeed);

    % Search at different depths.
    for d = 1:depth

        % Loop over nodes in search space.
        numNodes = numel(searchSpace);

        for k = 1:numNodes

            idxNew = searchSpace(k);

            % Find neighbors and weights.
            nbrs = neighbors(wordGraph,idxNew);
            idxWeights = findedge(wordGraph,idxNew,nbrs);
            weights = wordGraph.Edges.Weight(idxWeights);

            % Loop over neighbors.
            for j = 1:numel(nbrs)
```

```
                % Calculate scores.
                score = scores(idxSeed,nbrs(j));
                scoreNew = scores(idxSeed,idxNew);

                % Update score.
                scores(idxSeed,nbrs(j)) = max(score,scoreNew*weights(j));
            end

            % Appended nodes to search space for next depth iteration.
            searchSpace = [searchSpace nbrs'];
        end
    end
end

% Find seeds in vocabulary.
[~,idx] = ismember(seeds,vocabulary);

% Sum scores connected to seeds.
polarity = sum(scores(idx,:));

end
```

Bibliography

1 Velikovich, Lenid. "The Viability of Web-derived Polarity Lexicons." In *Proceedings of The Annual Conference of the North American Chapter of the Association for Computational Linguistics, 2010*, pp. 777-785. 2010.

2 Accessing EDGAR Data. https://www.sec.gov/edgar/searchedgar/accessing-edgar-data.htm

See Also

Train a Sentiment Classifier

This example shows how to train a classifier for sentiment analysis using an annotated list of positive and negative sentiment words and a pretrained word embedding.

The pretrained word embedding plays several roles in this workflow. It converts words into numeric vectors and forms the basis for a classifier. You can then use the classifier to predict the sentiment of other words using their vector representation, and use these classifications to calculate the sentiment of a piece of text. There are four steps in training and using the sentiment classifier:

- Load a pretrained word embedding.
- Load an opinion lexicon listing positive and negative words.
- Train a sentiment classifier using the word vectors of the positive and negative words.
- Calculate the mean sentiment scores of the words in a piece of text.

Load Pretrained Word Embedding

Word embeddings map words in a vocabulary to numeric vectors. These embeddings can capture semantic details of the words so that similar words have similar vectors. They also model relationships between words through vector arithmetic. For example, the relationship *Rome is to Paris as Italy is to France* is described by the equation $Rome - Italy + France \approx Paris$.

Load a pretrained word embedding using the `fastTextWordEmbedding` function. This function requires Text Analytics Toolbox™ Model *for fastText English 16 Billion Token Word Embedding* support package. If this support package is not installed, then the function provides a download link.

```
emb = fastTextWordEmbedding;
```

Load Opinion Lexicon

Load the positive and negative words from the opinion lexicon (also known as a sentiment lexicon) from https://www.cs.uic.edu/~liub/FBS/sentiment-analysis.html. [1] First, extract the files from the `.rar` file into a folder named `opinion-lexicon-English`, and then import the text.

Load the data using the function `readLexicon` listed at the end of this example. The output `data` is a table with variables `Word` containing the words, and `Label` containing a categorical sentiment label, `Positive` or `Negative`.

```
data = readLexicon;
```

View the first few words labeled as positive.

```
idx = data.Label == "Positive";
head(data(idx,:))
```

```
ans=8×2 table
        Word          Label
    ______________    ________

    "a+"              Positive
    "abound"          Positive
    "abounds"         Positive
    "abundance"       Positive
    "abundant"        Positive
    "accessable"      Positive
```

```
    "accessible"    Positive
    "acclaim"       Positive
```

View the first few words labeled as negative.

```
idx = data.Label == "Negative";
head(data(idx,:))
```

```
ans=8×2 table
        Word          Label
    ______________    ________

    "2-faced"         Negative
    "2-faces"         Negative
    "abnormal"        Negative
    "abolish"         Negative
    "abominable"      Negative
    "abominably"      Negative
    "abominate"       Negative
    "abomination"     Negative
```

Prepare Data for Training

To train the sentiment classifier, convert the words to word vectors using the pretrained word embedding `emb`. First remove the words that do not appear in the word embedding `emb`.

```
idx = ~isVocabularyWord(emb,data.Word);
data(idx,:) = [];
```

Set aside 10% of the words at random for testing.

```
numWords = size(data,1);
cvp = cvpartition(numWords,'HoldOut',0.1);
dataTrain = data(training(cvp),:);
dataTest = data(test(cvp),:);
```

Convert the words in the training data to word vectors using `word2vec`.

```
wordsTrain = dataTrain.Word;
XTrain = word2vec(emb,wordsTrain);
YTrain = dataTrain.Label;
```

Train Sentiment Classifier

Train a support vector machine (SVM) classifier which classifies word vectors into positive and negative categories.

```
mdl = fitcsvm(XTrain,YTrain);
```

Test Classifier

Convert the words in the test data to word vectors using `word2vec`.

```
wordsTest = dataTest.Word;
XTest = word2vec(emb,wordsTest);
YTest = dataTest.Label;
```

Predict the sentiment labels of the test word vectors.

```
[YPred,scores] = predict(mdl,XTest);
```

Visualize the classification accuracy in a confusion matrix.

```
figure
confusionchart(YTest,YPred);
```

Visualize the classifications in word clouds. Plot the words with positive and negative sentiments in word clouds with word sizes corresponding to the prediction scores.

```
figure
subplot(1,2,1)
idx = YPred == "Positive";
wordcloud(wordsTest(idx),scores(idx,1));
title("Predicted Positive Sentiment")

subplot(1,2,2)
wordcloud(wordsTest(~idx),scores(~idx,2));
title("Predicted Negative Sentiment")
```

Calculate Sentiment of Collections of Text

To calculate the sentiment of a piece of text, for example an update on social media, predict the sentiment score of each word in the text and take the mean sentiment score.

```
filename = "weekendUpdates.xlsx";
tbl = readtable(filename,'TextType','string');
textData = tbl.TextData;
textData(1:10)
```

```
ans = 10×1 string array
    "Happy anniversary! ❤ Next stop: Paris! ✈ #vacation"
    "Haha, BBQ on the beach, engage smug mode! □□□□❤ □□#vacation"
    "getting ready for Saturday night □□#yum #weekend □□"
    "Say it with me - I NEED A #VACATION!!! ☹"
    "□□Chilling □□at home for the first time in ages…This is the life! □□#weekend"
    "My last #weekend before the exam □□□□."
    "can’t believe my #vacation is over □□so unfair"
    "Can’t wait for tennis this #weekend □□□□□□"
    "I had so much fun! □□□□□□st trip EVER! □□□□□#vacation #weekend"
    "Hot weather and air con broke in car □□#sweaty #roadtrip #vacation"
```

Create a function which tokenizes and preprocesses the text data so it can be used for analysis. The function `preprocessText`, listed at the end of the example, performs the following steps in order:

1 Tokenize the text using `tokenizedDocument`.

2. Erase punctuation using `erasePunctuation`.
3. Remove stop words (such as "and", "of", and "the") using `removeStopWords`.
4. Convert to lowercase using `lower`.

Use the preprocessing function `preprocessText` to prepare the text data. This step can take a few minutes to run.

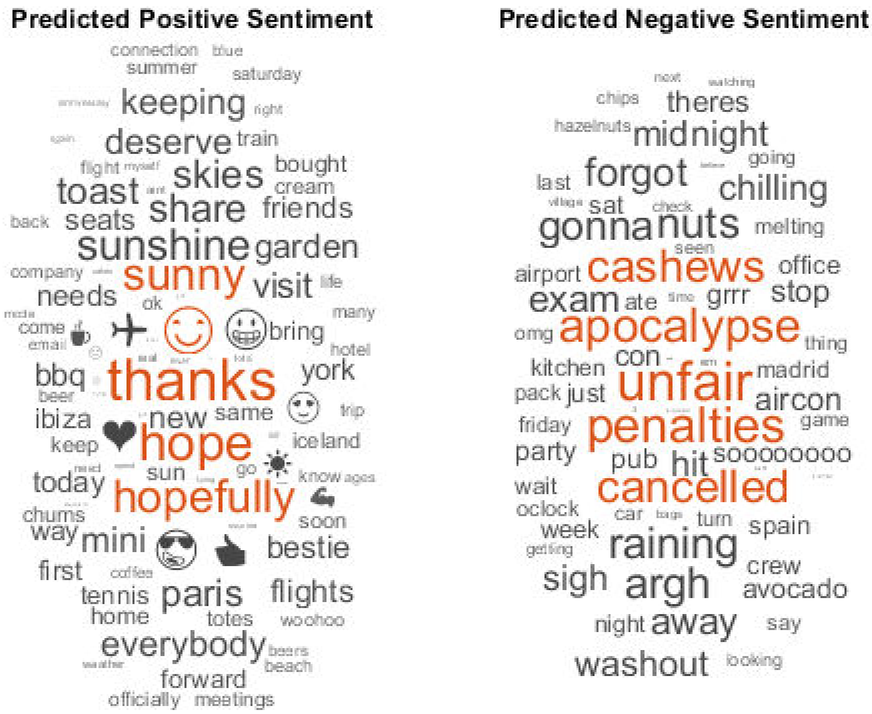
```
documents = preprocessText(textData);
```

Remove the words from the documents that do not appear in the word embedding `emb`.

```
idx = ~isVocabularyWord(emb,documents.Vocabulary);
documents = removeWords(documents,idx);
```

To visualize how well the sentiment classifier generalizes to the new text, classify the sentiments on the words that occur in the text, but not in the training data and visualize them in word clouds. Use the word clouds to manually check that the classifier behaves as expected.

```
words = documents.Vocabulary;
words(ismember(words,wordsTrain)) = [];

vec = word2vec(emb,words);
[YPred,scores] = predict(mdl,vec);

figure
subplot(1,2,1)
idx = YPred == "Positive";
wordcloud(words(idx),scores(idx,1));
title("Predicted Positive Sentiment")

subplot(1,2,2)
wordcloud(words(~idx),scores(~idx,2));
title("Predicted Negative Sentiment")
```



To calculate the sentiment of a given piece of text, compute the sentiment score for each word in the text and calculate the mean sentiment score.

Calculate the mean sentiment score of the updates. For each document, convert the words to word vectors, predict the sentiment score on the word vectors, transform the scores using the score-to-posterior transform function and then calculate the mean sentiment score.

```
for i = 1:numel(documents)
    words = string(documents(i));
    vec = word2vec(emb,words);
    [~,scores] = predict(mdl,vec);
    sentimentScore(i) = mean(scores(:,1));
end
```

View the predicted sentiment scores with the text data. Scores greater than 0 correspond to positive sentiment, scores less than 0 correspond to negative sentiment, and scores close to 0 correspond to neutral sentiment.

```
table(sentimentScore', textData)
```

```
ans=50×2 table
       Var1                                                                  textData
    __________    ________________________________________________________________________________

      1.8382     "Happy anniversary! ❤ Next stop: Paris! ✈ #vacation"
       1.294     "Haha, BBQ on the beach, engage smug mode! □□□□❤ □□#vacation"
      1.0922     "getting ready for Saturday night □□#yum #weekend □□"
```

```
      0.094709    "Say it with me - I NEED A #VACATION!!! ☹"
        1.4073    "□□Chilling □□at home for the first time in ages…This is the life! □□#weekend"
       -0.8356    "My last #weekend before the exam □□□□."
       -1.3556    "can't believe my #vacation is over □□so unfair"
        1.4312    "Can't wait for tennis this #weekend □□□□□□"
        3.0458    "I had so much fun! □□□□□Best trip EVER! □□□□#Vacation #weekend"
       -0.39243    "Hot weather and air con broke in car □□#sweaty #roadtrip #vacation"
        0.8028    "□□Check the out-of-office crew, we are officially ON #VACATION!! □□"
       0.38217    "Well that wasn't how I expected this #weekend to go □□Total washout!! □□"
          3.03    "So excited for my bestie to visit this #weekend! □□❤ □□"
        2.3849    "Who needs a #vacation when the weather is this good ☀ □□"
    -0.0006176    "I love meetings in summer that run into the weekend! Wait that was sarcasm. B
       0.52992    "You know we all worked hard for this! We totes deserve this □□#vacation □□Ibi
      ⋮
```

Sentiment Lexicon Reading Function

This function reads the positive and negative words from the sentiment lexicon and returns a table. The table contains variables `Word` and `Label`, where `Label` contains categorical values `Positive` and `Negative` corresponding to the sentiment of each word.

```
function data = readLexicon

% Read positive words
fidPositive = fopen(fullfile('opinion-lexicon-English','positive-words.txt'));
C = textscan(fidPositive,'%s','CommentStyle',';');
wordsPositive = string(C{1});

% Read negative words
fidNegative = fopen(fullfile('opinion-lexicon-English','negative-words.txt'));
C = textscan(fidNegative,'%s','CommentStyle',';');
wordsNegative = string(C{1});
fclose all;

% Create table of labeled words
words = [wordsPositive;wordsNegative];
labels = categorical(nan(numel(words),1));
labels(1:numel(wordsPositive)) = "Positive";
labels(numel(wordsPositive)+1:end) = "Negative";

data = table(words,labels,'VariableNames',{'Word','Label'});

end
```

Preprocessing Function

The function `preprocessText` performs the following steps:

1. Tokenize the text using `tokenizedDocument`.
2. Erase punctuation using `erasePunctuation`.
3. Remove stop words (such as "and", "of", and "the") using `removeStopWords`.
4. Convert to lowercase using `lower`.

```
function documents = preprocessText(textData)

% Tokenize the text.
```

```
documents = tokenizedDocument(textData);

% Erase punctuation.
documents = erasePunctuation(documents);

% Remove a list of stop words.
documents = removeStopWords(documents);

% Convert to lowercase.
documents = lower(documents);

end
```

Bibliography

1 Hu, Minqing, and Bing Liu. "Mining and summarizing customer reviews." In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 168-177. ACM, 2004.

See Also

`bagOfWords` | `erasePunctuation` | `fastTextWordEmbedding` | `removeStopWords` | `removeWords` | `tokenizedDocument` | `word2vec` | `wordcloud`

Related Examples

- “Analyze Sentiment in Text” on page 2-38
- “Generate Domain Specific Sentiment Lexicon” on page 2-41
- “Create Simple Text Model for Classification” on page 2-2
- “Analyze Text Data Containing Emojis” on page 2-32
- “Analyze Text Data Using Topic Models” on page 2-13
- “Analyze Text Data Using Multiword Phrases” on page 2-7
- “Classify Text Data Using Deep Learning” on page 2-65
- “Generate Text Using Deep Learning” (Deep Learning Toolbox)

See Also

`bleuEvaluationScore` | `bm25Similarity` | `cosineSimilarity` | `extractSummary` | `lexrankScores` | `mmrScores` | `rougeEvaluationScore` | `textrankScores` | `tokenizedDocument`

More About

- “Sequence-to-Sequence Translation Using Attention” on page 2-114

Extract Keywords from Text Data Using RAKE

This example shows how to extract keywords from text data using Rapid Automatic Keyword Extraction (RAKE).

The RAKE algorithm extracts keywords using a delimiter-based approach to identify candidate keywords and scores them using word co-occurrences that appear in the candidate keywords. Keywords can contain multiple tokens. Furthermore, the RAKE algorithm also merges keywords when they appear multiple times, separated by the same merging delimiter.

Extract Keywords

Create an array of tokenized document containing the text data.

```
textData = [
    "MATLAB provides tools for scientists and engineers. MATLAB is used by scientists and enginee
    "Analyze text and images. You can import text and images."
    "Analyze text and images. Analyze text, images, and videos in MATLAB."];
documents = tokenizedDocument(textData);
```

Extract the keywords using the `rakeKeywords` function.

```
tbl = rakeKeywords(documents)
```

```
tbl=12×3 table
                    Keyword                       DocumentNumber    Score
    ____________________________________________    ______________    _____

    "MATLAB"        "provides"    "tools"               1             8
    "MATLAB"        ""            ""                    1             2
    "scientists"    "and"         "engineers"           1             2
    "engineers"     ""            ""                    1             1
    "scientists"    ""            ""                    1             1
    "Analyze"       "text"        ""                    2             4
    "import"        "text"        ""                    2             4
    "images"        ""            ""                    2             1
    "Analyze"       "text"        ""                    3             4
    "MATLAB"        ""            ""                    3             1
    "images"        ""            ""                    3             1
    "videos"        ""            ""                    3             1
```

If a keyword contains multiple words, then the *i*th element of the string array corresponds to the *i*th word of the keyword. If the keyword has fewer words that the longest keyword, then remaining entries of the string array are the empty string `""`.

For readability, transform the multi-word keywords into a single sting using the `join` and `strip` functions.

```
if size(tbl.Keyword,2) > 1
    tbl.Keyword = strip(join(tbl.Keyword));
end
head(tbl)
```

```
ans=8×3 table
            Keyword              DocumentNumber    Score
    ____________________________    ______________    _____
```

```
    "MATLAB provides tools"            1             8
    "MATLAB"                           1             2
    "scientists and engineers"         1             2
    "engineers"                        1             1
    "scientists"                       1             1
    "Analyze text"                     2             4
    "import text"                      2             4
    "images"                           2             1
```

Specify Maximum Number of Keywords Per Document

The `rakeKeywords` function, by default, returns all identified keywords. To reduce the number of keywords, use the `'MaxNumKeywords'` option.

Extract the top three keywords for each document by setting the `'MaxNumKeywords'` option to 3.

```
tbl = rakeKeywords(documents,'MaxNumKeywords',3)
```

```
tbl=9×3 table
                    Keyword                     DocumentNumber    Score
    ____________________________________________    ______________    _____

    "MATLAB"        "provides"    "tools"             1              8
    "MATLAB"        ""            ""                  1              2
    "scientists"    "and"         "engineers"         1              2
    "Analyze"       "text"        ""                  2              4
    "import"        "text"        ""                  2              4
    "images"        ""            ""                  2              1
    "Analyze"       "text"        ""                  3              4
    "MATLAB"        ""            ""                  3              1
    "images"        ""            ""                  3              1
```

Specify Delimiters

Notice that in the extracted keywords above, the function extracts the multi-word keyword "scientists and engineers" from the first document, but does not extract the multi-word keyword "text and images" from the second document. This is because the RAKE algorithm uses tokens appearing between delimiters as candidate keywords, and the algorithm only merges keywords with delimiters when the merged phrase appears multiple times.

In this case, the instances of the token "text" appears within the two different multi-word keyword candidates "Analyze text" and "import text". Because, in this case, the function does not extract "text" as a separate candidate keyword, the algorithm does not consider merging candidates with the delimiter "and" and the candidate keyword "images".

You can specify the delimiters used for extracting keywords using the `'Delimiters'` and `'MergingDelimiters'`options. To specify delimiters that should not appear in extracted keywords, use the `'Delimiters'` option. To specify delimiters that can appear in extracted keywords, use the `'MergingDelimiters'` option.

Extract keywords from the same text as before and also specify the words "Analyze" and "import" as merging delimiters.

```
newDelimiters = ["Analyze" "import"];
mergingDelimiters = [stopWords newDelimiters];
```

```
tbl = rakeKeywords(documents,'MergingDelimiters', mergingDelimiters)
```

```
tbl=12×3 table
                     Keyword                     DocumentNumber    Score
    ____________________________________________    ______________    _____

    "MATLAB"        "provides"    "tools"               1              8
    "MATLAB"        ""            ""                    1              2
    "scientists"    "and"         "engineers"           1              2
    "engineers"     ""            ""                    1              1
    "scientists"    ""            ""                    1              1
    "text"          "and"         "images"              2              2
    "images"        ""            ""                    2              1
    "text"          ""            ""                    2              1
    "MATLAB"        ""            ""                    3              1
    "images"        ""            ""                    3              1
    "text"          ""            ""                    3              1
    "videos"        ""            ""                    3              1
```

Notice here that the function treats the tokens "text" and "images" as keywords and also extracts the merged keyword "text and images". To learn more about the RAKE algorithm, see “Rapid Automatic Keyword Extraction”.

Alternatives

You can experiment with different keyword extraction algorithms to see what works best with your data. Because the RAKE algorithm uses a delimiter-based approach to extract candidate keywords, the extracted keywords can be very long. Alternatively, you can try extracting keywords using TextRank algorithm which starts with individual tokens as candidate keywords and then merges them when appropriate. To extract keywords using TextRank, use the `textrankKeywords` function. To learn more, see “Extract Keywords from Text Data Using TextRank” on page 2-62.

References

[1] Rose, Stuart, Dave Engel, Nick Cramer, and Wendy Cowley. "Automatic keyword extraction from individual documents." *Text mining: applications and theory* 1 (2010): 1-20.

See Also

`extractSummary` | `rakeKeywords` | `textrankKeywords` | `tokenizedDocument`

More About

- “Extract Keywords from Text Data Using TextRank” on page 2-62

Extract Keywords from Text Data Using TextRank

This example shows to extract keywords from text data using TextRank.

The TextRank keyword extraction algorithm extracts keywords using a part-of-speech tag-based approach to identify candidate keywords and scores them using word co-occurrences determined by a sliding window. Keywords can contain multiple tokens. Furthermore, the TextRank keyword extraction algorithm also merges keywords when they appear consecutively in a document.

Extract Keywords

Create an array of tokenized document containing the text data.

```
textData = [
    "MATLAB provides really useful tools for engineers. Scientists use many useful MATLAB toolbox
    "MATLAB and Simulink have many features. MATLAB and Simulink makes it easy to develop models
    "You can easily import data in MATLAB. In particular, you can easily import text data."];
documents = tokenizedDocument(textData);
```

Extract the keywords using the `textrankKeywords` function.

```
tbl = textrankKeywords(documents)
```

```
tbl=6×3 table
                  Keyword                  DocumentNumber    Score
    ___________________________________    ______________    ______

    "useful"    "MATLAB"      "toolboxes"         1          4.8695
    "useful"    ""            ""                  1          2.3612
    "MATLAB"    ""            ""                  1          1.6212
    "many"      "features"    ""                  2          4.6152
    "text"      "data"        ""                  3          3.4781
    "data"      ""            ""                  3          1.7391
```

If a keyword contains multiple words, then the *i*th element of the string array corresponds to the *i*th word of the keyword. If the keyword has fewer words that the longest keyword, then remaining entries of the string array are the empty string `""`.

For readability, transform the multi-word keywords into a single sting using the `join` and `strip` functions.

```
if size(tbl.Keyword,2) > 1
    tbl.Keyword = strip(join(tbl.Keyword));
end
head(tbl)
```

```
ans=6×3 table
             Keyword              DocumentNumber    Score
    ____________________________    ______________    ______

    "useful MATLAB toolboxes"           1          4.8695
    "useful"                            1          2.3612
    "MATLAB"                            1          1.6212
    "many features"                     2          4.6152
    "text data"                         3          3.4781
    "data"                              3          1.7391
```

Specify Maximum Number of Keywords Per Document

The `textrankKeywords` function, by default, returns all identified keywords. To reduce the number of keywords, use the `'MaxNumKeywords'` option.

Extract the top two keywords for each document by setting the `'MaxNumKeywords'` option to 2.

```
tbl = textrankKeywords(documents,'MaxNumKeywords',2)
```

```
tbl=5×3 table
                  Keyword                  DocumentNumber    Score
    ___________________________________    ______________    ______

    "useful"    "MATLAB"      "toolboxes"         1          4.8695
    "useful"    ""            ""                  1          2.3612
    "many"      "features"    ""                  2          4.6152
    "text"      "data"        ""                  3          3.4781
    "data"      ""            ""                  3          1.7391
```

Specify Part-of-Speech Tags

Notice that in the extracted keywords above, the function does not consider the word "import" as a keyword. This is because the TextRank keyword extraction algorithm, by default, uses tokens with the part-of-speech tags "noun", "proper-noun" and "adjective" as candidate keywords. Because the word "import" is a verb, the algorithm does not consider this as a candidate keyword. Similarly, the algorithm does not consider the adverb "easily" as a candidate keyword.

To specify which part-of-speech tags to use for identifying candidate keywords, use the `'PartOfSpeech'` option.

Extract keywords from the same text as before and also specify also specify the part-of-speech tags `"adverb"` and `"verb"`.

```
newTags = ["adverb" "verb"];
tags = ["noun" "proper-noun" "adjective" newTags];
tbl = textrankKeywords(documents,'PartOfSpeech', tags)
```

```
tbl=7×3 table
                      Keyword                      DocumentNumber    Score
    ___________________________________________    ______________    ______

    "use"          "many"    "useful"    "MATLAB"         1          5.8839
    "useful"       ""        ""          ""               1          2.0169
    "MATLAB"       ""        ""          ""               1          1.5478
    "Simulink"     "have"    "many"      ""               2          4.5058
    "Simulink"     ""        ""          ""               2          1.5161
    "import"       "text"    "data"      ""               3          4.7921
    "import"       "data"    ""          ""               3          3.4195
```

Notice here that the function treats the token "import" as a candidate keyword and merges it into the multi-word keywords "import data" and "import text data".

Specify Windows Size

Notice that in the extracted keywords above, that the function does not extract the adverb "easily" as a keyword. This is because of the proximity of these words in the text to other candidate keywords.

The TextRank keyword extraction algorithm scores candidate keywords using the number of pairwise co-occurrences within a sliding window. To increase the window size, use the `'Window'` option. Increasing the window size enables the function to find more co-occurrences between keywords which increases the keyword importance scores. This can result in finding more relevant keywords at the cost of potentially over-scoring less relevant keywords.

Extract keywords from the same text as before and also specify also specify a window size of 3.

```
tbl = textrankKeywords(documents, ...
    'PartOfSpeech', tags, ...
    'Window',3)
```

```
tbl=8×3 table
                      Keyword                       DocumentNumber    Score
    ____________________________________________    ______________    ______

    "many"         "useful"    "MATLAB"    ""             1          4.2185
    "really"       "useful"    ""          ""             1          2.8851
    "MATLAB"       ""          ""          ""             1          1.3154
    "Simulink"     ""          ""          ""             2          1.4526
    "develop"      ""          ""          ""             2          1.0912
    "features"     ""          ""          ""             2          1.0794
    "easily"       "import"    "text"      "data"         3          5.2989
    "easily"       "import"    "data"      ""             3          4.0842
```

Notice here that the function treats the tokens "easily" as keywords and merges it into the multi-word keywords "easily import text data" and "easily import data".

To learn more about the TextRank keyword extraction algorithm, see “TextRank Keyword Extraction”.

Alternatives

You can experiment with different keyword extraction algorithms to see what works best with your data. Because the TextRank keywords algorithm uses a part-of-speech tag-based approach to extract candidate keywords, the extracted keywords can be short. Alternatively, you can try extracting keywords using RAKE algorithm which extracts sequences of tokens appearing between delimiters as candidate keywords. To extract keywords using RAKE, use the `rakeKeywords` function. To learn more, see “Extract Keywords from Text Data Using RAKE” on page 2-59.

References

[1] Mihalcea, Rada, and Paul Tarau. "Textrank: Bringing order into text." In *Proceedings of the 2004 conference on empirical methods in natural language processing*, pp. 404-411. 2004.

See Also

`extractSummary` | `rakeKeywords` | `textrankKeywords` | `tokenizedDocument`

More About

- “Extract Keywords from Text Data Using RAKE” on page 2-59

Classify Text Data Using Deep Learning

This example shows how to classify text data using a deep learning long short-term memory (LSTM) network.

Text data is naturally sequential. A piece of text is a sequence of words, which might have dependencies between them. To learn and use long-term dependencies to classify sequence data, use an LSTM neural network. An LSTM network is a type of recurrent neural network (RNN) that can learn long-term dependencies between time steps of sequence data.

To input text to an LSTM network, first convert the text data into numeric sequences. You can achieve this using a word encoding which maps documents to sequences of numeric indices. For better results, also include a word embedding layer in the network. Word embeddings map words in a vocabulary to numeric vectors rather than scalar indices. These embeddings capture semantic details of the words, so that words with similar meanings have similar vectors. They also model relationships between words through vector arithmetic. For example, the relationship "*Rome is to Italy as* Paris *is to France*" is described by the equation Italy – *Rome* + *Paris* = *France*.

There are four steps in training and using the LSTM network in this example:

- Import and preprocess the data.
- Convert the words to numeric sequences using a word encoding.
- Create and train an LSTM network with a word embedding layer.
- Classify new text data using the trained LSTM network.

Import Data

Import the factory reports data. This data contains labeled textual descriptions of factory events. To import the text data as strings, specify the text type to be `'string'`.

```
filename = "factoryReports.csv";
data = readtable(filename,'TextType','string');
head(data)
```

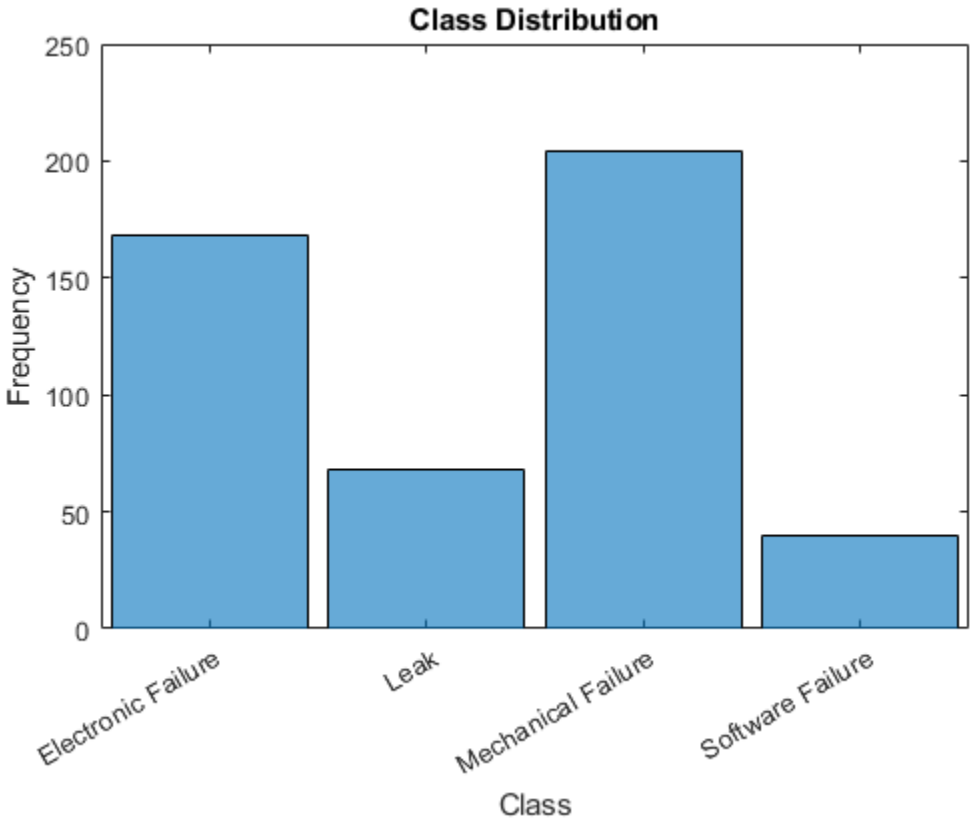
```
ans=8×5 table
                                Description                                 Category
    ____________________________________________________________________    ____________

    "Items are occasionally getting stuck in the scanner spools."           "Mechanical Failure"
    "Loud rattling and banging sounds are coming from assembler pistons."   "Mechanical Failure"
    "There are cuts to the power when starting the plant."                  "Electronic Failure"
    "Fried capacitors in the assembler."                                    "Electronic Failure"
    "Mixer tripped the fuses."                                              "Electronic Failure"
    "Burst pipe in the constructing agent is spraying coolant."             "Leak"
    "A fuse is blown in the mixer."                                         "Electronic Failure"
    "Things continue to tumble off of the belt."                            "Mechanical Failure"
```

The goal of this example is to classify events by the label in the `Category` column. To divide the data into classes, convert these labels to categorical.

```
data.Category = categorical(data.Category);
```

View the distribution of the classes in the data using a histogram.

```
figure
histogram(data.Category);
xlabel("Class")
ylabel("Frequency")
title("Class Distribution")
```



The next step is to partition it into sets for training and validation. Partition the data into a training partition and a held-out partition for validation and testing. Specify the holdout percentage to be 20%.

```
cvp = cvpartition(data.Category,'Holdout',0.2);
dataTrain = data(training(cvp),:);
dataValidation = data(test(cvp),:);
```

Extract the text data and labels from the partitioned tables.

```
textDataTrain = dataTrain.Description;
textDataValidation = dataValidation.Description;
YTrain = dataTrain.Category;
YValidation = dataValidation.Category;
```

To check that you have imported the data correctly, visualize the training text data using a word cloud.

```
figure
wordcloud(textDataTrain);
title("Training Data")
```

Preprocess Text Data

Create a function that tokenizes and preprocesses the text data. The function `preprocessText`, listed at the end of the example, performs these steps:

1. Tokenize the text using `tokenizedDocument`.
2. Convert the text to lowercase using `lower`.
3. Erase the punctuation using `erasePunctuation`.

Preprocess the training data and the validation data using the `preprocessText` function.

```
documentsTrain = preprocessText(textDataTrain);
documentsValidation = preprocessText(textDataValidation);
```

View the first few preprocessed training documents.

```
documentsTrain(1:5)
```

```
ans =
  5×1 tokenizedDocument:

     9 tokens: items are occasionally getting stuck in the scanner spools
    10 tokens: loud rattling and banging sounds are coming from assembler pistons
    10 tokens: there are cuts to the power when starting the plant
     5 tokens: fried capacitors in the assembler
     4 tokens: mixer tripped the fuses
```

Convert Document to Sequences

To input the documents into an LSTM network, use a word encoding to convert the documents into sequences of numeric indices.

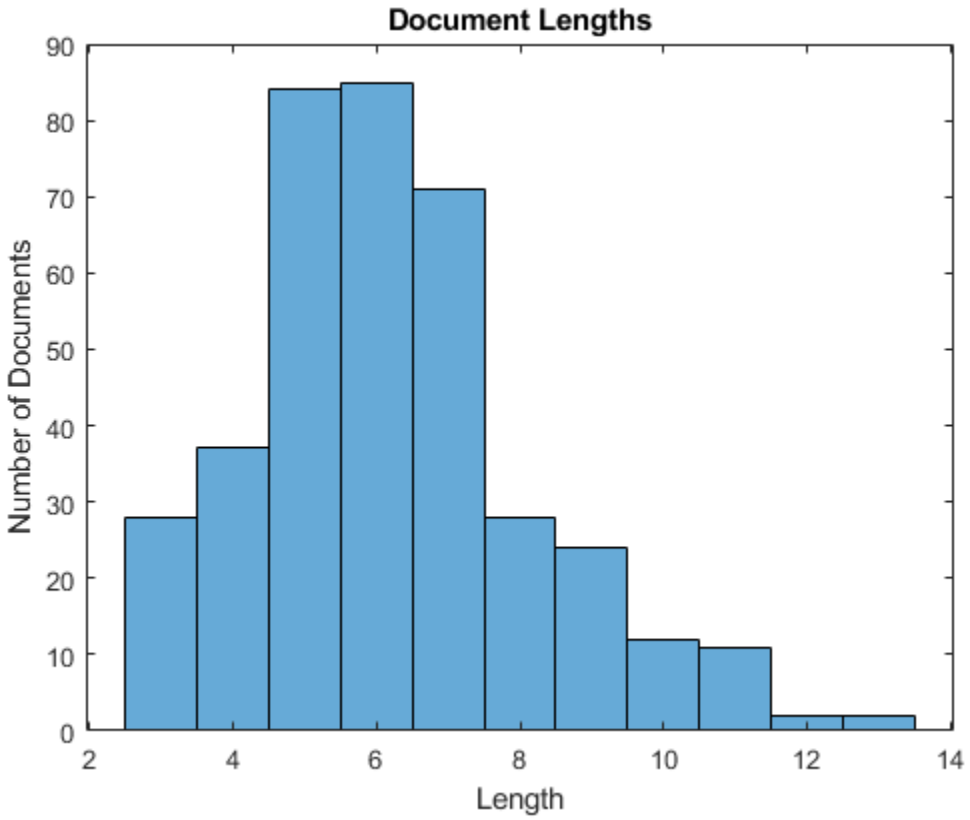
To create a word encoding, use the `wordEncoding` function.

```
enc = wordEncoding(documentsTrain);
```

The next conversion step is to pad and truncate documents so they are all the same length. The `trainingOptions` function provides options to pad and truncate input sequences automatically. However, these options are not well suited for sequences of word vectors. Instead, pad and truncate the sequences manually. If you *left-pad* and truncate the sequences of word vectors, then the training might improve.

To pad and truncate the documents, first choose a target length, and then truncate documents that are longer than it and left-pad documents that are shorter than it. For best results, the target length should be short without discarding large amounts of data. To find a suitable target length, view a histogram of the training document lengths.

```
documentLengths = doclength(documentsTrain);
figure
histogram(documentLengths)
title("Document Lengths")
xlabel("Length")
ylabel("Number of Documents")
```



Most of the training documents have fewer than 10 tokens. Use this as your target length for truncation and padding.

Convert the documents to sequences of numeric indices using `doc2sequence`. To truncate or left-pad the sequences to have length 10, set the `'Length'` option to 10.

```
sequenceLength = 10;
XTrain = doc2sequence(enc,documentsTrain,'Length',sequenceLength);
XTrain(1:5)
```

```
ans=5×1 cell array
    {1×10 double}
    {1×10 double}
    {1×10 double}
    {1×10 double}
    {1×10 double}
```

Convert the validation documents to sequences using the same options.

```
XValidation = doc2sequence(enc,documentsValidation,'Length',sequenceLength);
```

Create and Train LSTM Network

Define the LSTM network architecture. To input sequence data into the network, include a sequence input layer and set the input size to 1. Next, include a word embedding layer of dimension 50 and the same number of words as the word encoding. Next, include an LSTM layer and set the number of hidden units to 80. To use the LSTM layer for a sequence-to-label classification problem, set the output mode to `'last'`. Finally, add a fully connected layer with the same size as the number of classes, a softmax layer, and a classification layer.

```
inputSize = 1;
embeddingDimension = 50;
numHiddenUnits = 80;

numWords = enc.NumWords;
numClasses = numel(categories(YTrain));

layers = [ ...
    sequenceInputLayer(inputSize)
    wordEmbeddingLayer(embeddingDimension,numWords)
    lstmLayer(numHiddenUnits,'OutputMode','last')
    fullyConnectedLayer(numClasses)
    softmaxLayer
    classificationLayer]
```

```
layers =
  6x1 Layer array with layers:

     1   ''   Sequence Input          Sequence input with 1 dimensions
     2   ''   Word Embedding Layer    Word embedding layer with 50 dimensions and 423 unique word
     3   ''   LSTM                   LSTM with 80 hidden units
     4   ''   Fully Connected         4 fully connected layer
     5   ''   Softmax                softmax
     6   ''   Classification Output   crossentropyex
```

Specify Training Options

Specify the training options:

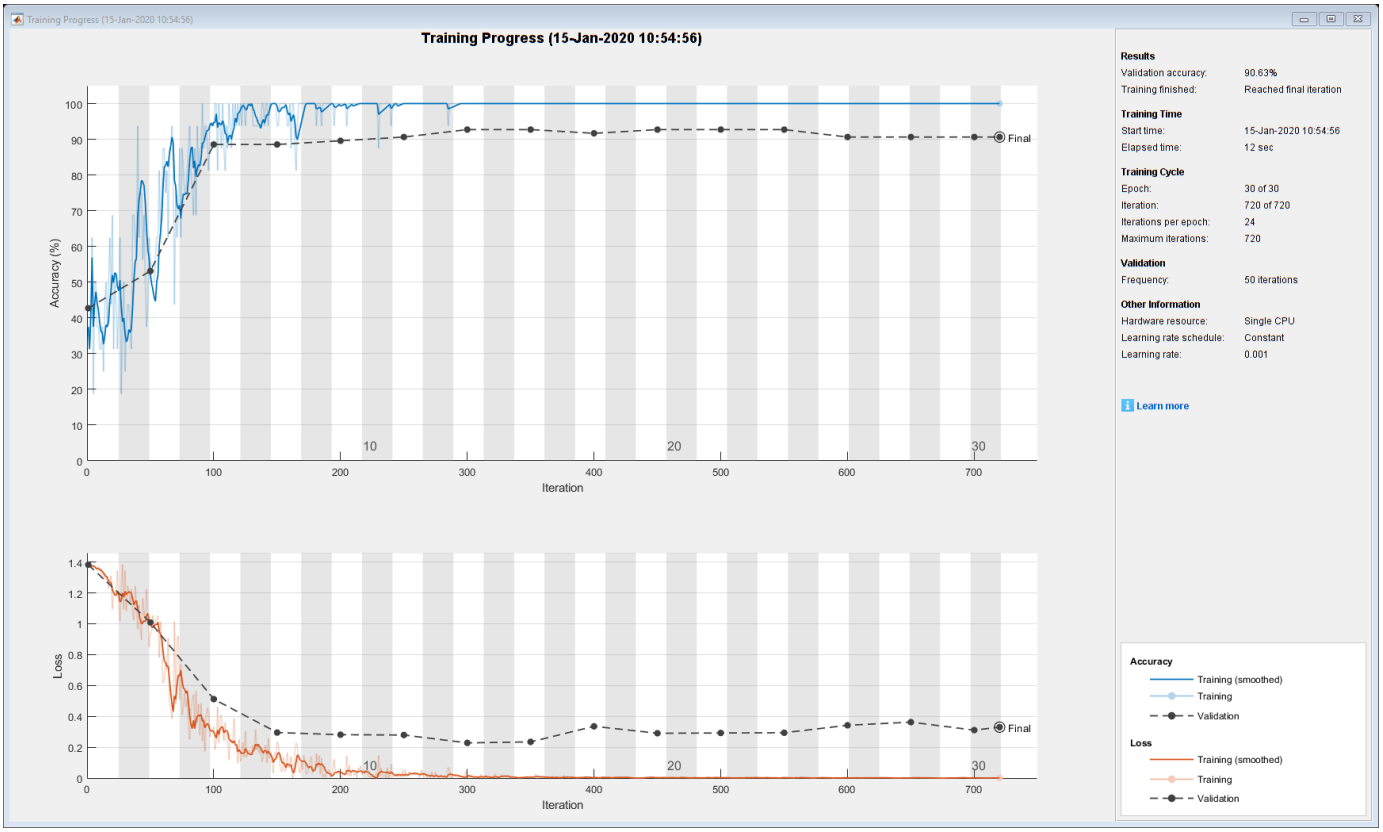
- Train using the Adam solver.
- Specify a mini-batch size of 16.
- Shuffle the data every epoch.
- Monitor the training progress by setting the `'Plots'` option to `'training-progress'`.
- Specify the validation data using the `'ValidationData'` option.
- Suppress verbose output by setting the `'Verbose'` option to `false`.

By default, `trainNetwork` uses a GPU if one is available (requires Parallel Computing Toolbox™ and a CUDA® enabled GPU with compute capability 3.0 or higher). Otherwise, it uses the CPU. To specify the execution environment manually, use the `'ExecutionEnvironment'` name-value pair argument of `trainingOptions`. Training on a CPU can take significantly longer than training on a GPU.

```
options = trainingOptions('adam', ...
    'MiniBatchSize',16, ...
    'GradientThreshold',2, ...
    'Shuffle','every-epoch', ...
    'ValidationData',{XValidation,YValidation}, ...
    'Plots','training-progress', ...
    'Verbose',false);
```

Train the LSTM network using the `trainNetwork` function.

```
net = trainNetwork(XTrain,YTrain,layers,options);
```



Predict Using New Data

Classify the event type of three new reports. Create a string array containing the new reports.

```
reportsNew = [ ...
    "Coolant is pooling underneath sorter."
    "Sorter blows fuses at start up."
    "There are some very loud rattling sounds coming from the assembler."];
```

Preprocess the text data using the preprocessing steps as the training documents.

```
documentsNew = preprocessText(reportsNew);
```

Convert the text data to sequences using `doc2sequence` with the same options as when creating the training sequences.

```
XNew = doc2sequence(enc,documentsNew,'Length',sequenceLength);
```

Classify the new sequences using the trained LSTM network.

```
labelsNew = classify(net,XNew)
```

```
labelsNew = 3×1 categorical
     Leak
     Electronic Failure
     Mechanical Failure
```

Preprocessing Function

The function `preprocessText` performs these steps:

1. Tokenize the text using `tokenizedDocument`.
2. Convert the text to lowercase using `lower`.
3. Erase the punctuation using `erasePunctuation`.

```
function documents = preprocessText(textData)

% Tokenize the text.
documents = tokenizedDocument(textData);

% Convert to lowercase.
documents = lower(documents);

% Erase punctuation.
documents = erasePunctuation(documents);

end
```

See Also

`doc2sequence` | `fastTextWordEmbedding` | `lstmLayer` | `sequenceInputLayer` | `tokenizedDocument` | `trainNetwork` | `trainingOptions` | `wordEmbeddingLayer` | `wordcloud`

Related Examples

- “Classify Text Data Using Convolutional Neural Network” on page 2-73

- “Classify Out-of-Memory Text Data Using Deep Learning” on page 2-130
- “Generate Text Using Deep Learning” (Deep Learning Toolbox)
- “Word-By-Word Text Generation Using Deep Learning” on page 2-142
- “Create Simple Text Model for Classification” on page 2-2
- “Analyze Text Data Using Topic Models” on page 2-13
- “Analyze Text Data Using Multiword Phrases” on page 2-7
- “Train a Sentiment Classifier” on page 2-51
- “Sequence Classification Using Deep Learning” (Deep Learning Toolbox)
- “Deep Learning in MATLAB” (Deep Learning Toolbox)

Classify Text Data Using Convolutional Neural Network

This example shows how to classify text data using a convolutional neural network.

To classify text data using convolutions, you must convert the text data into images. To do this, pad or truncate the observations to have constant length *S* and convert the documents into sequences of word vectors of length *C* using a word embedding. You can then represent a document as a 1-by-*S*-by-*C* image (an image with height 1, width *S*, and *C* channels).

To convert text data from a CSV file to images, create a `tabularTextDatastore` object. The convert the data read from the `tabularTextDatastore` object to images for deep learning by calling `transform` with a custom transformation function. The `transformTextData` function, listed at the end of the example, takes data read from the datastore and a pretrained word embedding, and converts each observation to an array of word vectors.

This example trains a network with 1-D convolutional filters of varying widths. The width of each filter corresponds the number of words the filter can see (the n-gram length). The network has multiple branches of convolutional layers, so it can use different n-gram lengths.

Load Pretrained Word Embedding

Load the pretrained fastText word embedding. This function requires the Text Analytics Toolbox™ Model *for fastText English 16 Billion Token Word Embedding* support package. If this support package is not installed, then the function provides a download link.

```
emb = fastTextWordEmbedding;
```

Load Data

Create a tabular text datastore from the data in `factoryReports.csv`. Read the data from the `"Description"` and `"Category"` columns only.

```
filenameTrain = "factoryReports.csv";
textName = "Description";
labelName = "Category";
ttdsTrain = tabularTextDatastore(filenameTrain,'SelectedVariableNames',[textName labelName]);
```

Preview the datastore.

```
ttdsTrain.ReadSize = 8;
preview(ttdsTrain)
```

```
ans=8×2 table
                                  Description                                      Category
    ______________________________________________________________________    ____________

    {'Items are occasionally getting stuck in the scanner spools.'       }    {'Mechanical Failu
    {'Loud rattling and banging sounds are coming from assembler pistons.'}    {'Mechanical Failu
    {'There are cuts to the power when starting the plant.'              }    {'Electronic Failu
    {'Fried capacitors in the assembler.'                                }    {'Electronic Failu
    {'Mixer tripped the fuses.'                                          }    {'Electronic Failu
    {'Burst pipe in the constructing agent is spraying coolant.'         }    {'Leak'
    {'A fuse is blown in the mixer.'                                     }    {'Electronic Failu
    {'Things continue to tumble off of the belt.'                        }    {'Mechanical Failu
```

Create a custom transform function that converts data read from the datastore to a table containing the predictors and the responses. The `transformTextData` function, listed at the end of the example, takes the data read from a `tabularTextDatastore` object and returns a table of predictors and responses. The predictors are 1-by-`sequenceLength`-by-*C* arrays of word vectors given by the word embedding `emb`, where *C* is the embedding dimension. The responses are categorical labels over the classes in `classNames`.

Read the labels from the training data using the `readLabels` function, listed at the end of the example, and find the unique class names.

```
labels = readLabels(ttdsTrain,labelName);
classNames = unique(labels);
numObservations = numel(labels);
```

Transform the datastore using `transformTextData` function and specify a sequence length of 14.

```
sequenceLength = 14;
tdsTrain = transform(ttdsTrain, @(data) transformTextData(data,sequenceLength,emb,classNames))
```

```
tdsTrain =
  TransformedDatastore with properties:

       UnderlyingDatastore: [1×1 matlab.io.datastore.TabularTextDatastore]
    SupportedOutputFormats: ["txt"    "csv"    "xlsx"    "xls"    "parquet"    "parq"    "png"
                Transforms: {@(data)transformTextData(data,sequenceLength,emb,classNames)}
              IncludeInfo: 0
```

Preview the transformed datastore. The predictors are 1-by-*S*-by-*C* arrays, where *S* is the sequence length and *C* is the number of features (the embedding dimension). The responses are the categorical labels.

```
preview(tdsTrain)
```

```
ans=8×2 table
       Predictors            Responses
    _________________    _________________

    {1×14×300 single}    Mechanical Failure
    {1×14×300 single}    Mechanical Failure
    {1×14×300 single}    Electronic Failure
    {1×14×300 single}    Electronic Failure
    {1×14×300 single}    Electronic Failure
    {1×14×300 single}    Leak
    {1×14×300 single}    Electronic Failure
    {1×14×300 single}    Mechanical Failure
```

Define Network Architecture

Define the network architecture for the classification task.

The following steps describe the network architecture.

- Specify an input size of 1-by-*S*-by-*C*, where *S* is the sequence length and *C* is the number of features (the embedding dimension).
- For the n-gram lengths 2, 3, 4, and 5, create blocks of layers containing a convolutional layer, a batch normalization layer, a ReLU layer, a dropout layer, and a max pooling layer.

- For each block, specify 200 convolutional filters of size 1-by-*N* and pooling regions of size 1-by-*S*, where *N* is the n-gram length.
- Connect the input layer to each block and concatenate the outputs of the blocks using a depth concatenation layer.
- To classify the outputs, include a fully connected layer with output size *K*, a softmax layer, and a classification layer, where *K* is the number of classes.

First, in a layer array, specify the input layer, the first block for unigrams, the depth concatenation layer, the fully connected layer, the softmax layer, and the classification layer.

```
numFeatures = emb.Dimension;
inputSize = [1 sequenceLength numFeatures];
numFilters = 200;

ngramLengths = [2 3 4 5];
numBlocks = numel(ngramLengths);

numClasses = numel(classNames);
```

Create a layer graph containing the input layer. Set the normalization option to `'none'` and the layer name to `'input'`.

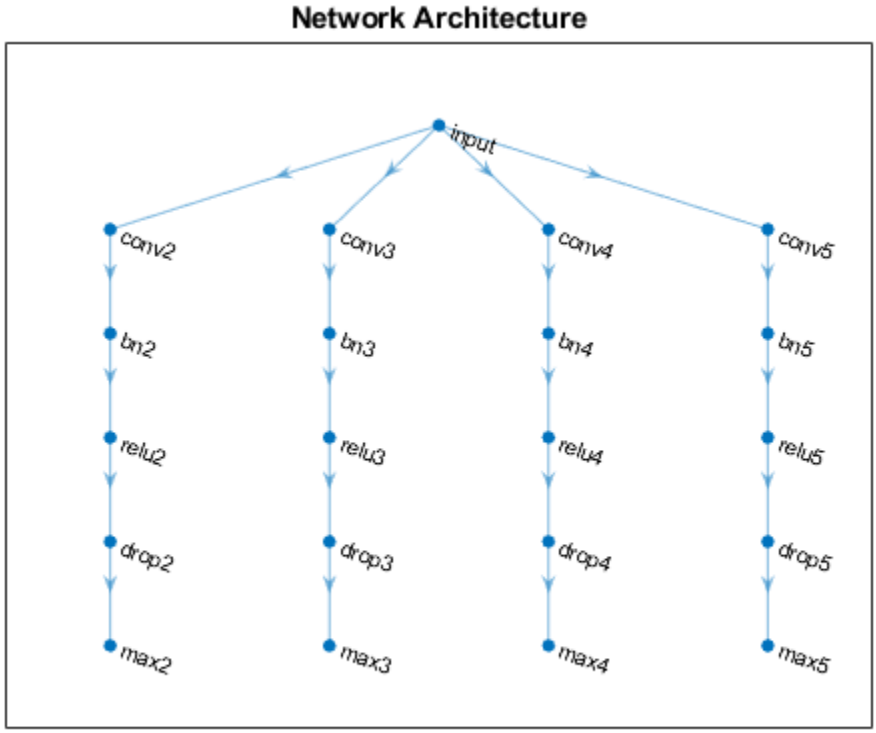
```
layer = imageInputLayer(inputSize,'Normalization','none','Name','input');
lgraph = layerGraph(layer);
```

For each of the n-gram lengths, create a block of convolution, batch normalization, ReLU, dropout, and max pooling layers. Connect each block to the input layer.

```
for j = 1:numBlocks
    N = ngramLengths(j);

    block = [
        convolution2dLayer([1 N],numFilters,'Name',"conv"+N,'Padding','same')
        batchNormalizationLayer('Name',"bn"+N)
        reluLayer('Name',"relu"+N)
        dropoutLayer(0.2,'Name',"drop"+N)
        maxPooling2dLayer([1 sequenceLength],'Name',"max"+N)];

    lgraph = addLayers(lgraph,block);
    lgraph = connectLayers(lgraph,'input',"conv"+N);
end
```

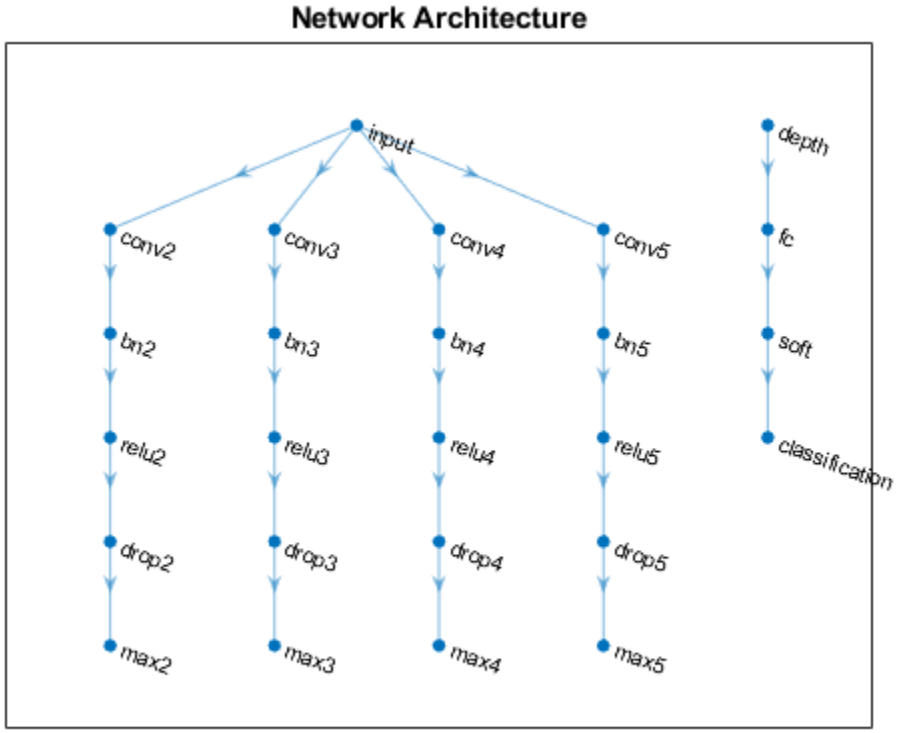
View the network architecture in a plot.

```
figure
plot(lgraph)
title("Network Architecture")
```



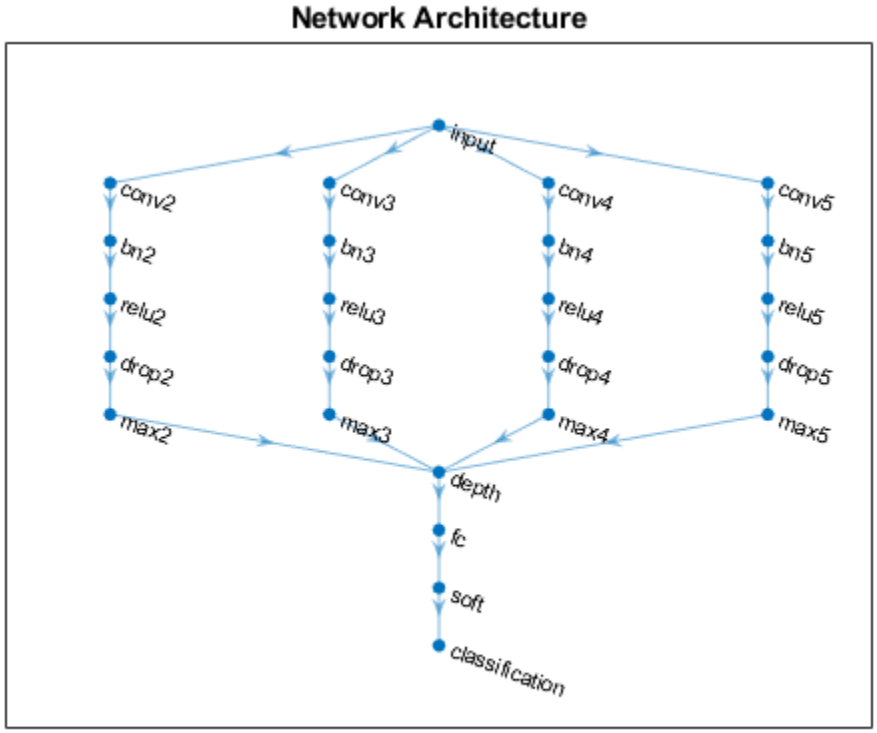
Add the depth concatenation layer, the fully connected layer, the softmax layer, and the classification layer.

```
layers = [
    depthConcatenationLayer(numBlocks,'Name','depth')
    fullyConnectedLayer(numClasses,'Name','fc')
    softmaxLayer('Name','soft')
    classificationLayer('Name','classification')];

lgraph = addLayers(lgraph,layers);

figure
plot(lgraph)
title("Network Architecture")
```



Connect the max pooling layers to the depth concatenation layer and view the final network architecture in a plot.

```
for j = 1:numBlocks
    N = ngramLengths(j);
    lgraph = connectLayers(lgraph,"max"+N,"depth/in"+j);
end

figure
plot(lgraph)
title("Network Architecture")
```



Train Network

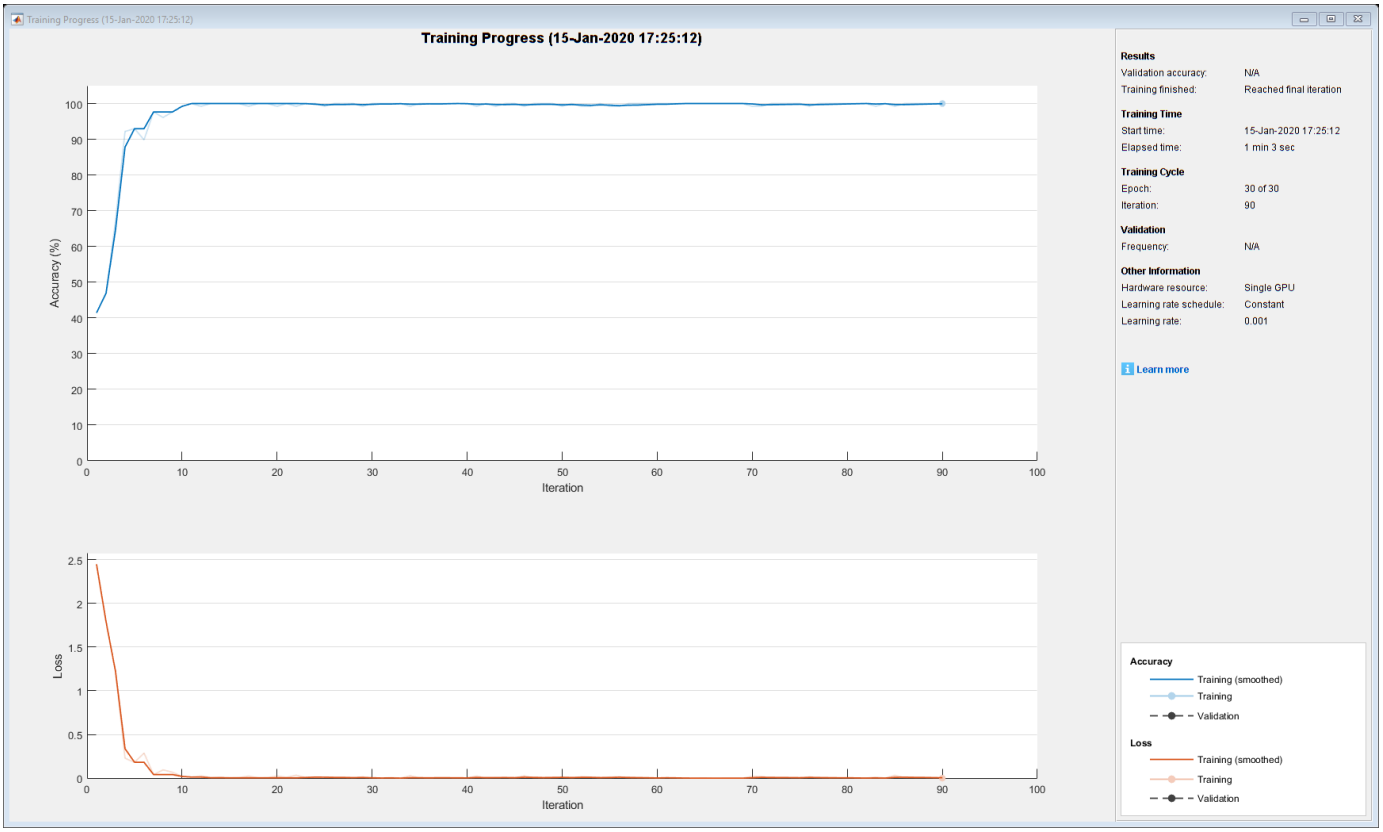
Specify the training options:

- Train with a mini-batch size of 128.
- Do not shuffle the data because the datastore is not shuffleable.
- Display the training progress plot and suppress the verbose output.

```
miniBatchSize = 128;
numIterationsPerEpoch = floor(numObservations/miniBatchSize);

options = trainingOptions('adam', ...
    'MiniBatchSize',miniBatchSize, ...
    'Shuffle','never', ...
    'Plots','training-progress', ...
    'Verbose',false);
```

Train the network using the `trainNetwork` function.

```
net = trainNetwork(tdsTrain,lgraph,options);
```



Predict Using New Data

Classify the event type of three new reports. Create a string array containing the new reports.

```
reportsNew = [
    "Coolant is pooling underneath sorter."
    "Sorter blows fuses at start up."
    "There are some very loud rattling sounds coming from the assembler."];
```

Preprocess the text data using the preprocessing steps as the training documents.

```
XNew = preprocessText(reportsNew,sequenceLength,emb);
```

Classify the new sequences using the trained LSTM network.

```
labelsNew = classify(net,XNew)
```

```
labelsNew = 3×1 categorical
     Leak
     Electronic Failure
     Mechanical Failure
```

Read Labels Function

The `readLabels` function creates a copy of the `tabularTextDatastore` object `ttds` and reads the labels from the `labelName` column.

```
function labels = readLabels(ttds,labelName)
```

```
ttdsNew = copy(ttds);
ttdsNew.SelectedVariableNames = labelName;
tbl = readall(ttdsNew);
labels = tbl.(labelName);

end
```

Transform Text Data Function

The `transformTextData` function takes the data read from a `tabularTextDatastore` object and returns a table of predictors and responses. The predictors are 1-by-`sequenceLength`-by-*C* arrays of word vectors given by the word embedding `emb`, where *C* is the embedding dimension. The responses are categorical labels over the classes in `classNames`.

```
function dataTransformed = transformTextData(data,sequenceLength,emb,classNames)

% Preprocess documents.
textData = data{:,1};

% Prepocess text
dataTransformed = preprocessText(textData,sequenceLength,emb);

% Read labels.
labels = data{:,2};
responses = categorical(labels,classNames);

% Convert data to table.
dataTransformed.Responses = responses;

end
```

Preprocess Text Function

The `preprocessTextData` function takes text data, a sequence length, and a word embedding and performs these steps:

1. Tokenize the text.
2. Convert the text to lowercase.
3. Converts the documents to sequences of word vectors of the specified length using the embedding.
4. Reshapes the word vector sequences to input into the network.

```
function tbl = preprocessText(textData,sequenceLength,emb)

documents = tokenizedDocument(textData);
documents = lower(documents);

% Convert documents to embeddingDimension-by-sequenceLength-by-1 images.
predictors = doc2sequence(emb,documents,'Length',sequenceLength);

% Reshape images to be of size 1-by-sequenceLength-embeddingDimension.
predictors = cellfun(@(X) permute(X,[3 2 1]),predictors,'UniformOutput',false);

tbl = table;
tbl.Predictors = predictors;
```

```
end
```

See Also

`batchNormalizationLayer` | `convolution2dLayer` | `doc2sequence` | `fastTextWordEmbedding` | `layerGraph` | `tokenizedDocument` | `trainNetwork` | `trainingOptions` | `wordEmbedding` | `wordcloud`

Related Examples

- “Classify Text Data Using Deep Learning” on page 2-65
- “Classify Out-of-Memory Text Data Using Deep Learning” on page 2-130
- “Create Simple Text Model for Classification” on page 2-2
- “Analyze Text Data Using Topic Models” on page 2-13
- “Analyze Text Data Using Multiword Phrases” on page 2-7
- “Train a Sentiment Classifier” on page 2-51
- “Sequence Classification Using Deep Learning” (Deep Learning Toolbox)
- “Deep Learning in MATLAB” (Deep Learning Toolbox)

Classify Text Data Using Custom Training Loop

This example shows how to classify text data using a deep learning bidirectional long short-term memory (BiLSTM) network with a custom training loop.

When training a deep learning network using the `trainNetwork` function, if `trainingOptions` does not provide the options you need (for example, a custom learning rate schedule), then you can define your own custom training loop using automatic differentiation. For an example showing how to classify text data using the `trainNetwork` function, see “Classify Text Data Using Deep Learning” (Deep Learning Toolbox).

This example trains a network to classify text data with the *time-based decay* learning rate schedule: for each iteration, the solver uses the learning rate given by $\rho_t = \frac{\rho_0}{1 + kt}$, where t is the iteration number, ρ_0 is the initial learning rate, and k is the decay.

Import Data

Import the factory reports data. This data contains labeled textual descriptions of factory events. To import the text data as strings, specify the text type to be `'string'`.

```
filename = "factoryReports.csv";
data = readtable(filename,'TextType','string');
head(data)
```

```
ans=8×5 table
                                Description                                 
    ____________________________________________________________________    

    "Items are occasionally getting stuck in the scanner spools."           
    "Loud rattling and banging sounds are coming from assembler pistons."   
    "There are cuts to the power when starting the plant."                  
    "Fried capacitors in the assembler."                                    
    "Mixer tripped the fuses."                                              
    "Burst pipe in the constructing agent is spraying coolant."             
    "A fuse is blown in the mixer."                                         
    "Things continue to tumble off of the belt."                            
```

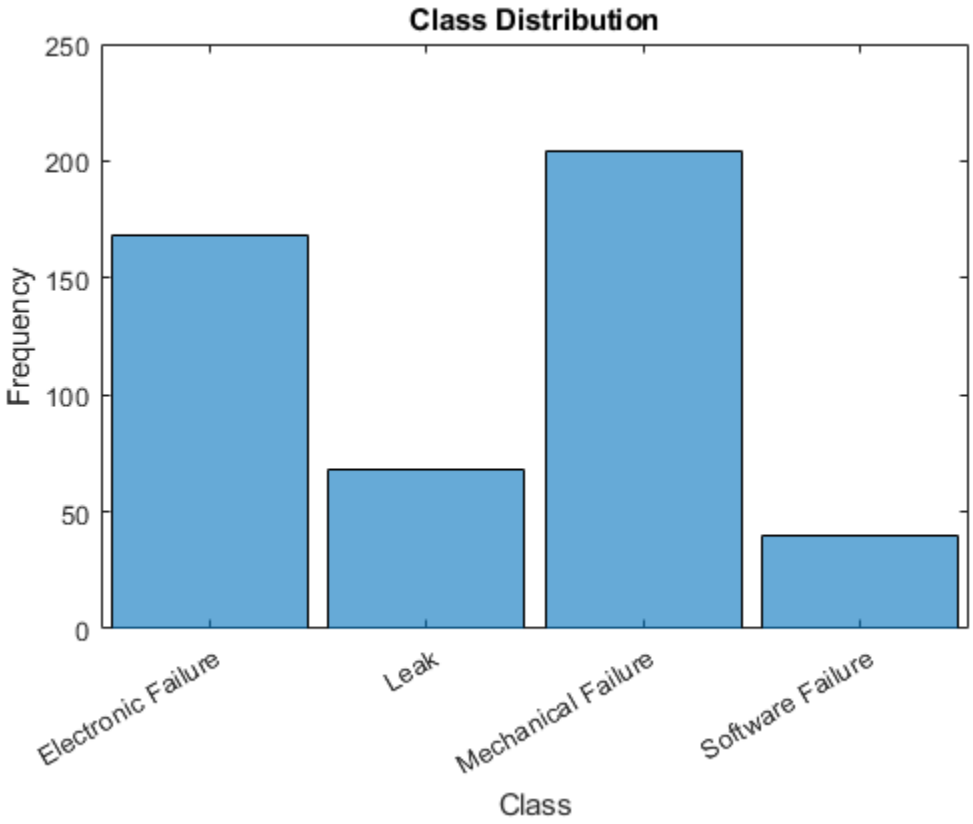
```
                Category         
    _____________________________

    "Mechanical Failure"
    "Mechanical Failure"
    "Electronic Failure"
    "Electronic Failure"
    "Electronic Failure"
    "Leak"
    "Electronic Failure"
    "Mechanical Failure"
```

The goal of this example is to classify events by the label in the `Category` column. To divide the data into classes, convert these labels to categorical.

```
data.Category = categorical(data.Category);
```

View the distribution of the classes in the data using a histogram.

```
figure
histogram(data.Category);
xlabel("Class")
ylabel("Frequency")
title("Class Distribution")
```



The next step is to partition it into sets for training and validation. Partition the data into a training partition and a held-out partition for validation and testing. Specify the holdout percentage to be 20%.

```
cvp = cvpartition(data.Category,'Holdout',0.2);
dataTrain = data(training(cvp),:);
dataValidation = data(test(cvp),:);
```

Extract the text data and labels from the partitioned tables.

```
textDataTrain = dataTrain.Description;
textDataValidation = dataValidation.Description;
YTrain = dataTrain.Category;
YValidation = dataValidation.Category;
```

To check that you have imported the data correctly, visualize the training text data using a word cloud.

```
figure
wordcloud(textDataTrain);
title("Training Data")
```

View the number of classes.

```
classes = categories(YTrain);
numClasses = numel(classes)
```

```
numClasses = 4
```

Preprocess Text Data

Create a function that tokenizes and preprocesses the text data. The function `preprocessText`, listed at the end of the example, performs these steps:

1. Tokenize the text using `tokenizedDocument`.
2. Convert the text to lowercase using `lower`.
3. Erase the punctuation using `erasePunctuation`.

Preprocess the training data and the validation data using the `preprocessText` function.

```
documentsTrain = preprocessText(textDataTrain);
documentsValidation = preprocessText(textDataValidation);
```

View the first few preprocessed training documents.

```
documentsTrain(1:5)
```

```
ans =
  5×1 tokenizedDocument:
```

```
     9 tokens: items are occasionally getting stuck in the scanner spools
    10 tokens: loud rattling and banging sounds are coming from assembler pistons
     5 tokens: fried capacitors in the assembler
     4 tokens: mixer tripped the fuses
     9 tokens: burst pipe in the constructing agent is spraying coolant
```

Create a single datastore that contains both the documents and the labels by creating `arrayDatastore` objects, then combining them using the `combine` function.

```
dsDocumentsTrain = arrayDatastore(documentsTrain,'OutputType','cell');
dsYTrain = arrayDatastore(YTrain,'OutputType','cell');
dsTrain = combine(dsDocumentsTrain,dsYTrain);
```

Create a datastore for the validation data using the same steps.

```
dsDocumentsValidation = arrayDatastore(documentsValidation,'OutputType','cell');
dsYValidation = arrayDatastore(YValidation,'OutputType','cell');
dsValidation = combine(dsDocumentsValidation,dsYValidation);
```

Create Word Encoding

To input the documents into a BiLSTM network, use a word encoding to convert the documents into sequences of numeric indices.

To create a word encoding, use the `wordEncoding` function.

```
enc = wordEncoding(documentsTrain)
```

```
enc =
  wordEncoding with properties:

      NumWords: 421
    Vocabulary: [1×421 string]
```

Define Network

Define the BiLSTM network architecture. To input sequence data into the network, include a sequence input layer and set the input size to 1. Next, include a word embedding layer of dimension 25 and the same number of words as the word encoding. Next, include a BiLSTM layer and set the number of hidden units to 40. To use the BiLSTM layer for a sequence-to-label classification problem, set the output mode to `'last'`. Finally, add a fully connected layer with the same size as the number of classes, and a softmax layer.

```
inputSize = 1;
embeddingDimension = 25;
numHiddenUnits = 40;

numWords = enc.NumWords;

layers = [
    sequenceInputLayer(inputSize,'Name','in')
    wordEmbeddingLayer(embeddingDimension,numWords,'Name','emb')
    bilstmLayer(numHiddenUnits,'OutputMode','last','Name','bilstm')
    fullyConnectedLayer(numClasses,'Name','fc')
    softmaxLayer('Name','sm')]
```

```
layers =
  5×1 Layer array with layers:

     1   'in'       Sequence Input         Sequence input with 1 dimensions
     2   'emb'      Word Embedding Layer   Word embedding layer with 25 dimensions and 421 unique
     3   'bilstm'   BiLSTM                 BiLSTM with 40 hidden units
     4   'fc'       Fully Connected        4 fully connected layer
     5   'sm'       Softmax                softmax
```

Convert the layer array to a layer graph and create a `dlnetwork` object.

```
lgraph = layerGraph(layers);
dlnet = dlnetwork(lgraph)
```

```
dlnet =
  dlnetwork with properties:

         Layers: [5×1 nnet.cnn.layer.Layer]
    Connections: [4×2 table]
     Learnables: [6×3 table]
          State: [2×3 table]
     InputNames: {'in'}
    OutputNames: {'sm'}
```

Define Model Gradients Function

Create the function `modelGradients`, listed at the end of the example, that takes a `dlnetwork` object, a mini-batch of input data with corresponding labels, and returns the gradients of the loss with respect to the learnable parameters in the network and the corresponding loss.

Specify Training Options

Train for 30 epochs with a mini-batch size of 16.

```
numEpochs = 30;
miniBatchSize = 16;
```

Specify the options for Adam optimization. Specify an initial learn rate of 0.001 with a decay of 0.01, gradient decay factor 0.9, and squared gradient decay factor 0.999.

```
initialLearnRate = 0.001;
decay = 0.01;
gradientDecayFactor = 0.9;
squaredGradientDecayFactor = 0.999;
```

Train Model

Train the model using a custom training loop.

Initialize the training progress plot.

```
figure
lineLossTrain = animatedline('Color',[0.85 0.325 0.098]);

lineLossValidation = animatedline( ...
    'LineStyle','--', ...
    'Marker','o', ...
```

```
    'MarkerFaceColor','black');

ylim([0 inf])
xlabel("Iteration")
ylabel("Loss")
grid on
```

Initialize the parameters for Adam.

```
trailingAvg = [];
trailingAvgSq = [];
```

Create a `minibatchqueue` object that processes and manages the mini-batches of data. For each mini-batch:

- Use the custom mini-batch preprocessing function `preprocessMiniBatch` (defined at the end of this example) to convert documents to sequences and one-hot encode the labels. To pass the word encoding to the mini-batch, create an anonymous function that takes two inputs.
- Format the predictors with the dimension labels `'BTC'` (batch, time, channel). The `minibatchqueue` object, by default, converts the data to `dlarray` objects with underlying type `single`.
- Train on a GPU if one is available. The `minibatchqueue` object, by default, converts each output to `gpuArray` if a GPU is available. Using a GPU requires Parallel Computing Toolbox™ and a CUDA® enabled NVIDIA® GPU with compute capability 3.0 or higher.

```
mbq = minibatchqueue(dsTrain, ...
    'MiniBatchSize',miniBatchSize,...
    'MiniBatchFcn', @(X,Y) preprocessMiniBatch(X,Y,enc), ...
    'MiniBatchFormat',{'BTC',''});
```

Create a `minibatchqueue` object for the validation data using the same options and also specify to return partial mini-batches.

```
mbqValidation = minibatchqueue(dsValidation, ...
    'MiniBatchSize',miniBatchSize, ...
    'MiniBatchFcn', @(X,Y) preprocessMiniBatch(X,Y,enc), ...
    'MiniBatchFormat',{'BTC',''}, ...
    'PartialMiniBatch','return');
```

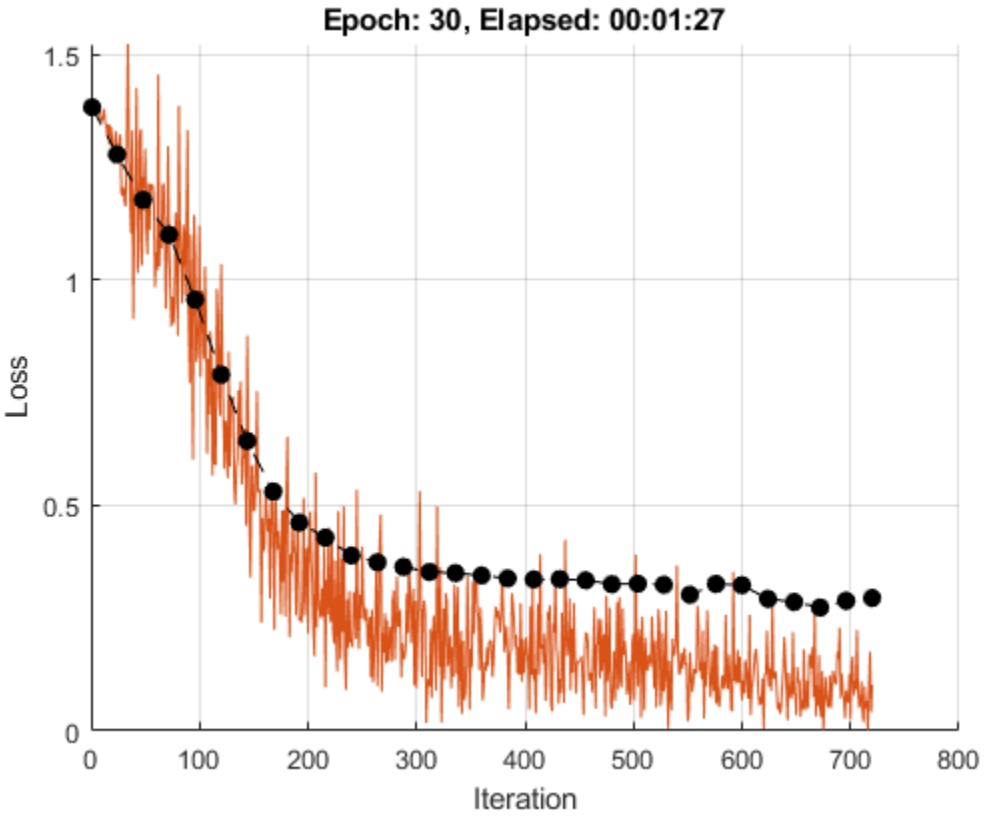
Train the network. For each epoch, shuffle the data and loop over mini-batches of data. At the end of each iteration, display the training progress. At the end of each epoch, validate the network using the validation data.

For each mini-batch:

- Convert the documents to sequences of integers and one-hot encode the labels.
- Convert the data to `dlarray` objects with underlying type single and specify the dimension labels `'BTC'` (batch, time, channel).
- For GPU training, convert to `gpuArray` objects.
- Evaluate the model gradients, state, and loss using `dlfeval` and the `modelGradients` function and update the network state.
- Determine the learning rate for the time-based decay learning rate schedule.
- Update the network parameters using the `adamupdate` function.

- Update the training plot.

```
iteration = 0;
start = tic;

% Loop over epochs.
for epoch = 1:numEpochs

    % Shuffle data.
    shuffle(mbq);

    % Loop over mini-batches.
    while hasdata(mbq)
        iteration = iteration + 1;

        % Read mini-batch of data.
        [dlX, dlY] = next(mbq);

        % Evaluate the model gradients, state, and loss using dlfeval and the
        % modelGradients function.
        [gradients,loss] = dlfeval(@modelGradients,dlnet,dlX,dlY);

        % Determine learning rate for time-based decay learning rate schedule.
        learnRate = initialLearnRate/(1 + decay*iteration);

        % Update the network parameters using the Adam optimizer.
        [dlnet,trailingAvg,trailingAvgSq] = adamupdate(dlnet, gradients, ...
            trailingAvg, trailingAvgSq, iteration, learnRate, ...
            gradientDecayFactor, squaredGradientDecayFactor);

        % Display the training progress.
        D = duration(0,0,toc(start),'Format','hh:mm:ss');
        addpoints(lineLossTrain,iteration,loss)
        title("Epoch: " + epoch + ", Elapsed: " + string(D))
        drawnow

        % Validate network.
        if iteration == 1 || ~hasdata(mbq)
            % Validation predictions.
            [~,lossValidation] = modelPredictions(dlnet,mbqValidation,classes);

            % Update plot.
            addpoints(lineLossValidation,iteration,lossValidation)
            drawnow
        end
    end
end
```



Test Model

Test the classification accuracy of the model by comparing the predictions on the validation set with the true labels.

Classify the validation data using `modelPredictions` function, listed at the end of the example.

```
dlYPred = modelPredictions(dlnet,mbqValidation,classes);
YPred = onehotdecode(dlYPred,classes,1)';
```

Evaluate the classification accuracy.

```
accuracy = mean(YPred == YValidation)
```

```
accuracy = 0.9167
```

Predict Using New Data

Classify the event type of three new reports. Create a string array containing the new reports.

```
reportsNew = [
    "Coolant is pooling underneath sorter."
    "Sorter blows fuses at start up."
    "There are some very loud rattling sounds coming from the assembler."];
```

Preprocess the text data using the preprocessing steps as the training documents.

```
documentsNew = preprocessText(reportsNew);
dsNew = arrayDatastore(documentsNew,'OutputType','cell');
```

Create a `minibatchqueue` object that processes and manages the mini-batches of data. For each mini-batch:

- Use the custom mini-batch preprocessing function `preprocessMiniBatchPredictors` (defined at the end of this example) to convert documents to sequences. This preprocessing function does not require label data. To pass the word encoding to the mini-batch, create an anonymous function that takes one input only.
- Format the predictors with the dimension labels `'BTC'` (batch, time, channel). The `minibatchqueue` object, by default, converts the data to `dlarray` objects with underlying type `single`.
- To make predictions for all observations, return any partial mini-batches.

```
mbqNew = minibatchqueue(dsNew, ...
    'MiniBatchSize',miniBatchSize, ...
    'MiniBatchFcn',@(X) preprocessMiniBatchPredictors(X,enc), ...
    'MiniBatchFormat','BTC', ...
    'PartialMiniBatch','return');
```

Classify the text data using `modelPredictions` function, listed at the end of the example and find the classes with the highest scores.

```
dlYPred = modelPredictions(dlnet,mbqNew,classes);
YPred = onehotdecode(dlYPred,classes,1)'
```

```
YPred = 3×1 categorical
     Leak
     Electronic Failure
     Mechanical Failure
```

Text Preprocessing Function

The function `preprocessText` performs these steps:

1. Tokenize the text using `tokenizedDocument`.
2. Convert the text to lowercase using `lower`.
3. Erase the punctuation using `erasePunctuation`.

```
function documents = preprocessText(textData)

% Tokenize the text.
documents = tokenizedDocument(textData);

% Convert to lowercase.
documents = lower(documents);

% Erase punctuation.
documents = erasePunctuation(documents);

end
```

Mini-Batch Preprocessing Function

The `preprocessMiniBatch` function converts a mini-batch of documents to sequences of integers and one-hot encodes label data.

```
function [X, Y] = preprocessMiniBatch(documentsCell,labelsCell,enc)

% Preprocess predictors.
X = preprocessMiniBatchPredictors(documentsCell,enc);

% Extract labels from cell and concatenate.
Y = cat(1,labelsCell{1:end});

% One-hot encode labels.
Y = onehotencode(Y,2);

% Transpose the encoded labels to match the network output.
Y = Y';

end
```

Mini-Batch Predictors Preprocessing Function

The `preprocessMiniBatchPredictors` function converts a mini-batch of documents to sequences of integers.

```
function X = preprocessMiniBatchPredictors(documentsCell,enc)

% Extract documents from cell and concatenate.
documents = cat(4,documentsCell{1:end});

% Convert documents to sequences of integers.
X = doc2sequence(enc,documents);
X = cat(1,X{:});

end
```

Model Gradients Function

The `modelGradients` function takes a `dlnetwork` object `dlnet`, a mini-batch of input data `dlX` with corresponding target labels `T` and returns the gradients of the loss with respect to the learnable parameters in `dlnet`, and the loss. To compute the gradients automatically, use the `dlgradient` function.

```
function [gradients,loss] = modelGradients(dlnet,dlX,T)

dlYPred = forward(dlnet,dlX);

loss = crossentropy(dlYPred,T);
gradients = dlgradient(loss,dlnet.Learnables);

loss = double(gather(extractdata(loss)));

end
```

Model Predictions Function

The `modelPredictions` function takes a `dlnetwork` object `dlnet`, a mini-batch queue, and outputs the model predictions by iterating over mini-batches in the queue. To evaluate validation data, this function optionally calcualtes the loss when given a mini-batch queue with two outputs.

```
function [dlYPred,loss] = modelPredictions(dlnet,mbq,classes)
```

```
% Initialize predictions.
numClasses = numel(classes);
outputCast = mbq.OutputCast{1};
dlYPred = dlarray(zeros(numClasses,0,outputCast),'CB');

% Reset mini-batch queue.
reset(mbq);

% For mini-batch queues with two ouputs, also compute the loss.
if mbq.NumOutputs == 1

    % Loop over mini-batches.
    while hasdata(mbq)

        % Make predictions.
        dlX = next(mbq);
        dlY = predict(dlnet,dlX);
        dlYPred = [dlYPred dlY];
    end

else
    % Initialize loss.
    numObservations = 0;
    loss = 0;

    % Loop over mini-batches.
    while hasdata(mbq)

        % Make predictions.
        [dlX,dlT] = next(mbq);
        dlY = predict(dlnet,dlX);
        dlYPred = [dlYPred dlY];

        % Calculate unnormalized loss.
        miniBatchSize = size(dlX,2);
        loss = loss + miniBatchSize * crossentropy(dlY, dlT);

        % Count observations.
        numObservations = numObservations + miniBatchSize;
    end

    % Normalize loss.
    loss = loss / numObservations;

    % Convert to double.
    loss = double(gather(extractdata(loss)));
end

end
```

See Also

dlarray | dlfeval | dlgradient | doc2sequence | lstmLayer | sequenceInputLayer | tokenizedDocument | wordEmbeddingLayer | wordcloud

Related Examples

- “Define Custom Training Loops, Loss Functions, and Networks” (Deep Learning Toolbox)
- “Classify Text Data Using Deep Learning” on page 2-65
- “Create Simple Text Model for Classification” on page 2-2
- “Analyze Text Data Using Topic Models” on page 2-13
- “Analyze Text Data Using Multiword Phrases” on page 2-7
- “Train a Sentiment Classifier” on page 2-51
- “Sequence Classification Using Deep Learning” (Deep Learning Toolbox)
- “Deep Learning in MATLAB” (Deep Learning Toolbox)

Multilabel Text Classification Using Deep Learning

This example shows how to classify text data that has multiple independent labels.

For classification tasks where there can be multiple independent labels for each observation—for example, tags on an scientific article—you can train a deep learning model to predict probabilities for each independent class. To enable a network to learn multilabel classification targets, you can optimize the loss of each class independently using binary cross-entropy loss.

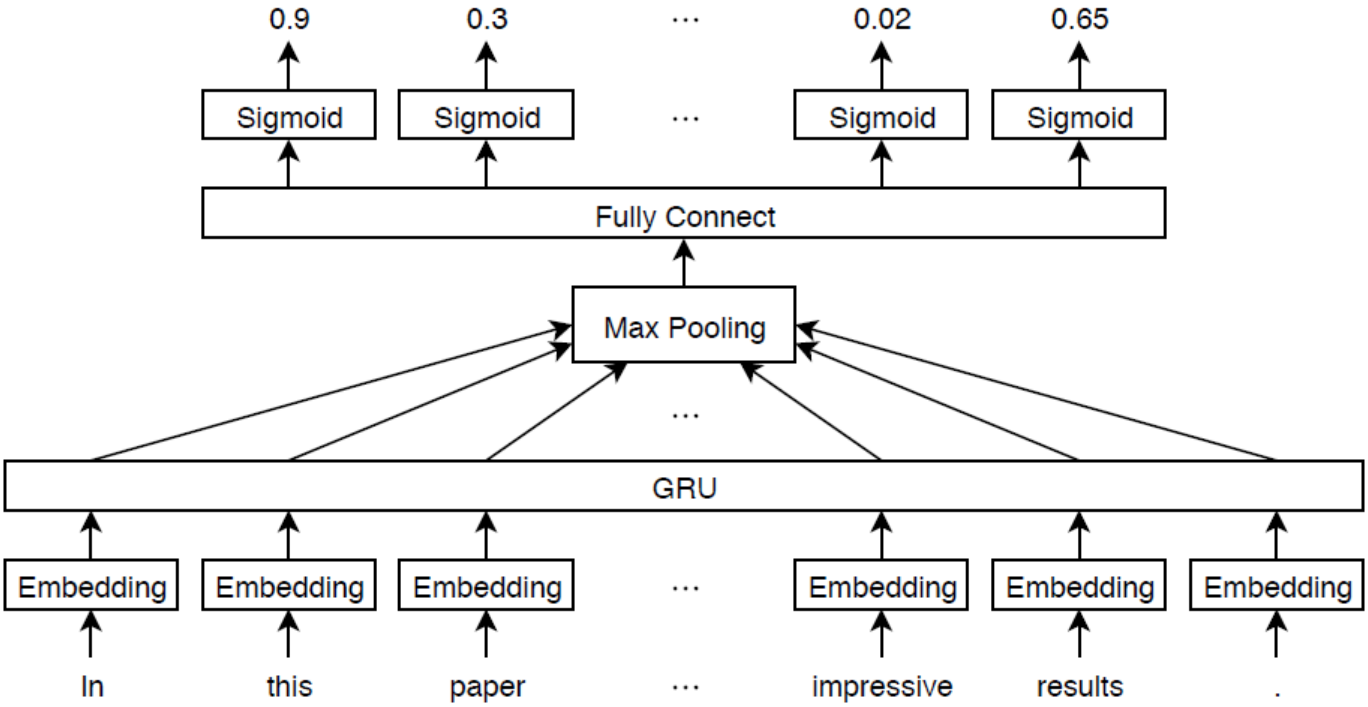
This example defines a deep learning model that classifies subject areas given the abstracts of mathematical papers collected using the arXiv API [1]. The model consists of a word embedding and GRU, max pooling operation, fully connected, and sigmoid operations.

To measure the performance of multilabel classification, you can use the labeling F-score [2]. The labeling F-score evaluates multilabel classification by focusing on per-text classification with partial matches. The measure is the normalized proportion of matching labels against the total number of true and predicted labels.

This example defines the following model:

- A word embedding that maps a sequence of words to a sequence of numeric vectors.
- A GRU operation that learns dependencies between the embedding vectors.
- A max pooling operation that reduces a sequence of feature vectors to a single feature vector.
- A fully connected layer that maps the features to the binary outputs.
- A sigmoid operation for learning the binary cross entropy loss between the outputs and the target labels.

This diagram shows a piece of text propagating through the model architecture and outputting a vector of probabilities. The probabilities are independent, so they need not sum to one.



Import Text Data

Import a set of abstracts and category labels from math papers using the arXiv API. Specify the number of records to import using the `importSize` variable. Note that the arXiv API is rate limited to querying 1000 articles at a time and requires waiting between requests.

```
importSize = 50000;
```

Import the first set of records.

```
url = "https://export.arxiv.org/oai2?verb=ListRecords" + ...
    "&set=math" + ...
    "&metadataPrefix=arXiv";
options = weboptions('Timeout',160);
code = webread(url,options);
```

Parse the returned XML content and create an array of `htmlTree` objects containing the record information.

```
tree = htmlTree(code);
subtrees = findElement(tree,"record");
numel(subtrees)
```

Iteratively import more chunks of records until the required amount is reached, or there are no more records. To continue importing records from where you left off, use the `resumptionToken` attribute from the previous result. To adhere to the rate limits imposed by the arXiv API, add a delay of 20 seconds before each query using the `pause` function.

```
while numel(subtrees) < importSize
    subtreeResumption = findElement(tree,"resumptionToken");
```

```
    if isempty(subtreeResumption)
        break
    end

    resumptionToken = extractHTMLText(subtreeResumption);

    url = "https://export.arxiv.org/oai2?verb=ListRecords" + ...
        "&resumptionToken=" + resumptionToken;

    pause(20)
    code = webread(url,options);

    tree = htmlTree(code);

    subtrees = [subtrees; findElement(tree,"record")];
end
```

Extract and Preprocess Text Data

Extract the abstracts and labels from the parsed HTML trees.

Find the `"<abstract>"` and `"<categories>"` elements using the `findElement` function.

```
subtreeAbstract = htmlTree("");
subtreeCategory = htmlTree("");

for i = 1:numel(subtrees)
    subtreeAbstract(i) = findElement(subtrees(i),"abstract");
    subtreeCategory(i) = findElement(subtrees(i),"categories");
end
```

Extract the text data from the subtrees containing the abstracts using the `extractHTMLText` function.

```
textData = extractHTMLText(subtreeAbstract);
```

Tokenize and preprocess the text data using the `preprocessText` function, listed at the end of the example.

```
documentsAll = preprocessText(textData);
documentsAll(1:5)
```

```
ans =
  5×1 tokenizedDocument:

    72 tokens: describe new algorithm $(k,\ell)$ pebble game color obtain characterization family
    22 tokens: show determinant stirling cycle number count unlabeled acyclic singlesource automa
    18 tokens: "paper" "show" "compute" "$\lambda_{\alpha}$" "norm" "$\alpha\ge 0$" "dyadic" "gri
    62 tokens: partial cube isometric subgraphs hypercubes structure graph define mean semicubes
    29 tokens: paper present algorithm compute hecke eigensystems hilbertsiegel cusp form real qu
```

Extract the labels from the subtrees containing the labels.

```
strLabels = extractHTMLText(subtreeCategory);
labelsAll = arrayfun(@split,strLabels,'UniformOutput',false);
```

Remove labels that do not belong to the `"math"` set.

```
for i = 1:numel(labelsAll)
    labelsAll{i} = labelsAll{i}(startsWith(labelsAll{i},"math."));
end
```

Visualize some of the classes in a word cloud. Find the documents corresponding to the following:

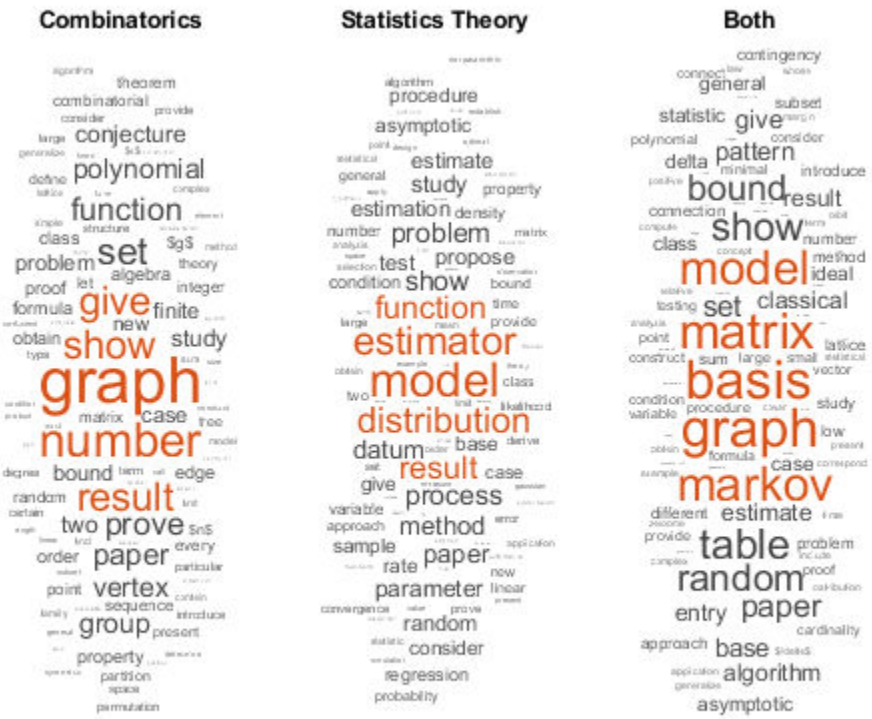
- Abstracts tagged with "Combinatorics" and not tagged with `"Statistics Theory"`
- Abstracts tagged with "Statistics Theory" and not tagged with `"Combinatorics"`
- Abstracts tagged with both `"Combinatorics"` and `"Statistics Theory"`

Find the document indices for each of the groups using the `ismember` function.

```
idxCO = cellfun(@(lbls) ismember("math.CO",lbls) && ~ismember("math.ST",lbls),labelsAll);
idxST = cellfun(@(lbls) ismember("math.ST",lbls) && ~ismember("math.CO",lbls),labelsAll);
idxCOST = cellfun(@(lbls) ismember("math.CO",lbls) && ismember("math.ST",lbls),labelsAll);
```

Visualize the documents for each group in a word cloud.

```
figure
subplot(1,3,1)
wordcloud(documentsAll(idxCO));
title("Combinatorics")

subplot(1,3,2)
wordcloud(documentsAll(idxST));
title("Statistics Theory")

subplot(1,3,3)
wordcloud(documentsAll(idxCOST));
title("Both")
```



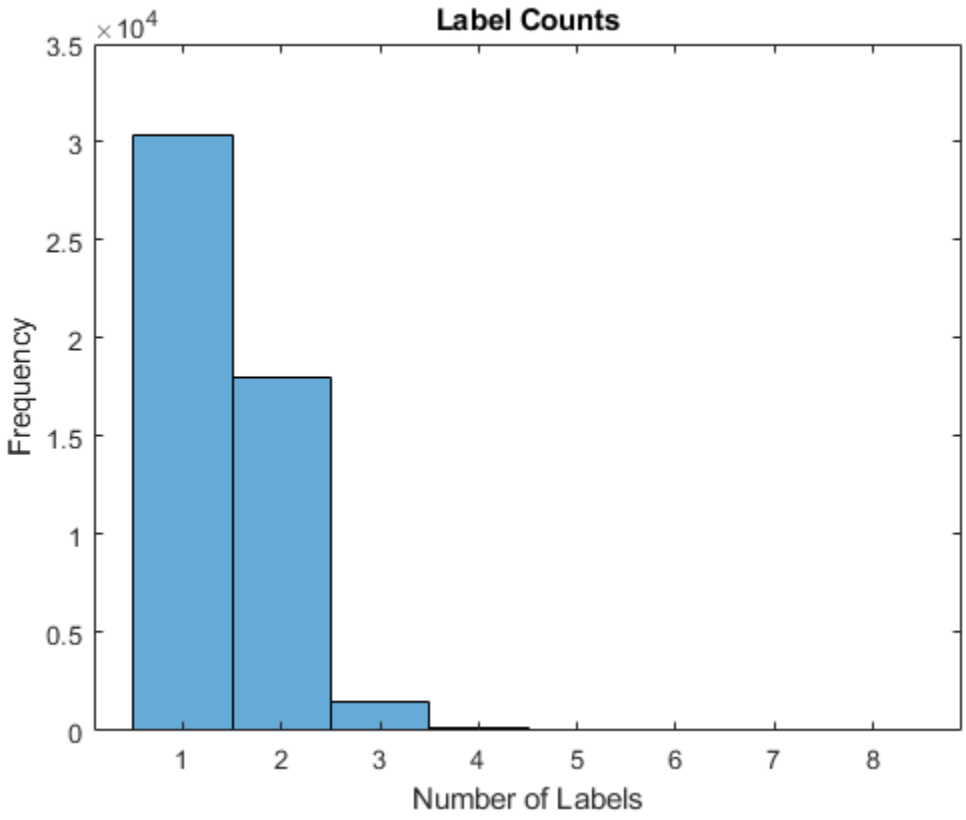
View the number of classes.

```
classNames = unique(cat(1,labelsAll{:}));
numClasses = numel(classNames)
```

```
numClasses = 32
```

Visualize the number of per-document labels using a histogram.

```
labelCounts = cellfun(@numel, labelsAll);
figure
histogram(labelCounts)
xlabel("Number of Labels")
ylabel("Frequency")
title("Label Counts")
```



Prepare Text Data for Deep Learning

Partition the data into training and validation partitions using the `cvpartition` function. Hold out 10% of the data for validation by setting the `'HoldOut'` option to 0.1.

```
cvp = cvpartition(numel(documentsAll),'HoldOut',0.1);
documentsTrain = documentsAll(training(cvp));
documentsValidation = documentsAll(test(cvp));

labelsTrain = labelsAll(training(cvp));
labelsValidation = labelsAll(test(cvp));
```

Create a word encoding object that encodes the training documents as sequences of word indices. Specify a vocabulary of the 5000 words by setting the `'Order'` option to `'frequency'`, and the `'MaxNumWords'` option to 5000.

```
enc = wordEncoding(documentsTrain,'Order','frequency','MaxNumWords',5000)
```

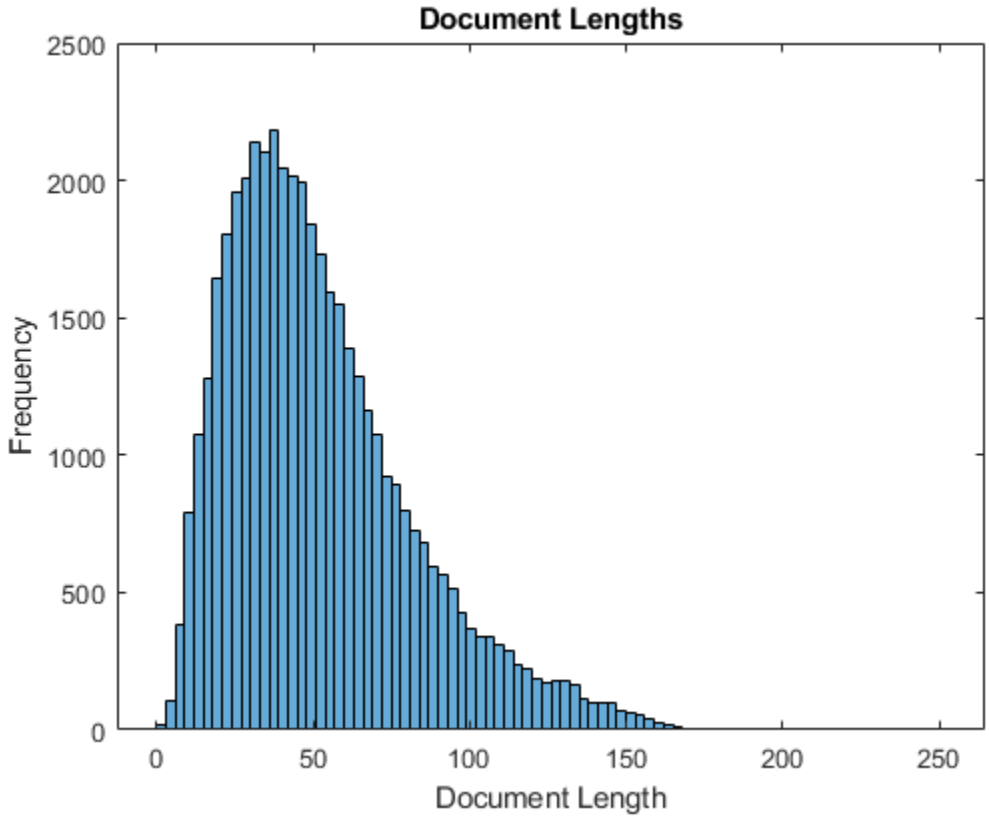
```
enc =
  wordEncoding with properties:

      NumWords: 5000
    Vocabulary: [1×5000 string]
```

To improve training, use the following techniques:

1. When training, truncate the documents to a length that reduces the amount of padding used and does not does discard too much data.
2. Train for one epoch with the documents sorted by length in ascending order, then shuffle the data each epoch. This technique is known as *sortagrad*.

To choose a sequence length for truncation, visualize the document lengths in a histogram and choose a value that captures most of the data.

```
documentLengths = doclength(documentsTrain);

figure
histogram(documentLengths)
xlabel("Document Length")
ylabel("Frequency")
title("Document Lengths")
```



Most of the training documents have fewer than 175 tokens. Use 175 tokens as the target length for truncation and padding.

```
maxSequenceLength = 175;
```

To use the sortagrad technique, sort the documents by length in ascending order.

```
[~,idx] = sort(documentLengths);
documentsTrain = documentsTrain(idx);
labelsTrain = labelsTrain(idx);
```

Define and Initialize Model Parameters

Define the parameters for each of the operations and include them in a struct. Use the format `parameters.OperationName.ParameterName`, where `parameters` is the struct, `OperationName` is the name of the operation (for example `"fc"`), and `ParameterName` is the name of the parameter (for example, `"Weights"`).

Create a struct `parameters` containing the model parameters. Initialize the bias with zeros. Use the following weight initializers for the operations:

- For the embedding, initialize the weights with random normal values.
- For the GRU operation, initialize the weights using the `initializeGlorot` function, listed at the end of the example.
- For the fully connect operation, initialize the weights using the `initializeGaussian` function, listed at the end of the example.

```
embeddingDimension = 300;
numHiddenUnits = 250;
inputSize = enc.NumWords + 1;

parameters = struct;
parameters.emb.Weights = dlarray(randn([embeddingDimension inputSize]));

parameters.gru.InputWeights = dlarray(initializeGlorot(3*numHiddenUnits,embeddingDimension));
parameters.gru.RecurrentWeights = dlarray(initializeGlorot(3*numHiddenUnits,numHiddenUnits));
parameters.gru.Bias = dlarray(zeros(3*numHiddenUnits,1,'single'));

parameters.fc.Weights = dlarray(initializeGaussian([numClasses,numHiddenUnits]));
parameters.fc.Bias = dlarray(zeros(numClasses,1,'single'));
```

View the `parameters` struct.

```
parameters
```

```
parameters = struct with fields:
    emb: [1×1 struct]
    gru: [1×1 struct]
     fc: [1×1 struct]
```

View the parameters for the GRU operation.

```
parameters.gru
```

```
ans = struct with fields:
        InputWeights: [750×300 dlarray]
    RecurrentWeights: [750×250 dlarray]
                Bias: [750×1 dlarray]
```

Define Model Function

Create the function `model`, listed at the end of the example, which computes the outputs of the deep learning model described earlier. The function `model` takes as input the input data `dlX` and the model parameters `parameters`. The network outputs the predictions for the labels.

Define Model Gradients Function

Create the function `modelGradients`, listed at the end of the example, which takes as input a mini-batch of input data `dlX` and the corresponding targets `T` containing the labels, and returns the gradients of the loss with respect to the learnable parameters, the corresponding loss, and the network outputs.

Specify Training Options

Train for 5 epochs with a mini-batch size of 256.

```
numEpochs = 5;
miniBatchSize = 256;
```

Train using the Adam optimizer, with a learning rate of 0.01, and specify gradient decay and squared gradient decay factors of 0.5 and 0.999, respectively.

```
learnRate = 0.01;
gradientDecayFactor = 0.5;
squaredGradientDecayFactor = 0.999;
```

Clip the gradients with a threshold of 1 using L_2 norm gradient clipping.

```
gradientThreshold = 1;
```

Visualize the training progress in a plot.

```
plots = "training-progress";
```

To convert a vector of probabilities to labels, use the labels with probabilities higher than a specified threshold. Specify a label threshold of 0.5.

```
labelThreshold = 0.5;
```

Validate the network every epoch.

```
numObservationsTrain = numel(documentsTrain);
numIterationsPerEpoch = floor(numObservationsTrain/miniBatchSize);
validationFrequency = numIterationsPerEpoch;
```

Train on a GPU if one is available. This requires Parallel Computing Toolbox™. Using a GPU requires Parallel Computing Toolbox™ and a CUDA® enabled NVIDIA® GPU with compute capability 3.0 or higher.

```
executionEnvironment = "auto";
```

Train Model

Train the model using a custom training loop.

For each epoch, loop over mini-batches of data. At the end of each epoch, shuffle the data. At the end of each iteration, update the training progress plot.

For each mini-batch:

- Convert the documents to sequences of word indices and convert the labels to dummy variables.
- Convert the sequences to `dlarray` objects with underlying type single and specify the dimension labels `'BCT'` (batch, channel, time).

- For GPU training, convert to `gpuArray` objects.
- Evaluate the model gradients and loss using `dlfeval` and the `modelGradients` function.
- Clip the gradients.
- Update the network parameters using the `adamupdate` function.
- If necessary, validate the network using the `modelPredictions` function, listed at the end of the example.
- Update the training plot.

Initialize the training progress plot.

```
if plots == "training-progress"
    figure

    % Labeling F-Score.
    subplot(2,1,1)
    lineFScoreTrain = animatedline('Color',[0 0.447 0.741]);
    lineFScoreValidation = animatedline( ...
        'LineStyle','--', ...
        'Marker','o', ...
        'MarkerFaceColor','black');
    ylim([0 1])
    xlabel("Iteration")
    ylabel("Labeling F-Score")
    grid on

    % Loss.
    subplot(2,1,2)
    lineLossTrain = animatedline('Color',[0.85 0.325 0.098]);
    lineLossValidation = animatedline( ...
        'LineStyle','--', ...
        'Marker','o', ...
        'MarkerFaceColor','black');
    ylim([0 inf])
    xlabel("Iteration")
    ylabel("Loss")
    grid on
end
```

Initialize parameters for the Adam optimizer.

```
trailingAvg = [];
trailingAvgSq = [];
```

Prepare the validation data. Create a one-hot encoded matrix where non-zero entries correspond to the labels of each observation.

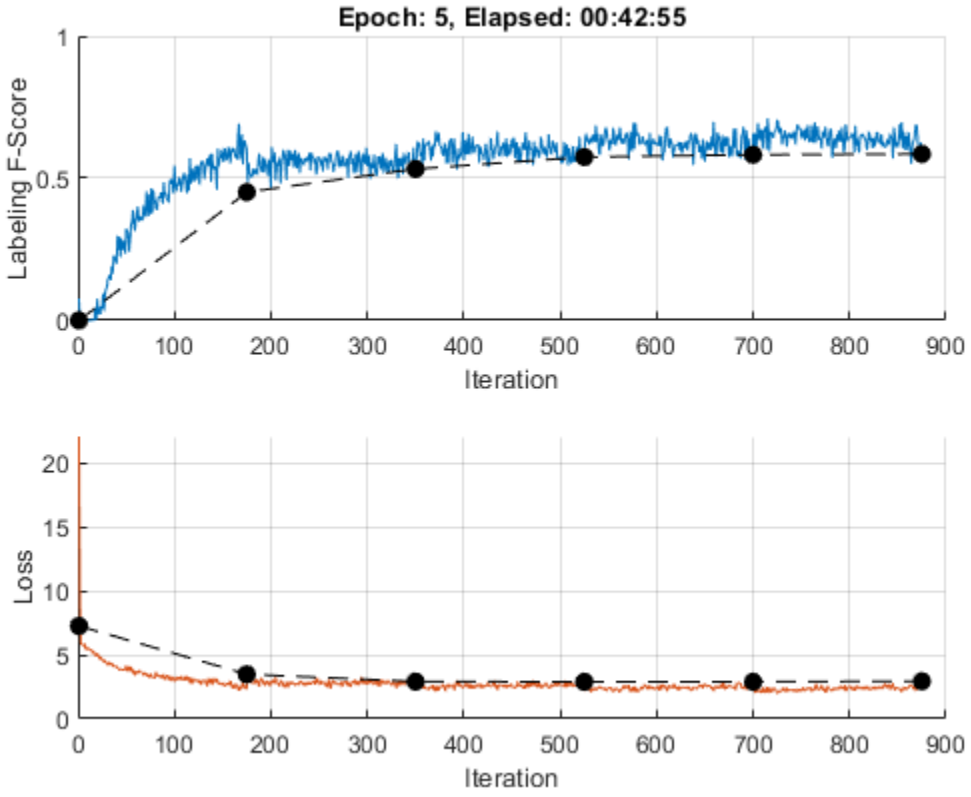
```
numObservationsValidation = numel(documentsValidation);
TValidation = zeros(numClasses, numObservationsValidation, 'single');
for i = 1:numObservationsValidation
    [~,idx] = ismember(labelsValidation{i},classNames);
    TValidation(idx,i) = 1;
end
```

Train the model.

```
iteration = 0;
start = tic;
```

```
% Loop over epochs.
for epoch = 1:numEpochs

    % Loop over mini-batches.
    for i = 1:numIterationsPerEpoch
        iteration = iteration + 1;
        idx = (i-1)*miniBatchSize+1:i*miniBatchSize;

        % Read mini-batch of data and convert the labels to dummy
        % variables.
        documents = documentsTrain(idx);
        labels = labelsTrain(idx);

        % Convert documents to sequences.
        len = min(maxSequenceLength,max(doclength(documents)));
        X = doc2sequence(enc,documents, ...
            'PaddingValue',inputSize, ...
            'Length',len);
        X = cat(1,X{:});

        % Dummify labels.
        T = zeros(numClasses, miniBatchSize, 'single');
        for j = 1:miniBatchSize
            [~,idx2] = ismember(labels{j},classNames);
            T(idx2,j) = 1;
        end

        % Convert mini-batch of data to dlarray.
        dlX = dlarray(X,'BTC');

        % If training on a GPU, then convert data to gpuArray.
        if (executionEnvironment == "auto" && canUseGPU) || executionEnvironment == "gpu"
            dlX = gpuArray(dlX);
        end

        % Evaluate the model gradients, state, and loss using dlfeval and the
        % modelGradients function.
        [gradients,loss,dlYPred] = dlfeval(@modelGradients, dlX, T, parameters);

        % Gradient clipping.
        gradients = dlupdate(@(g) thresholdL2Norm(g, gradientThreshold),gradients);

        % Update the network parameters using the Adam optimizer.
        [parameters,trailingAvg,trailingAvgSq] = adamupdate(parameters,gradients, ...
            trailingAvg,trailingAvgSq,iteration,learnRate,gradientDecayFactor,squaredGradientDeca

        % Display the training progress.
        if plots == "training-progress"
            subplot(2,1,1)
            D = duration(0,0,toc(start),'Format','hh:mm:ss');
            title("Epoch: " + epoch + ", Elapsed: " + string(D))

            % Loss.
            addpoints(lineLossTrain,iteration,double(gather(extractdata(loss))))

            % Labeling F-score.
            YPred = extractdata(dlYPred) > labelThreshold;
```

```
            score = labelingFScore(YPred,T);
            addpoints(lineFScoreTrain,iteration,double(gather(score)))

            drawnow

            % Display validation metrics.
            if iteration == 1 || mod(iteration,validationFrequency) == 0
                dlYPredValidation = modelPredictions(parameters,enc,documentsValidation,miniBatch

                % Loss.
                lossValidation = crossentropy(dlYPredValidation,TValidation, ...
                    'TargetCategories','independent', ...
                    'DataFormat','CB');
                addpoints(lineLossValidation,iteration,double(gather(extractdata(lossValidation)

                % Labeling F-score.
                YPredValidation = extractdata(dlYPredValidation) > labelThreshold;
                score = labelingFScore(YPredValidation,TValidation);
                addpoints(lineFScoreValidation,iteration,double(gather(score)))

                drawnow
            end
        end
    end

    % Shuffle data.
    idx = randperm(numObservationsTrain);
    documentsTrain = documentsTrain(idx);
    labelsTrain = labelsTrain(idx);
end
```



Test Model

To make predictions on a new set of data, use the `modelPredictions` function, listed at the end of the example. The `modelPredictions` function takes as input the model parameters, a word encoding, and an array of tokenized documents, and outputs the model predictions corresponding to the specified mini-batch size and the maximum sequence length.

```
dlYPredValidation = modelPredictions(parameters,enc,documentsValidation,miniBatchSize,maxSequence
```

To convert the network outputs to an array of labels, find the labels with scores higher than the specified label threshold.

```
YPredValidation = extractdata(dlYPredValidation) > labelThreshold;
```

To evaluate the performance, calculate the labeling F-score using the `labelingFScore` function, listed at the end of the example. The labeling F-score evaluates multilabel classification by focusing on per-text classification with partial matches.

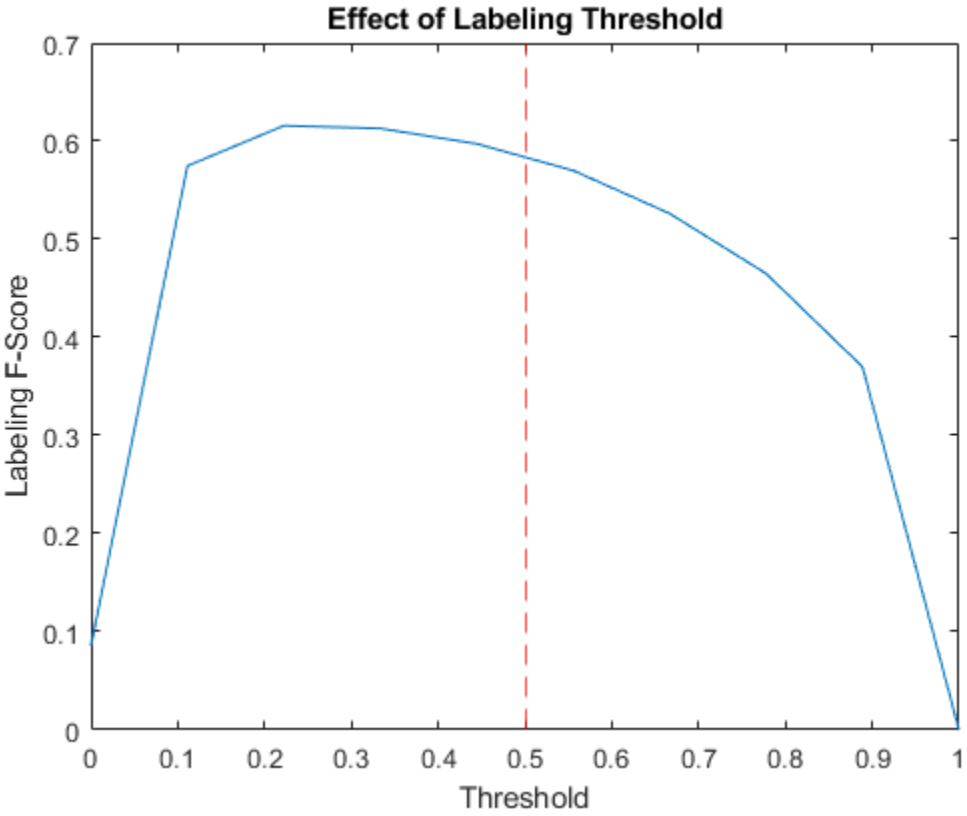
```
score = labelingFScore(YPredValidation,TValidation)
```

```
score = single
    0.5852
```

View the effect of the labeling threshold on the labeling F-score by trying a range of values for the threshold and comparing the results.

```
thr = linspace(0,1,10);
score = zeros(size(thr));
```

```
for i = 1:numel(thr)
    YPredValidationThr = extractdata(dlYPredValidation) >= thr(i);
    score(i) = labelingFScore(YPredValidationThr,TValidation);
end

figure
plot(thr,score)
xline(labelThreshold,'r--');
xlabel("Threshold")
ylabel("Labeling F-Score")
title("Effect of Labeling Threshold")
```



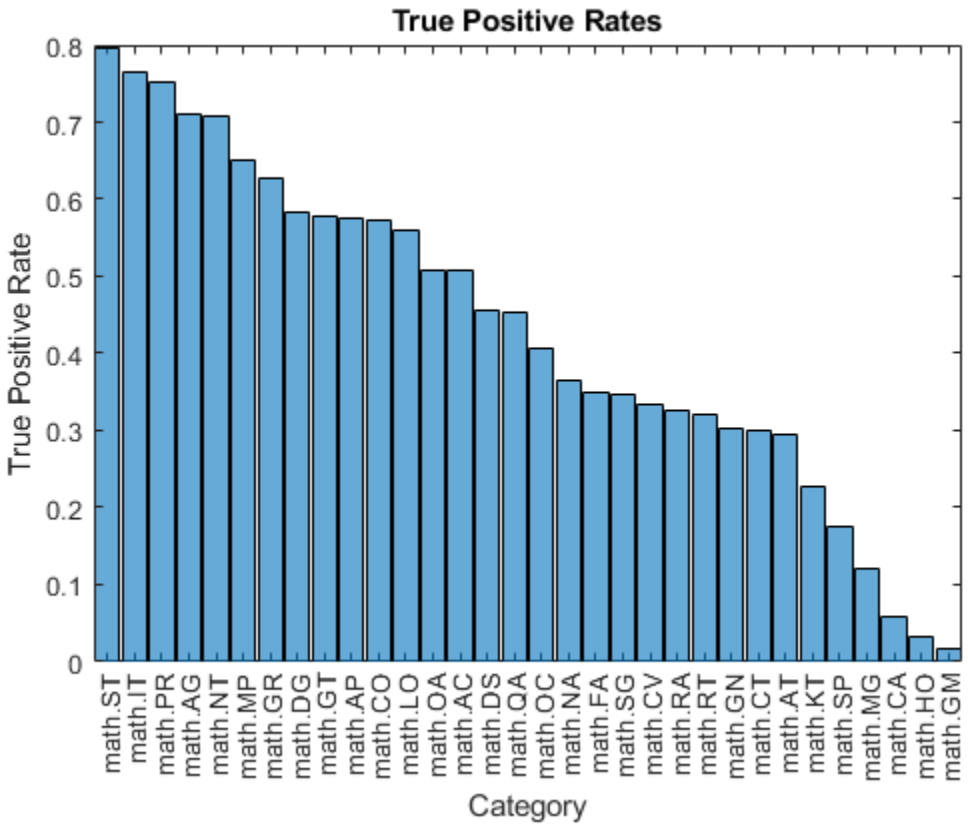
Visualize Predictions

To visualize the correct predictions of the classifier, calculate the numbers of true positives. A true positive is an instance of a classifier correctly predicting a particular class for an observation.

```
Y = YPredValidation;
T = TValidation;

numTruePositives = sum(T & Y,2);

numObservationsPerClass = sum(T,2);
truePositiveRates = numTruePositives ./ numObservationsPerClass;
```

Visualize the numbers of true positives for each class in a histogram.

```
figure
[~,idx] = sort(truePositiveRates,'descend');
histogram('Categories',classNames(idx),'BinCounts',truePositiveRates(idx))
xlabel("Category")
ylabel("True Positive Rate")
title("True Positive Rates")
```



Visualize the instances where the classifier predicts incorrectly by showing the distribution of true positives, false positives, and false negatives. A false positive is an instance of a classifier assigning a particular incorrect class to an observation. A false negative is an instance of a classifier failing to assign a particular correct class to an observation.

Create a confusion matrix showing the true positive, false positive, and false negative counts:

- For each class, display the true positive counts on the diagonal.
- For each pair of classes (i,j), display the number of instances of a false positive for j when the instance is also a false negative for i.

That is, the confusion matrix with elements given by:

$$\mathrm{TPFN}_{ij} = \begin{cases} \text{numTruePositives}(i), & \text{if } i = j \\ \text{numFalsePositives}(j \,|\, i \text{ is a false negative}), & \text{if } i \neq j \end{cases}$$

Calculate the false negatives and false positives.

```
falseNegatives = T & ~Y;
falsePositives = ~T & Y;
```

Calculate the off-diagonal elements.

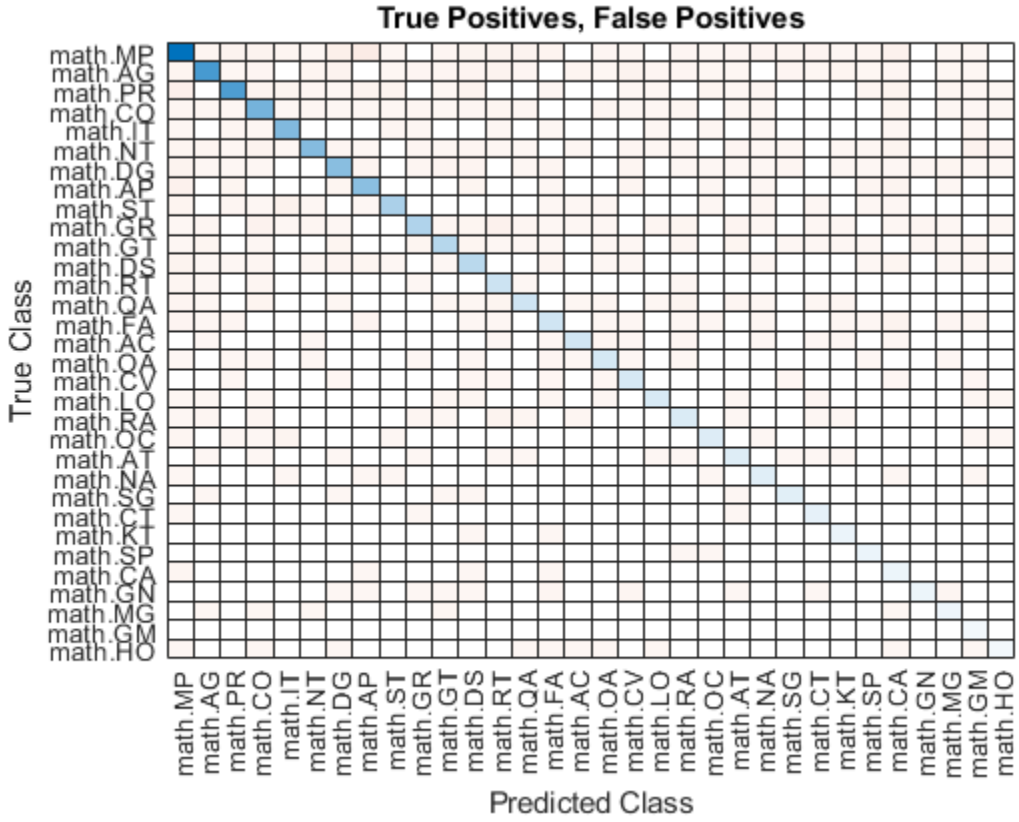
```
falseNegatives = permute(falseNegatives,[3 2 1]);
numConditionalFalsePositives = sum(falseNegatives & falsePositives, 2);
numConditionalFalsePositives = squeeze(numConditionalFalsePositives);

tpfnMatrix = numConditionalFalsePositives;
```

Set the diagonal elements to the true positive counts.

```
idxDiagonal = 1:numClasses+1:numClasses^2;
tpfnMatrix(idxDiagonal) = numTruePositives;
```

Visualize the true positive and false positive counts in a confusion matrix using the `confusionchart` function and sort the matrix such that the elements on the diagonal are in descending order.

```
figure
cm = confusionchart(tpfnMatrix,classNames);
sortClasses(cm,"descending-diagonal");
title("True Positives, False Positives")
```



To view the matrix in more detail, open this example as a live script and open the figure in a new window.

Preprocess Text Function

The `preprocessText` function tokenizes and preprocesses the input text data using the following steps:

1. Tokenize the text using the `tokenizedDocument` function. Extract mathematical equations as a single token using the `'RegularExpressions'` option by specifying the regular expression `"\$.*?\$"`, which captures text appearing between two "$" symbols.
2. Erase the punctuation using the `erasePunctuation` function.
3. Convert the text to lowercase using the `lower` function.
4. Remove the stop words using the `removeStopWords` function.
5. Lemmatize the text using the `normalizeWords` function with the `'Style'` option set to `'lemma'`.

```
function documents = preprocessText(textData)

% Tokenize the text.
regularExpressions = table;
regularExpressions.Pattern = "\$.*?\$";
regularExpressions.Type = "equation";

documents = tokenizedDocument(textData,'RegularExpressions',regularExpressions);

% Erase punctuation.
documents = erasePunctuation(documents);

% Convert to lowercase.
documents = lower(documents);

% Lemmatize.
documents = addPartOfSpeechDetails(documents);
documents = normalizeWords(documents,'Style','Lemma');

% Remove stop words.
documents = removeStopWords(documents);

% Remove short words.
documents = removeShortWords(documents,2);

end
```

Model Function

The function `model` takes as input the input data `dlX` and the model parameters `parameters`, and returns the predictions for the labels.

```
function dlY = model(dlX,parameters)

% Embedding
weights = parameters.emb.Weights;
dlX = embedding(dlX, weights);

% GRU
inputWeights = parameters.gru.InputWeights;
recurrentWeights = parameters.gru.RecurrentWeights;
bias = parameters.gru.Bias;

numHiddenUnits = size(inputWeights,1)/3;
hiddenState = dlarray(zeros([numHiddenUnits 1]));

dlY = gru(dlX, hiddenState, inputWeights, recurrentWeights, bias,'DataFormat','CBT');
```

```
% Max pooling along time dimension
dlY = max(dlY,[],3);

% Fully connect
weights = parameters.fc.Weights;
bias = parameters.fc.Bias;
dlY = fullyconnect(dlY,weights,bias,'DataFormat','CB');

% Sigmoid
dlY = sigmoid(dlY);

end
```

Model Gradients Function

The `modelGradients` function takes as input a mini-batch of input data `dlX` with corresponding targets `T` containing the labels and returns the gradients of the loss with respect to the learnable parameters, the corresponding loss, and the network outputs.

```
function [gradients,loss,dlYPred] = modelGradients(dlX,T,parameters)

dlYPred = model(dlX,parameters);

loss = crossentropy(dlYPred,T,'TargetCategories','independent','DataFormat','CB');

gradients = dlgradient(loss,parameters);

end
```

Model Predictions Function

The `modelPredictions` function takes as input the model parameters, a word encoding, an array of tokenized documents, a mini-batch size, and a maximum sequence length, and returns the model predictions by iterating over mini-batches of the specified size.

```
function dlYPred = modelPredictions(parameters,enc,documents,miniBatchSize,maxSequenceLength)

inputSize = enc.NumWords + 1;

numObservations = numel(documents);
numIterations = ceil(numObservations / miniBatchSize);

numFeatures = size(parameters.fc.Weights,1);
dlYPred = zeros(numFeatures,numObservations,'like',parameters.fc.Weights);

for i = 1:numIterations

    idx = (i-1)*miniBatchSize+1:min(i*miniBatchSize,numObservations);

    len = min(maxSequenceLength,max(doclength(documents(idx))));
    X = doc2sequence(enc,documents(idx), ...
        'PaddingValue',inputSize, ...
        'Length',len);
    X = cat(1,X{:});

    dlX = dlarray(X,'BTC');
```

```
    dlYPred(:,idx) = model(dlX,parameters);
end

end
```

Labeling F-Score Function

The labeling F-score function [2] evaluates multilabel classification by focusing on per-text classification with partial matches. The measure is the normalized proportion of matching labels against the total number of true and predicted labels given by

$$\frac{1}{N}\sum_{n=1}^{N}\left(\frac{2\sum_{c=1}^{C}Y_{nc}T_{nc}}{\sum_{c=1}^{C}(Y_{nc}+T_{nc})}\right),$$

where N and C correspond to the number of observations and classes, respectively, and Y and T correspond to the predictions and targets, respectively.

```
function score = labelingFScore(Y,T)

numObservations = size(T,2);

scores = (2 * sum(Y .* T)) ./ sum(Y + T);
score = sum(scores) / numObservations;

end
```

Glorot Weights Initialization Function

The `initializeGlorot` function generates an array of weights according to Glorot initialization.

```
function weights = initializeGlorot(numOut, numIn)

varWeights = sqrt( 6 / (numIn + numOut) );
weights = varWeights * (2 * rand([numOut, numIn], 'single') - 1);

end
```

Gaussian Weights Initialization Function

The `initializeGaussian` function samples weights from a Gaussian distribution with mean 0 and standard deviation 0.01.

```
function parameter = initializeGaussian(sz)

parameter = randn(sz,'single') .* 0.01;

end
```

Embedding Function

The `embedding` function maps numeric indices to the corresponding vector given by the input weights.

```
function Z = embedding(X, weights)
% Reshape inputs into a vector.
[N, T] = size(X, 2:3);
X = reshape(X, N*T, 1);
```

```
% Index into embedding matrix.
Z = weights(:, X);

% Reshape outputs by separating batch and sequence dimensions.
Z = reshape(Z, [], N, T);
end
```

L_2 Norm Gradient Clipping Function

The `thresholdL2Norm` function scales the input gradients so that their L_2 norm values equal the specified gradient threshold when the L_2 norm value of the gradient of a learnable parameter is larger than the specified threshold.

```
function gradients = thresholdL2Norm(gradients,gradientThreshold)

gradientNorm = sqrt(sum(gradients(:).^2));
if gradientNorm > gradientThreshold
    gradients = gradients * (gradientThreshold / gradientNorm);
end

end
```

References

1. arXiv. "arXiv API." Accessed January 15, 2020. https://arxiv.org/help/api
2. Sokolova, Marina, and Guy Lapalme. "A Sytematic Analysis of Performance Measures for Classification Tasks." *Information Processing & Management* 45, no. 4 (2009): 427–437.

See Also

`adamupdate` | `dlarray` | `dlfeval` | `dlgradient` | `dlupdate` | `doc2sequence` | `fullyconnect` | `tokenizedDocument` | `wordEncoding`

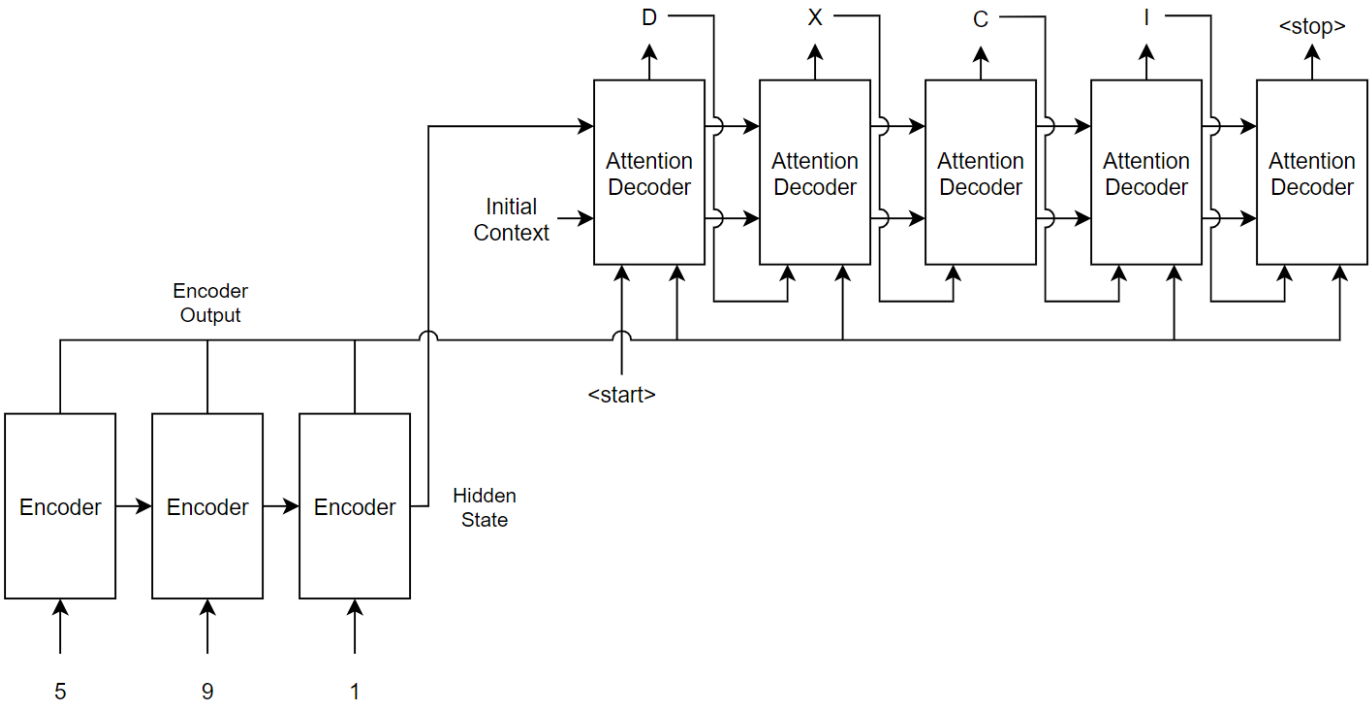
Related Examples

- “Classify Text Data Using Deep Learning” on page 2-65
- “Create Simple Text Model for Classification” on page 2-2
- “Deep Learning in MATLAB” (Deep Learning Toolbox)

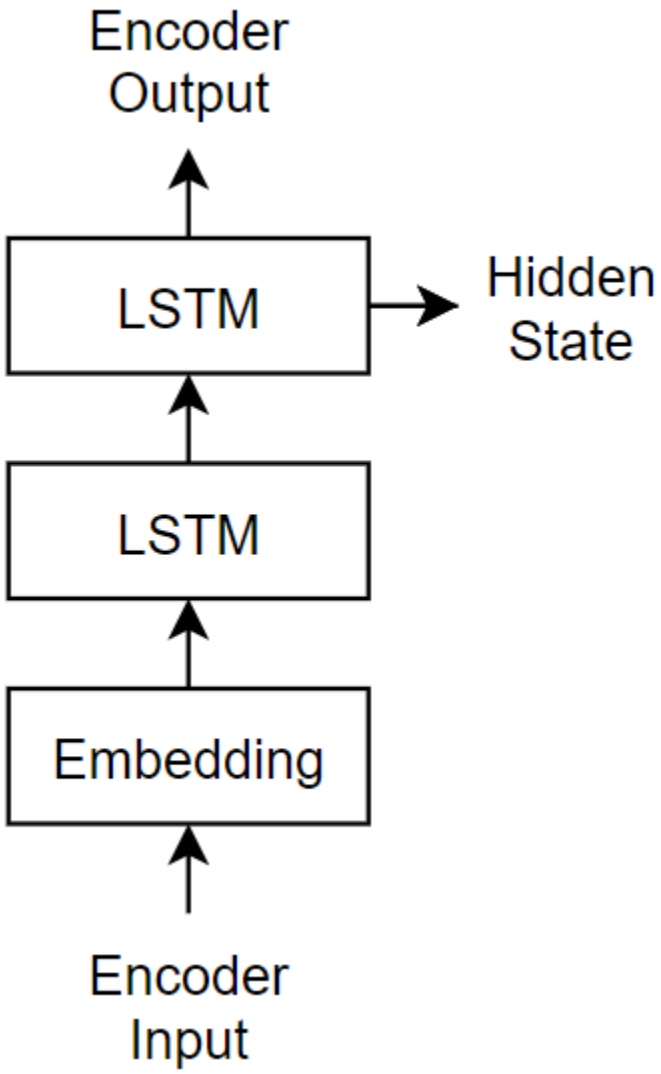
Sequence-to-Sequence Translation Using Attention

This example shows how to convert decimal strings to Roman numerals using a recurrent sequence-to-sequence encoder-decoder model with attention.

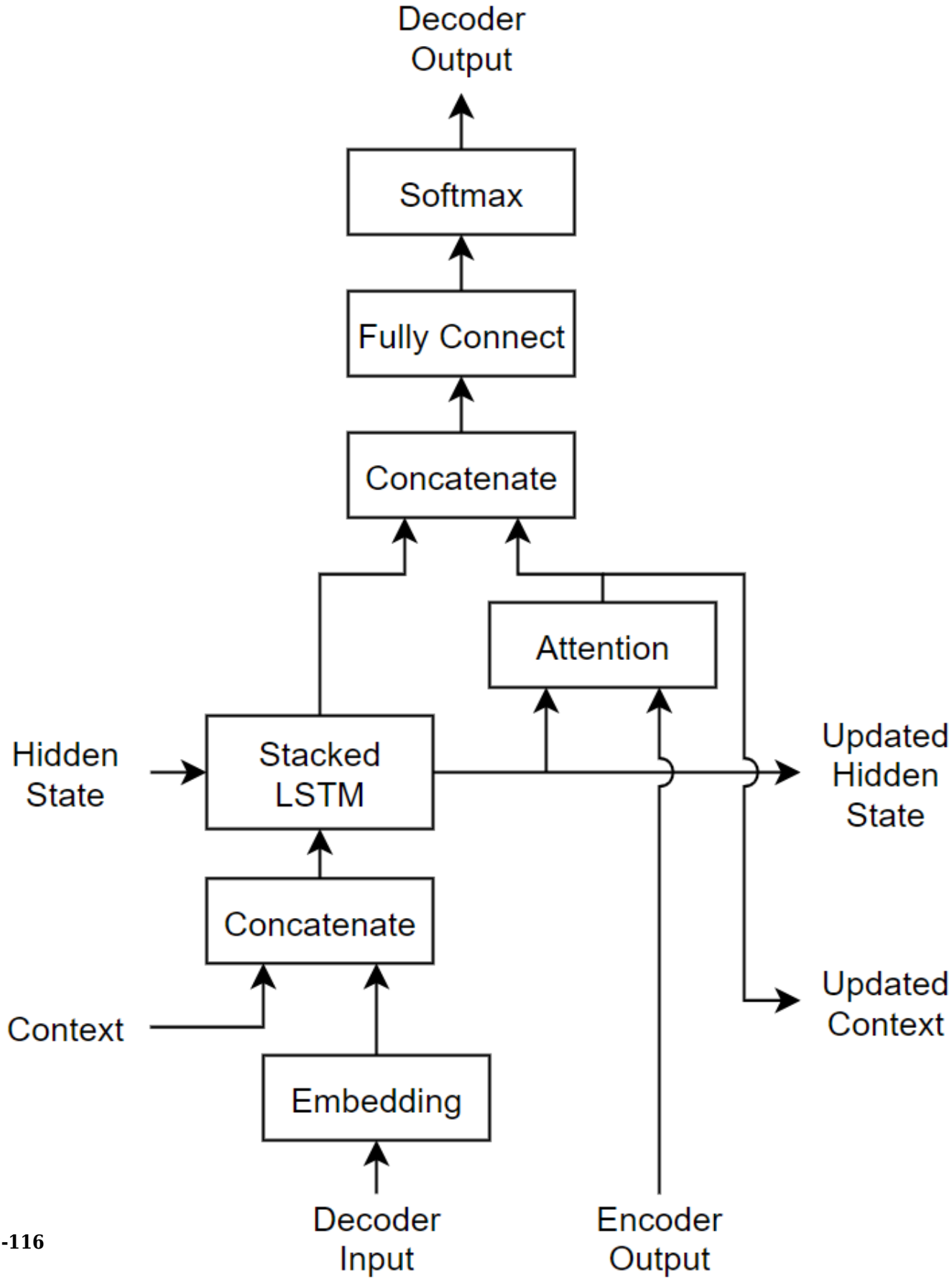
Recurrent encoder-decoder models have proven successful at tasks like abstractive text summarization and neural machine translation. The models consistent of an *encoder* which typically processes input data with a recurrent layer such as LSTM, and a *decoder* which maps the encoded input into the desired output, typically with a second recurrent layer. Models that incorporate *attention mechanisms* into the models allows the decoder to focus on parts of the encoded input while generating the translation.



For the encoder model, this example uses a simple network consisting of an embedding followed by two LSTM operations. Embedding is a method of converting categorical tokens into numeric vectors.



For the decoder model, this example uses a network very similar to the encoder that contains two LSTMs. However, an important difference is that the decoder contains an attention mechanism. The attention mechanism allows the decoder to *attend* to specific parts of the encoder output.


Decoder Output
Softmax
Fully Connect
Concatenate
Attention
Hidden State
Stacked LSTM
Updated Hidden State
Concatenate
Context
Updated Context
Embedding
Decoder Input
Encoder Output

Load Training Data

Download the decimal-Roman numeral pairs from "romanNumerals.csv"

```
filename = fullfile("romanNumerals.csv");

options = detectImportOptions(filename, ...
    'TextType','string', ...
    'ReadVariableNames',false);
options.VariableNames = ["Source" "Target"];
options.VariableTypes = ["string" "string"];

data = readtable(filename,options);
```

Split the data into training and test partitions containing 50% of the data each.

```
idx = randperm(size(data,1),500);
dataTrain = data(idx,:);
dataTest = data;
dataTest(idx,:) = [];
```

View some of the decimal-Roman numeral pairs.

```
head(dataTrain)
```

```
ans=8×2 table
    Source      Target
    ______    __________

    "228"     "CCXXVIII"
    "267"     "CCLXVII"
    "294"     "CCXCIV"
    "179"     "CLXXIX"
    "396"     "CCCXCVI"
    "2"       "II"
    "4"       "IV"
    "270"     "CCLXX"
```

Preprocess Data

Preprocess the text data using the `transformText` function, listed at the end of the example. The `transformText` function preprocesses and tokenizes the input text for translation by splitting the text into characters and adding start and stop tokens. To translate text by splitting the text into words instead of characters, skip the first step.

```
startToken = "<start>";
stopToken = "<stop>";

strSource = dataTrain{:,1};
documentsSource = transformText(strSource,startToken,stopToken);
```

Create a `wordEncoding` object that maps tokens to a numeric index and vice-versa using a vocabulary.

```
encSource = wordEncoding(documentsSource);
```

Using the word encoding, convert the source text data to numeric sequences.

```
sequencesSource = doc2sequence(encSource, documentsSource,'PaddingDirection','none');
```

Convert the target data to sequences using the same steps.

```
strTarget = dataTrain{:,2};
documentsTarget = transformText(strTarget,startToken,stopToken);
encTarget = wordEncoding(documentsTarget);
sequencesTarget = doc2sequence(encTarget, documentsTarget,'PaddingDirection','none');
```

Initialize Model Parameters

Initialize the model parameters. for both the encoder and decoder, specify an embedding dimension of 128, two LSTM layers with 200 hidden units, and dropout layers with random dropout with probability 0.05.

```
embeddingDimension = 128;
numHiddenUnits = 200;
dropout = 0.05;
```

Initialize Encoder Model Parameters

Initialize the weights of the encoding embedding using the Gaussian using the `initializeGaussian` function which is attached to this example as a supporting file. Specify a mean of 0 and a standard deviation of 0.01. To learn more, see “Gaussian Initialization” (Deep Learning Toolbox).

```
inputSize = encSource.NumWords + 1;
sz = [embeddingDimension inputSize];
mu = 0;
sigma = 0.01;
parameters.encoder.emb.Weights = initializeGaussian(sz,mu,sigma);
```

Initialize the learnable parameters for the encoder LSTM operations:

- Initialize the input weights with the Glorot initializer using the `initializeGlorot` function which is attached to this example as a supporting file. To learn more, see “Glorot Initialization” (Deep Learning Toolbox).
- Initialize the recurrent weights with the orthogonal initializer using the `initializeOrthogonal` function which is attached to this example as a supporting file. To learn more, see “Orthogonal Initialization” (Deep Learning Toolbox).
- Initialize the bias with the unit forget gate initializer using the `initializeUnitForgetGate` function which is attached to this example as a supporting file. To learn more, see “Unit Forget Gate Initialization” (Deep Learning Toolbox).

Initialize the learnable parameters for the first encoder LSTM operation.

```
sz = [4*numHiddenUnits embeddingDimension];
numOut = 4*numHiddenUnits;
numIn = embeddingDimension;

parameters.encoder.lstm1.InputWeights = initializeGlorot(sz,numOut,numIn);
parameters.encoder.lstm1.RecurrentWeights = initializeOrthogonal([4*numHiddenUnits numHiddenUnits
parameters.encoder.lstm1.Bias = initializeUnitForgetGate(numHiddenUnits);
```

Initialize the learnable parameters for the second encoder LSTM operation.

```
sz = [4*numHiddenUnits numHiddenUnits];
numOut = 4*numHiddenUnits;
```

```
numIn = numHiddenUnits;

parameters.encoder.lstm2.InputWeights = initializeGlorot(sz,numOut,numIn);
parameters.encoder.lstm2.RecurrentWeights = initializeOrthogonal([4*numHiddenUnits numHiddenUnits
parameters.encoder.lstm2.Bias = initializeUnitForgetGate(numHiddenUnits);
```

Initialize Decoder Model Parameters

Initialize the weights of the encoding embedding using the Gaussian using the `initializeGaussian` function. Specify a mean of 0 and a standard deviation of 0.01.

```
outputSize = encTarget.NumWords + 1;
sz = [embeddingDimension outputSize];
mu = 0;
sigma = 0.01;
parameters.decoder.emb.Weights = initializeGaussian(sz,mu,sigma);
```

Initialize the weights of the attention mechanism using the Glorot initializer using the `initializeGlorot` function.

```
sz = [numHiddenUnits numHiddenUnits];
numOut = numHiddenUnits;
numIn = numHiddenUnits;
parameters.decoder.attn.Weights = initializeGlorot(sz,numOut,numIn);
```

Initialize the learnable parameters for the decoder LSTM operations:

- Initialize the input weights with the Glorot initializer using the `initializeGlorot` function.
- Initialize the recurrent weights with the orthogonal initializer using the `initializeOrthogonal` function.
- Initialize the bias with the unit forget gate initializer using the `initializeUnitForgetGate` function.

Initialize the learnable parameters for the first decoder LSTM operation.

```
sz = [4*numHiddenUnits embeddingDimension+numHiddenUnits];
numOut = 4*numHiddenUnits;
numIn = embeddingDimension + numHiddenUnits;

parameters.decoder.lstm1.InputWeights = initializeGlorot(sz,numOut,numIn);
parameters.decoder.lstm1.RecurrentWeights = initializeOrthogonal([4*numHiddenUnits numHiddenUnits
parameters.decoder.lstm1.Bias = initializeUnitForgetGate(numHiddenUnits);
```

Initialize the learnable parameters for the second decoder LSTM operation.

```
sz = [4*numHiddenUnits numHiddenUnits];
numOut = 4*numHiddenUnits;
numIn = numHiddenUnits;

parameters.decoder.lstm2.InputWeights = initializeGlorot(sz,numOut,numIn);
parameters.decoder.lstm2.RecurrentWeights = initializeOrthogonal([4*numHiddenUnits numHiddenUnits
parameters.decoder.lstm2.Bias = initializeUnitForgetGate(numHiddenUnits);
```

Initialize the learnable parameters for the decoder fully connected operation:

- Initialize the weights with the Glorot initializer.

- Initialize the bias with zeros using the `initializeZeros` function which is attached to this example as a supporting file. To learn more, see “Zeros Initialization” (Deep Learning Toolbox).

```
sz = [outputSize 2*numHiddenUnits];
numOut = outputSize;
numIn = 2*numHiddenUnits;

parameters.decoder.fc.Weights = initializeGlorot(sz,numOut,numIn);
parameters.decoder.fc.Bias = initializeZeros([outputSize 1]);
```

Define Model Functions

Create the functions `modelEncoder` and `modelDecoder`, listed at the end of the example, that compute the outputs of the encoder and decoder models, respectively.

The `modelEncoder` function, listed in the Encoder Model Function on page 2-0 section of the example, takes the input data, the model parameters, the optional mask that is used to determine the correct outputs for training and returns the model outputs and the LSTM hidden state.

The `modelDecoder` function, listed in the Decoder Model Function on page 2-0 section of the example, takes the input data, the model parameters, the context vector, the LSTM initial hidden state, the outputs of the encoder, and the dropout probability and outputs the decoder output, the updated context vector, the updated LSTM state, and the attention scores.

Define Model Gradients Function

Create the function `modelGradients`, listed in the Model Gradients Function on page 2-0 section of the example, that takes the encoder and decoder model parameters, a mini-batch of input data and the padding masks corresponding to the input data, and the dropout probability and returns the gradients of the loss with respect to the learnable parameters in the models and the corresponding loss.

Specify Training Options

Train with a mini-batch size of 32 for 75 epochs with a learning rate of 0.002.

```
miniBatchSize = 32;
numEpochs = 75;
learnRate = 0.002;
```

Initialize the options from Adam.

```
gradientDecayFactor = 0.9;
squaredGradientDecayFactor = 0.999;
```

Train Model

Train the model using a custom training loop.

Train with the sequences sorted by increasing sequence length. This results in batches with sequences of approximately the same sequence length and ensures smaller sequence batches are used to update the model before longer sequence batches.

Sort the sequences by length.

```
sequenceLengths = cellfun(@(sequence) size(sequence,2), sequencesSource);
[~,idx] = sort(sequenceLengths);
```

```
sequencesSource = sequencesSource(idx);
sequencesTarget = sequencesTarget(idx);
```

Initialize the training progress plot.

```
figure
lineLossTrain = animatedline('Color',[0.85 0.325 0.098]);
ylim([0 inf])

xlabel("Iteration")
ylabel("Loss")
grid on
```

Initialize the values for the `adamupdate` function.

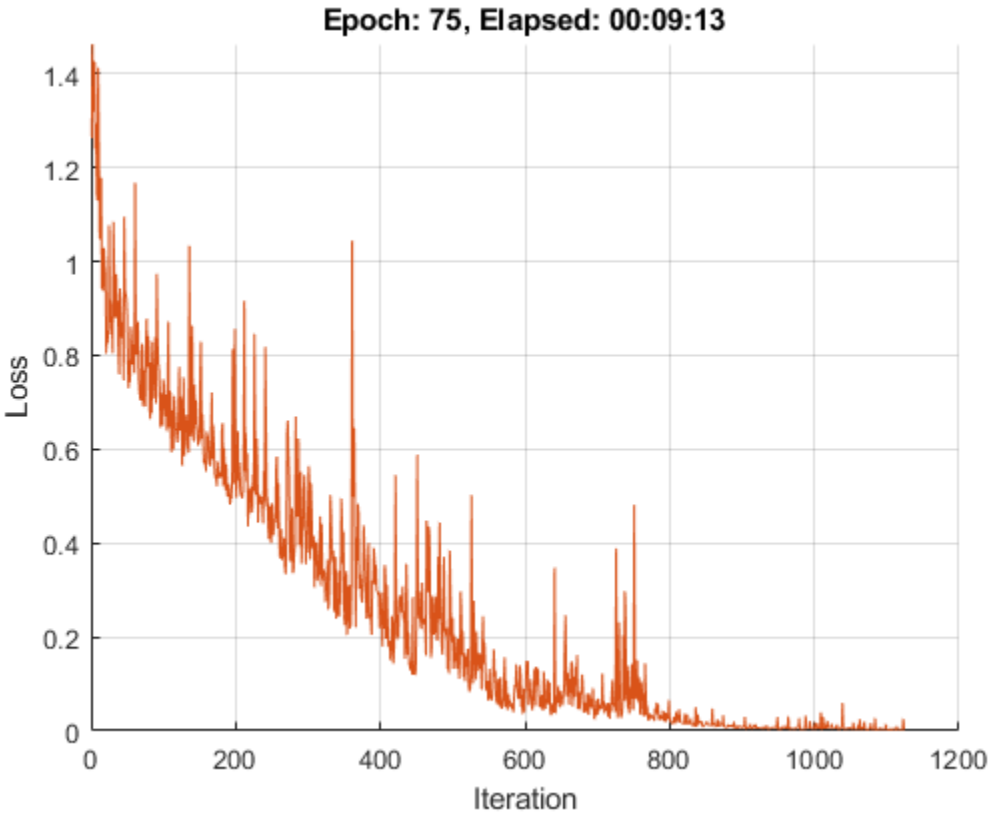
```
trailingAvg = [];
trailingAvgSq = [];
```

Train the model. For each mini-batch:

- Read a mini-batch of sequences and add padding.
- Convert the data to `dlarray`.
- Compute loss and gradients.
- Update the encoder and decoder model parameters using the `adamupdate` function.
- Update the training progress plot.

```
numObservations = numel(sequencesSource);
numIterationsPerEpoch = floor(numObservations/miniBatchSize);

iteration = 0;
start = tic;

% Loop over epochs.
for epoch = 1:numEpochs

    % Loop over mini-batches.
    for i = 1:numIterationsPerEpoch
        iteration = iteration + 1;

        % Read mini-batch of data and pad.
        idx = (i-1)*miniBatchSize+1:i*miniBatchSize;
        [X, sequenceLengthsSource] = padSequences(sequencesSource(idx), inputSize);
        [T, sequenceLengthsTarget] = padSequences(sequencesTarget(idx), outputSize);

        % Convert mini-batch of data to dlarray.
        dlX = dlarray(X);

        % Compute loss and gradients.
        [gradients, loss] = dlfeval(@modelGradients, parameters, dlX, T, ...
            sequenceLengthsSource, sequenceLengthsTarget, dropout);

        % Update parameters using adamupdate.
        [parameters, trailingAvg, trailingAvgSq] = adamupdate(parameters, gradients, trailingAvg
            iteration, learnRate, gradientDecayFactor, squaredGradientDecayFactor);

        % Display the training progress.
        D = duration(0,0,toc(start),'Format','hh:mm:ss');
```

```
        addpoints(lineLossTrain,iteration,double(gather(loss)))
        title("Epoch: " + epoch + ", Elapsed: " + string(D))
        drawnow
    end
end
```



Generate Translations

To generate translations for new data using the trained model, convert the text data to numeric sequences using the same steps as when training and input the sequences into the encoder-decoder model and convert the resulting sequences back into text using the token indices.

Preprocess the text data using the same steps as when training. Use the `transformText` function, listed at the end of the example, to split the text into characters and add the start and stop tokens.

```
strSource = dataTest{:,1};
strTarget = dataTest{:,2};
```

Translate the text using the `modelPredictions` function.

```
maxSequenceLength = 10;
delimiter = "";

strTranslated = translateText(parameters,strSource,maxSequenceLength,miniBatchSize, ...
    encSource,encTarget,startToken,stopToken,delimiter);
```

Create a table containing the test source text, target text, and translations.

```
tbl = table;
tbl.Source = strSource;
tbl.Target = strTarget;
tbl.Translated = strTranslated;
```

View a random selection of the translations.

```
idx = randperm(size(dataTest,1),miniBatchSize);
tbl(idx,:)
```

```
ans=32×3 table
    Source      Target      Translated
    ______    __________    __________

    "936"     "CMXXXVI"     "CMXXXVI"
    "423"     "CDXXIII"     "CDXXIII"
    "981"     "CMLXXXI"     "CMLXXXIX"
    "200"     "CC"          "CC"
    "224"     "CCXXIV"      "CCXXIV"
    "56"      "LVI"         "DLVI"
    "330"     "CCCXXX"      "CCCXXX"
    "336"     "CCCXXXVI"    "CCCXXXVI"
    "524"     "DXXIV"       "DXXIV"
    "860"     "DCCCLX"      "DCCCLX"
    "318"     "CCCXVIII"    "CCCXVIII"
    "902"     "CMII"        "CMII"
    "681"     "DCLXXXI"     "DCLXXXI"
    "299"     "CCXCIX"      "CCXCIX"
    "931"     "CMXXXI"      "CMXXXIX"
    "859"     "DCCCLIX"     "DCCCLIX"
      ⋮
```

Text Transformation Function

The `transformText` function preprocesses and tokenizes the input text for translation by splitting the text into characters and adding start and stop tokens. To translate text by splitting the text into words instead of characters, skip the first step.

```
function documents = transformText(str,startToken,stopToken)

str = strip(replace(str,"","," "));
str = startToken + str + stopToken;
documents = tokenizedDocument(str,'CustomTokens',[startToken stopToken]);

end
```

Sequence Padding Function

The `padSequences` function takes a mini-batch of sequences and a padding value and returns padded sequences with the corresponding padding masks.

```
function [X, sequenceLengths] = padSequences(sequences, paddingValue)

% Initialize mini-batch with padding.
numObservations = size(sequences,1);
sequenceLengths = cellfun(@(x) size(x,2), sequences);
maxLength = max(sequenceLengths);
```

```
X = repmat(paddingValue, [1 numObservations maxLength]);

% Insert sequences.
for n = 1:numObservations
    X(1,n,1:sequenceLengths(n)) = sequences{n};
end

end
```

Model Gradients Function

The `modelGradients` function takes the encoder and decoder model parameters, a mini-batch of input data and the padding masks corresponding to the input data, and the dropout probability and returns the gradients of the loss with respect to the learnable parameters in the models and the corresponding loss.

```
function [gradients, loss] = modelGradients(parameters, dlX, T, ...
    sequenceLengthsSource, sequenceLengthsTarget, dropout)

% Forward through encoder.
[dlZ, hiddenState] = modelEncoder(parameters.encoder, dlX, sequenceLengthsSource);

% Decoder Output.
doTeacherForcing = rand < 0.5;
sequenceLength = size(T,3);
dlY = decoderPredictions(parameters.decoder,dlZ,T,hiddenState,dropout,...
     doTeacherForcing,sequenceLength);

% Masked loss.
dlY = dlY(:,:,1:end-1);
T = T(:,:,2:end);
T = onehotencode(T,1,'ClassNames',1:size(dlY,1));
loss = maskedCrossEntropy(dlY,T,sequenceLengthsTarget-1);

% Update gradients.
gradients = dlgradient(loss, parameters);

% For plotting, return loss normalized by sequence length.
loss = extractdata(loss) ./ sequenceLength;

end
```

Encoder Model Function

The function `modelEncoder` takes the input data, the model parameters, the optional mask that is used to determine the correct outputs for training and returns the model output and the LSTM hidden state.

If `sequenceLengths` is empty, then the function does not mask the output. Specify and empty value for `sequenceLengths` when using the `modelEncoder` function for prediction.

```
function [dlZ, hiddenState] = modelEncoder(parametersEncoder, dlX, sequenceLengths)

% Embedding.
weights = parametersEncoder.emb.Weights;
dlZ = embed(dlX,weights,'DataFormat','CBT');

% LSTM 1.
```

```
inputWeights = parametersEncoder.lstm1.InputWeights;
recurrentWeights = parametersEncoder.lstm1.RecurrentWeights;
bias = parametersEncoder.lstm1.Bias;

numHiddenUnits = size(recurrentWeights, 2);
initialHiddenState = dlarray(zeros([numHiddenUnits 1]));
initialCellState = dlarray(zeros([numHiddenUnits 1]));

dlZ = lstm(dlZ, initialHiddenState, initialCellState, inputWeights, ...
    recurrentWeights, bias, 'DataFormat', 'CBT');

% LSTM 2.
inputWeights = parametersEncoder.lstm2.InputWeights;
recurrentWeights = parametersEncoder.lstm2.RecurrentWeights;
bias = parametersEncoder.lstm2.Bias;

[dlZ, hiddenState] = lstm(dlZ,initialHiddenState, initialCellState, ...
    inputWeights, recurrentWeights, bias, 'DataFormat', 'CBT');

% Masking for training.
if ~isempty(sequenceLengths)
    miniBatchSize = size(dlZ,2);
    for n = 1:miniBatchSize
        hiddenState(:,n) = dlZ(:,n,sequenceLengths(n));
    end
end

end
```

Decoder Model Function

The function `modelDecoder` takes the input data, the model parameters, the context vector, the LSTM initial hidden state, the outputs of the encoder, and the dropout probability and outputs the decoder output, the updated context vector, the updated LSTM state, and the attention scores.

```
function [dlY, context, hiddenState, attentionScores] = modelDecoder(parametersDecoder, dlX, con
    hiddenState, dlZ, dropout)

% Embedding.
weights = parametersDecoder.emb.Weights;
dlX = embed(dlX, weights,'DataFormat','CBT');

% RNN input.
sequenceLength = size(dlX,3);
dlY = cat(1, dlX, repmat(context, [1 1 sequenceLength]));

% LSTM 1.
inputWeights = parametersDecoder.lstm1.InputWeights;
recurrentWeights = parametersDecoder.lstm1.RecurrentWeights;
bias = parametersDecoder.lstm1.Bias;

initialCellState = dlarray(zeros(size(hiddenState)));

dlY = lstm(dlY, hiddenState, initialCellState, inputWeights, recurrentWeights, bias, 'DataFormat

% Dropout.
mask = ( rand(size(dlY), 'like', dlY) > dropout );
dlY = dlY.*mask;
```

```
% LSTM 2.
inputWeights = parametersDecoder.lstm2.InputWeights;
recurrentWeights = parametersDecoder.lstm2.RecurrentWeights;
bias = parametersDecoder.lstm2.Bias;

[dlY, hiddenState] = lstm(dlY, hiddenState, initialCellState,inputWeights, recurrentWeights, bias

% Attention.
weights = parametersDecoder.attn.Weights;
[attentionScores, context] = attention(hiddenState, dlZ, weights);

% Concatenate.
dlY = cat(1, dlY, repmat(context, [1 1 sequenceLength]));

% Fully connect.
weights = parametersDecoder.fc.Weights;
bias = parametersDecoder.fc.Bias;
dlY = fullyconnect(dlY,weights,bias,'DataFormat','CBT');

% Softmax.
dlY = softmax(dlY,'DataFormat','CBT');

end
```

Attention Function

The `attention` function returns the attention scores according to Luong "general" scoring and the updated context vector. The energy at each time step is the dot product of the hidden state and the learnable attention weights times the encoder output.

```
function [attentionScores, context] = attention(hiddenState, encoderOutputs, weights)

% Initialize attention energies.
[miniBatchSize, sequenceLength] = size(encoderOutputs, 2:3);
attentionEnergies = zeros([sequenceLength miniBatchSize],'like',hiddenState);

% Attention energies.
hWX = hiddenState .* pagemtimes(weights,encoderOutputs);
for tt = 1:sequenceLength
    attentionEnergies(tt, :) = sum(hWX(:, :, tt), 1);
end

% Attention scores.
attentionScores = softmax(attentionEnergies, 'DataFormat', 'CB');

% Context.
encoderOutputs = permute(encoderOutputs, [1 3 2]);
attentionScores = permute(attentionScores,[1 3 2]);
context = pagemtimes(encoderOutputs,attentionScores);
context = squeeze(context);

end
```

Masked Cross Entropy Loss

The `maskedCrossEntropy` function calculates the loss between the specified input sequences and target sequences ignoring any time steps containing padding using the specified vector of sequence lengths.

```
function loss = maskedCrossEntropy(dlY,T,sequenceLengths)

% Initialize loss.
loss = 0;

% Loop over mini-batch.
miniBatchSize = size(dlY,2);
for n = 1:miniBatchSize
    idx = 1:sequenceLengths(n);
    loss = loss + crossentropy(dlY(:,n,idx), T(:,n,idx),'DataFormat','CBT');
end

% Normalize.
loss = loss / miniBatchSize;

end
```

Decoder Model Predicitons Function

The `decoderModelPredictions` function returns the predicted sequence `dlY` given the input sequence, target sequence, hidden state, dropout probability, flag to enable teacher forcing, and the sequence length.

```
function dlY = decoderPredictions(parametersDecoder,dlZ,T,hiddenState,dropout, ...
    doTeacherForcing,sequenceLength)

% Convert to dlarray.
dlT = dlarray(T);

% Initialize context.
miniBatchSize = size(dlT,2);
numHiddenUnits = size(dlZ,1);
context = zeros([numHiddenUnits miniBatchSize],'like',dlZ);

if doTeacherForcing
    % Forward through decoder.
    dlY = modelDecoder(parametersDecoder, dlT, context, hiddenState, dlZ, dropout);
else
    % Get first time step for decoder.
    decoderInput = dlT(:,:,1);

    % Initialize output.
    numClasses = numel(parametersDecoder.fc.Bias);
    dlY = zeros([numClasses miniBatchSize sequenceLength],'like',decoderInput);

    % Loop over time steps.
    for t = 1:sequenceLength
        % Forward through decoder.
        [dlY(:,:,t), context, hiddenState] = modelDecoder(parametersDecoder, decoderInput, contex
            hiddenState, dlZ, dropout);

        % Update decoder input.
```

```
        [~, decoderInput] = max(dlY(:,:,t),[],1);
    end
end

end
```

Text Translation Function

The `translateText` function translates an array of text by iterating over mini-batches. The function takes as input the model parameters, the input string array, a maximum sequence length, the mini-batch size, the source and target word encoding objects, the start and stop tokens, and the delimiter for assembling the output.

```
function strTranslated = translateText(parameters,strSource,maxSequenceLength,miniBatchSize, ...
    encSource,encTarget,startToken,stopToken,delimiter)

% Transform text.
documentsSource = transformText(strSource,startToken,stopToken);
sequencesSource = doc2sequence(encSource,documentsSource, ...
    'PaddingDirection','right', ...
    'PaddingValue',encSource.NumWords + 1);

% Convert to dlarray.
X = cat(3,sequencesSource{:});
X = permute(X,[1 3 2]);
dlX = dlarray(X);

% Initialize output.
numObservations = numel(strSource);
strTranslated = strings(numObservations,1);

% Loop over mini-batches.
numIterations = ceil(numObservations / miniBatchSize);
for i = 1:numIterations
    idxMiniBatch = (i-1)*miniBatchSize+1:min(i*miniBatchSize,numObservations);
    miniBatchSize = numel(idxMiniBatch);

    % Encode using model encoder.
    sequenceLengths = [];
    [dlZ, hiddenState] = modelEncoder(parameters.encoder, dlX(:,idxMiniBatch,:), sequenceLengths

    % Decoder predictions.
    doTeacherForcing = false;
    dropout = 0;
    decoderInput = repmat(word2ind(encTarget,startToken),[1 miniBatchSize]);
    decoderInput = dlarray(decoderInput);
    dlY = decoderPredictions(parameters.decoder,dlZ,decoderInput,hiddenState,dropout, ...
        doTeacherForcing,maxSequenceLength);
    [~, idxPred] = max(extractdata(dlY), [], 1);

    % Keep translating flag.
    idxStop = word2ind(encTarget,stopToken);
    keepTranslating = idxPred ~= idxStop;

    % Loop over time steps.
    t = 1;
    while t <= maxSequenceLength && any(keepTranslating(:,:,t))
```

```
        % Update output.
        newWords = ind2word(encTarget, idxPred(:,:,t))';
        idxUpdate = idxMiniBatch(keepTranslating(:,:,t));
        strTranslated(idxUpdate) = strTranslated(idxUpdate) + delimiter + newWords(keepTranslati

        t = t + 1;
    end
end

end
```

See Also

`adamupdate` | `crossentropy` | `dlarray` | `dlfeval` | `dlgradient` | `dlupdate` | `doc2sequence` | `lstm` | `softmax` | `tokenizedDocument` | `word2ind` | `wordEncoding`

More About

- “Define Custom Training Loops, Loss Functions, and Networks” (Deep Learning Toolbox)
- “Prepare Text Data for Analysis” on page 1-10
- “Analyze Text Data Using Topic Models” on page 2-13
- “Classify Text Data Using Deep Learning” on page 2-65
- “Classify Text Data Using Convolutional Neural Network” on page 2-73
- “Train a Sentiment Classifier” on page 2-51
- “Visualize Word Embeddings Using Text Scatter Plots” on page 3-8

Classify Out-of-Memory Text Data Using Deep Learning

This example shows how to classify out-of-memory text data with a deep learning network using a transformed datastore.

A transformed datastore transforms or processes data read from an underlying datastore. You can use a transformed datastore as a source of training, validation, test, and prediction data sets for deep learning applications. Use transformed datastores to read out-of-memory data or to perform specific preprocessing operations when reading batches of data.

When training the network, the software creates mini-batches of sequences of the same length by padding, truncating, or splitting the input data. The `trainingOptions` function provides options to pad and truncate input sequences, however, these options are not well suited for sequences of word vectors. Furthermore, this function does not support padding data in a custom datastore. Instead, you must pad and truncate the sequences manually. If you *left-pad* and truncate the sequences of word vectors, then the training might improve.

The “Classify Text Data Using Deep Learning” on page 2-65 example manually truncates and pads all the documents to the same length. This process adds lots of padding to very short documents and discards lots of data from very long documents.

Alternatively, to prevent adding too much padding or discarding too much data, create a transformed datastore that inputs mini-batches into the network. The datastore created in this example converts mini-batches of documents to sequences or word indices and left-pads each mini-batch to the length of the longest document in the mini-batch.

Load Pretrained Word Embedding

The datastore requires a word embedding to convert documents to sequences of vectors. Load a pretrained word embedding using `fastTextWordEmbedding`. This function requires Text Analytics Toolbox™ Model *for fastText English 16 Billion Token Word Embedding* support package. If this support package is not installed, then the function provides a download link.

```
emb = fastTextWordEmbedding;
```

Load Data

Create a tabular text datastore from the data in `factoryReports.csv`. Specify to read the data from the `"Description"` and `"Category"` columns only.

```
filenameTrain = "factoryReports.csv";
textName = "Description";
labelName = "Category";
ttdsTrain = tabularTextDatastore(filenameTrain,'SelectedVariableNames',[textName labelName]);
```

View a preview of the datastore.

```
preview(ttdsTrain)
```

```
ans=8×2 table
                                Description                                                Category
    ____________________________________________________________________________    ____________________

    {'Items are occasionally getting stuck in the scanner spools.'        }    {'Mechanical Failu
    {'Loud rattling and banging sounds are coming from assembler pistons.'}    {'Mechanical Failu
    {'There are cuts to the power when starting the plant.'               }    {'Electronic Failu
```

```
    {'Fried capacitors in the assembler.'                                  }    {'Electronic Failu
    {'Mixer tripped the fuses.'                                            }    {'Electronic Failu
    {'Burst pipe in the constructing agent is spraying coolant.'           }    {'Leak'
    {'A fuse is blown in the mixer.'                                       }    {'Electronic Failu
    {'Things continue to tumble off of the belt.'                          }    {'Mechanical Failu
```

Transform Datastore

Create a custom transform function that converts data read from the datastore to a table containing the predictors and the responses. The `transformText` function takes the data read from a `tabularTextDatastore` object and returns a table of predictors and responses. The predictors are *C*-by-*S* arrays of word vectors given by the word embedding `emb`, where *C* is the embedding dimension and *S* is the sequence length. The responses are categorical labels over the classes.

To get the class names, read the labels from the training data using the `readLabels` function, listed and the end of the example, and find the unique class names.

```
labels = readLabels(ttdsTrain,labelName);
classNames = unique(labels);
numObservations = numel(labels);
```

Because tabular text datastores can read multiple rows of data in a single read, you can process a full mini-batch of data in the transform function. To ensure that the transform function processes a full mini-batch of data, set the read size of the tabular text datastore to the mini-batch size that will be used for training.

```
miniBatchSize = 64;
ttdsTrain.ReadSize = miniBatchSize;
```

To convert the output of the tabular text data to sequences for training, transform the datastore using the `transform` function.

```
tdsTrain = transform(ttdsTrain, @(data) transformText(data,emb,classNames))
```

```
tdsTrain =
  TransformedDatastore with properties:

       UnderlyingDatastore: [1×1 matlab.io.datastore.TabularTextDatastore]
    SupportedOutputFormats: ["txt"    "csv"    "xlsx"    "xls"    "parquet"    "parq"    "png"
                Transforms: {@(data)transformText(data,emb,classNames)}
               IncludeInfo: 0
```

Preview of the transformed datastore. The predictors are *C*-by-*S* arrays, where *S* is the sequence length and *C* is the number of features (the embedding dimension). The responses are the categorical labels.

```
preview(tdsTrain)
```

```
ans=8×2 table
      predictors           responses
    _______________    ________________

    {300×11 single}    Mechanical Failure
    {300×11 single}    Mechanical Failure
    {300×11 single}    Electronic Failure
    {300×11 single}    Electronic Failure
```

```
    {300×11 single}    Electronic Failure
    {300×11 single}    Leak
    {300×11 single}    Electronic Failure
    {300×11 single}    Mechanical Failure
```

Create and Train LSTM Network

Define the LSTM network architecture. To input sequence data into the network, include a sequence input layer and set the input size to the embedding dimension. Next, include an LSTM layer with 180 hidden units. To use the LSTM layer for a sequence-to-label classification problem, set the output mode to `'last'`. Finally, add a fully connected layer with output size equal to the number of classes, a softmax layer, and a classification layer.

```
numFeatures = emb.Dimension;
numHiddenUnits = 180;
numClasses = numel(classNames);
layers = [ ...
    sequenceInputLayer(numFeatures)
    lstmLayer(numHiddenUnits,'OutputMode','last')
    fullyConnectedLayer(numClasses)
    softmaxLayer
    classificationLayer];
```

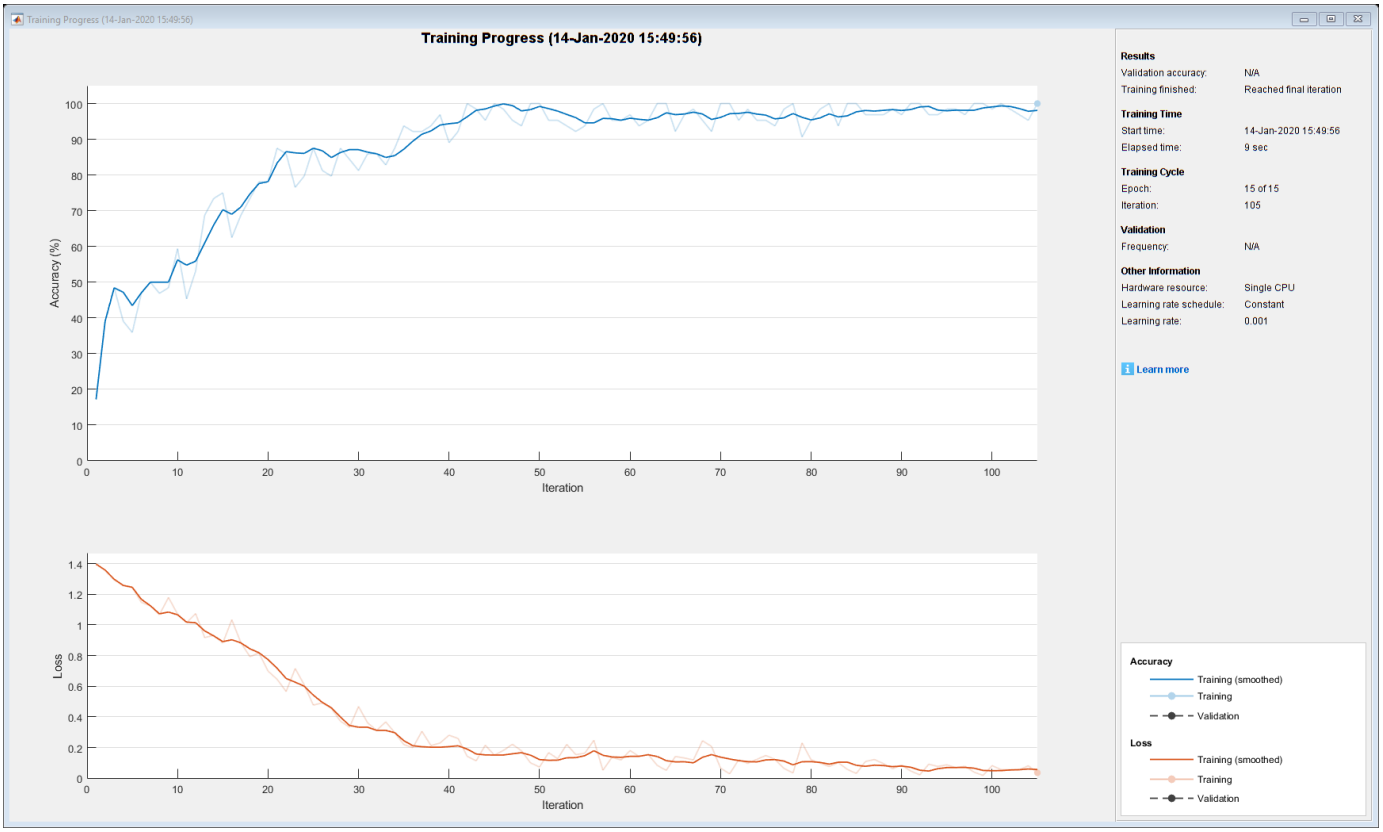
Specify the training options. Specify the solver to be `'adam'` and the gradient threshold to be 2. The datastore does not support shuffling, so set `'Shuffle'`, to `'never'`. Validate the network once per epoch. To monitor the training progress, set the `'Plots'` option to `'training-progress'`. To suppress verbose output, set `'Verbose'` to `false`.

By default, `trainNetwork` uses a GPU if one is available (requires Parallel Computing Toolbox™ and a CUDA® enabled GPU with compute capability 3.0 or higher). Otherwise, it uses the CPU. To specify the execution environment manually, use the `'ExecutionEnvironment'` name-value pair argument of `trainingOptions`. Training on a CPU can take significantly longer than training on a GPU.

```
numIterationsPerEpoch = floor(numObservations / miniBatchSize);

options = trainingOptions('adam', ...
    'MaxEpochs',15, ...
    'MiniBatchSize',miniBatchSize, ...
    'GradientThreshold',2, ...
    'Shuffle','never', ...
    'Plots','training-progress', ...
    'Verbose',false);
```

Train the LSTM network using the `trainNetwork` function.

```
net = trainNetwork(tdsTrain,layers,options);
```



Predict Using New Data

Classify the event type of three new reports. Create a string array containing the new reports.

```
reportsNew = [ ...
    "Coolant is pooling underneath sorter."
    "Sorter blows fuses at start up."
    "There are some very loud rattling sounds coming from the assembler."];
```

Preprocess the text data using the preprocessing steps as the training documents.

```
documentsNew = preprocessText(reportsNew);
```

Convert the text data to sequences of embedding vectors using `doc2sequence`.

```
XNew = doc2sequence(emb,documentsNew);
```

Classify the new sequences using the trained LSTM network.

```
labelsNew = classify(net,XNew)
```

```
labelsNew = 3×1 categorical
     Leak
     Electronic Failure
     Mechanical Failure
```

Transform Text Function

The `transformText` function takes the data read from a `tabularTextDatastore` object and returns a table of predictors and responses. The predictors are *C*-by-*S* arrays of word vectors given by the word embedding `emb`, where *C* is the embedding dimension and *S* is the sequence length. The responses are categorical labels over the classes in `classNames`.

```
function dataTransformed = transformText(data,emb,classNames)

% Preprocess documents.
textData = data{:,1};
documents = preprocessText(textData);

% Convert to sequences.
predictors = doc2sequence(emb,documents);

% Read labels.
labels = data{:,2};
responses = categorical(labels,classNames);

% Convert data to table.
dataTransformed = table(predictors,responses);

end
```

Preprocessing Function

The function `preprocessText` performs these steps:

1. Tokenize the text using `tokenizedDocument`.
2. Convert the text to lowercase using `lower`.
3. Erase the punctuation using `erasePunctuation`.

```
function documents = preprocessText(textData)

documents = tokenizedDocument(textData);
documents = lower(documents);
documents = erasePunctuation(documents);

end
```

Read Labels Function

The `readLabels` function creates a copy of the `tabularTextDatastore` object `ttds` and reads the labels from the `labelName` column.

```
function labels = readLabels(ttds,labelName)

ttdsNew = copy(ttds);
ttdsNew.SelectedVariableNames = labelName;
tbl = readall(ttdsNew);
labels = tbl.(labelName);
```

```
end
```

See Also

`doc2sequence` | `fastTextWordEmbedding` | `lstmLayer` | `sequenceInputLayer` | `tokenizedDocument` | `trainNetwork` | `trainingOptions` | `transform` | `wordEmbeddingLayer`

Related Examples

- “Create Simple Text Model for Classification” on page 2-2
- “Analyze Text Data Using Topic Models” on page 2-13
- “Analyze Text Data Using Multiword Phrases” on page 2-7
- “Train a Sentiment Classifier” on page 2-51
- “Sequence Classification Using Deep Learning” (Deep Learning Toolbox)
- “Deep Learning in MATLAB” (Deep Learning Toolbox)

Pride and Prejudice and MATLAB

This example shows how to train a deep learning LSTM network to generate text using character embeddings.

To train a deep learning network for text generation, train a sequence-to-sequence LSTM network to predict the next character in a sequence of characters. To train the network to predict the next character, specify the responses to be the input sequences shifted by one time step.

To use character embeddings, convert each training observation to a sequence of integers, where the integers index into a vocabulary of characters. Include a word embedding layer in the network which learns an embedding of the characters and maps the integers to vectors.

Load Training Data

Read the HTML code from The Project Gutenberg EBook of Pride and Prejudice, by Jane Austen and parse it using `webread` and `htmlTree`.

```
url = "https://www.gutenberg.org/files/1342/1342-h/1342-h.htm";
code = webread(url);
tree = htmlTree(code);
```

Extract the paragraphs by finding the `p` elements. Specify to ignore paragraph elements with class `"toc"` using the CSS selector `':not(.toc)'`.

```
paragraphs = findElement(tree,'p:not(.toc)');
```

Extract the text data from the paragraphs using `extractHTMLText`. and remove the empty strings.

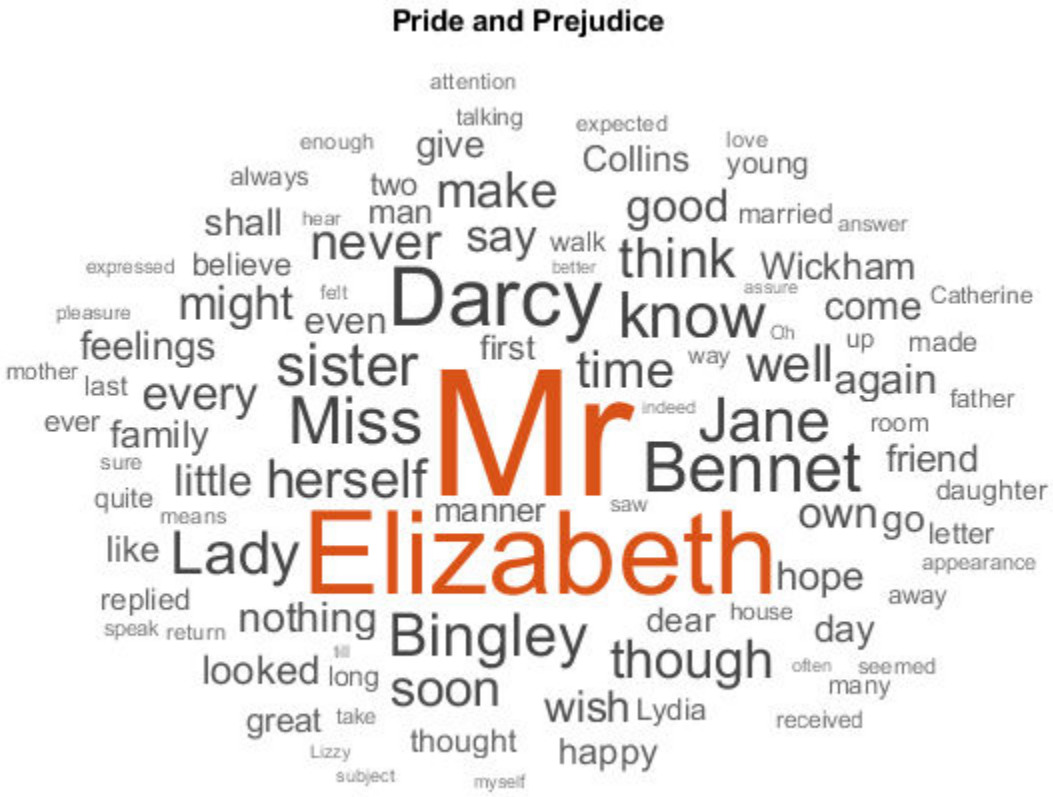
```
textData = extractHTMLText(paragraphs);
textData(textData == "") = [];
```

Remove strings shorter than 20 characters.

```
idx = strlength(textData) < 20;
textData(idx) = [];
```

Visualize the text data in a word cloud.

```
figure
wordcloud(textData);
title("Pride and Prejudice")
```



Convert Text Data to Sequences

Convert the text data to sequences of character indices for the predictors and categorical sequences for the responses.

The categorical function treats newline and whitespace entries as undefined. To create categorical elements for these characters, replace them with the special characters "¶" (pilcrow, `"\x00B6"`) and "·" (middle dot, `"\x00B7"`) respectively. To prevent ambiguity, you must choose special characters that do not appear in the text. These characters do not appear in the training data so can be used for this purpose.

```
newlineCharacter = compose("\x00B6");
whitespaceCharacter = compose("\x00B7");
textData = replace(textData,[newline " "],[newlineCharacter whitespaceCharacter]);
```

Loop over the text data and create a sequence of character indices representing the characters of each observation and a categorical sequence of characters for the responses. To denote the end of each observation, include the special character "ETX" (end of text, `"\x2403"`).

```
endOfTextCharacter = compose("\x2403");
numDocuments = numel(textData);
for i = 1:numDocuments
    characters = textData{i};
    X = double(characters);

    % Create vector of categorical responses with end of text character.
    charactersShifted = [cellstr(characters(2:end)')' endOfTextCharacter];
```

```
    Y = categorical(charactersShifted);

    XTrain{i} = X;
    YTrain{i} = Y;
end
```

During training, by default, the software splits the training data into mini-batches and pads the sequences so that they have the same length. Too much padding can have a negative impact on the network performance.

To prevent the training process from adding too much padding, you can sort the training data by sequence length, and choose a mini-batch size so that sequences in a mini-batch have a similar length.

Get the sequence lengths for each observation.

```
numObservations = numel(XTrain);
for i=1:numObservations
    sequence = XTrain{i};
    sequenceLengths(i) = size(sequence,2);
end
```

Sort the data by sequence length.

```
[~,idx] = sort(sequenceLengths);
XTrain = XTrain(idx);
YTrain = YTrain(idx);
```

Create and Train LSTM Network

Define the LSTM architecture. Specify a sequence-to-sequence LSTM classification network with 400 hidden units. Set the input size to be the feature dimension of the training data. For sequences of character indices, the feature dimension is 1. Specify a word embedding layer with dimension 200 and specify the number of words (which correspond to characters) to be the highest character value in the input data. Set the output size of the fully connected layer to be the number of categories in the responses. To help prevent overfitting, include a dropout layer after the LSTM layer.

The word embedding layer learns an embedding of characters and maps each character to a 200-dimension vector.

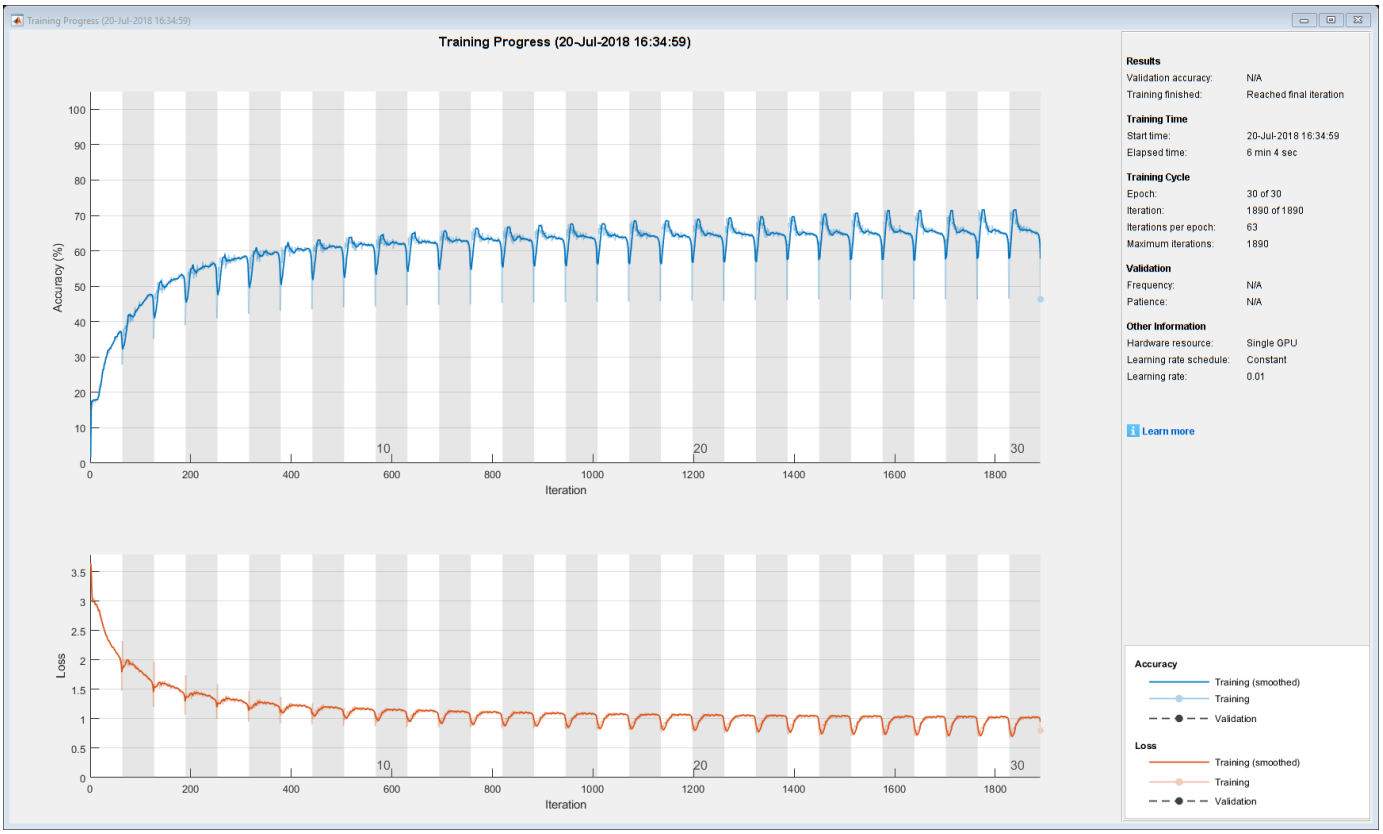
```
inputSize = size(XTrain{1},1);
numClasses = numel(categories([YTrain{:}]));
numCharacters = max([textData{:}]);

layers = [
    sequenceInputLayer(inputSize)
    wordEmbeddingLayer(200,numCharacters)
    lstmLayer(400,'OutputMode','sequence')
    dropoutLayer(0.2);
    fullyConnectedLayer(numClasses)
    softmaxLayer
    classificationLayer];
```

Specify the training options. Specify to train with a mini-batch size of 32 and initial learn rate 0.01. To prevent the gradients from exploding, set the gradient threshold to 1. To ensure the data remains sorted, set `'Shuffle'` to `'never'`. To monitor the training progress, set the `'Plots'` option to `'training-progress'`. To suppress verbose output, set `'Verbose'` to `false`.

```
options = trainingOptions('adam', ...
    'MiniBatchSize',32,...
    'InitialLearnRate',0.01, ...
    'GradientThreshold',1, ...
    'Shuffle','never', ...
    'Plots','training-progress', ...
    'Verbose',false);
```

Train the network.

```
net = trainNetwork(XTrain,YTrain,layers,options);
```



Generate New Text

Generate the first character of the text by sampling a character from a probability distribution according to the first characters of the text in the training data. Generate the remaining characters by using the trained LSTM network to predict the next sequence using the current sequence of generated text. Keep generating characters one-by-one until the network predicts the "end of text" character.

Sample the first character according to the distribution of the first characters in the training data.

```
initialCharacters = extractBefore(textData,2);
firstCharacter = datasample(initialCharacters,1);
generatedText = firstCharacter;
```

Convert the first character to a numeric index.

```
X = double(char(firstCharacter));
```

For the remaining predictions, sample the next character according to the prediction scores of the network. The prediction scores represent the probability distribution of the next character. Sample the characters from the vocabulary of characters given by the class names of the output layer of the network. Get the vocabulary from the classification layer of the network.

```
vocabulary = string(net.Layers(end).ClassNames);
```

Make predictions character by character using `predictAndUpdateState`. For each prediction, input the index of the previous character. Stop predicting when the network predicts the end of text character or when the generated text is 500 characters long. For large collections of data, long sequences, or large networks, predictions on the GPU are usually faster to compute than predictions on the CPU. Otherwise, predictions on the CPU are usually faster to compute. For single time step predictions, use the CPU. To use the CPU for prediction, set the `'ExecutionEnvironment'` option of `predictAndUpdateState` to `'cpu'`.

```
maxLength = 500;
while strlength(generatedText) < maxLength
    % Predict the next character scores.
    [net,characterScores] = predictAndUpdateState(net,X,'ExecutionEnvironment','cpu');

    % Sample the next character.
    newCharacter = datasample(vocabulary,1,'Weights',characterScores);

    % Stop predicting at the end of text.
    if newCharacter == endOfTextCharacter
        break
    end

    % Add the character to the generated text.
    generatedText = generatedText + newCharacter;

    % Get the numeric index of the character.
    X = double(char(newCharacter));
end
```

Reconstruct the generated text by replacing the special characters with their corresponding whitespace and new line characters.

```
generatedText = replace(generatedText,[newlineCharacter whitespaceCharacter],[newline " "])
```

```
generatedText =
""“I wish Mr. Darcy, upon latter of my sort sincerely fixed in the regard to relanth. We were to
```

To generate multiple pieces of text, reset the network state between generations using `resetState`.

```
net = resetState(net);
```

See Also

`doc2sequence` | `extractHTMLText` | `findElement` | `htmlTree` | `lstmLayer` | `sequenceInputLayer` | `tokenizedDocument` | `trainNetwork` | `trainingOptions` | `wordEmbeddingLayer` | `wordcloud`

Related Examples

- “Generate Text Using Deep Learning” (Deep Learning Toolbox)

- “Word-By-Word Text Generation Using Deep Learning” on page 2-142
- “Create Simple Text Model for Classification” on page 2-2
- “Analyze Text Data Using Topic Models” on page 2-13
- “Analyze Text Data Using Multiword Phrases” on page 2-7
- “Train a Sentiment Classifier” on page 2-51
- “Sequence Classification Using Deep Learning” (Deep Learning Toolbox)
- “Deep Learning in MATLAB” (Deep Learning Toolbox)

Word-By-Word Text Generation Using Deep Learning

This example shows how to train a deep learning LSTM network to generate text word-by-word.

To train a deep learning network for word-by-word text generation, train a sequence-to-sequence LSTM network to predict the next word in a sequence of words. To train the network to predict the next word, specify the responses to be the input sequences shifted by one time step.

This example reads text from a website. It reads and parses the HTML code to extract the relevant text, then uses a custom mini-batch datastore `documentGenerationDatastore` to input the documents to the network as mini-batches of sequence data. The datastore converts documents to sequences of numeric word indices. The deep learning network is an LSTM network that contains a word embedding layer.

A mini-batch datastore is an implementation of a datastore with support for reading data in batches. You can use a mini-batch datastore as a source of training, validation, test, and prediction data sets for deep learning applications. Use mini-batch datastores to read out-of-memory data or to perform specific preprocessing operations when reading batches of data.

You can adapt the custom mini-batch datastore `documentGenerationDatastore.m` to your data by customizing the functions. For an example showing how to create your own custom mini-batch datastore, see “Develop Custom Mini-Batch Datastore” (Deep Learning Toolbox).

Load Training Data

Load the training data. Read the HTML code from Alice's Adventures in Wonderland by Lewis Carroll from Project Gutenberg.

```
url = "https://www.gutenberg.org/files/11/11-h/11-h.htm";
code = webread(url);
```

Parse HTML Code

The HTML code contains the relevant text inside <p> (paragraph) elements. Extract the relevant text by parsing the HTML code using `htmlTree` and then finding all the elements with element name `"p"`.

```
tree = htmlTree(code);
selector = "p";
subtrees = findElement(tree,selector);
```

Extract the text data from the HTML subtrees using `extractHTMLText` and view the first 10 paragraphs.

```
textData = extractHTMLText(subtrees);
textData(1:10)
```

```
ans = 10×1 string array
    ""
    ""
    ""
    ""
    ""
    ""
    "Alice was beginning to get very tired of sitting by her sister on the bank, and of having no
    "So she was considering in her own mind (as well as she could, for the hot day made her feel
```

```
    "There was nothing so very remarkable in that; nor did Alice think it so very much out of the
    "In another moment down went Alice after it, never once considering how in the world she was
```

Remove the empty paragraphs and view the first 10 remaining paragraphs.

```
textData(textData == "") = [];
textData(1:10)
```

```
ans = 10×1 string array
    "Alice was beginning to get very tired of sitting by her sister on the bank, and of having no
    "So she was considering in her own mind (as well as she could, for the hot day made her feel
    "There was nothing so very remarkable in that; nor did Alice think it so very much out of the
    "In another moment down went Alice after it, never once considering how in the world she was
    "The rabbit-hole went straight on like a tunnel for some way, and then dipped suddenly down,
    "Either the well was very deep, or she fell very slowly, for she had plenty of time as she we
    "'Well!' thought Alice to herself, 'after such a fall as this, I shall think nothing of tumbl
    "Down, down, down. Would the fall never come to an end! 'I wonder how many miles I've fallen
    "Presently she began again. 'I wonder if I shall fall right through the earth! How funny it'l
    "Down, down, down. There was nothing else to do, so Alice soon began talking again. 'Dinah'll
```

Visualize the text data in a word cloud.

```
figure
wordcloud(textData);
title("Alice's Adventures in Wonderland")
```

Prepare Data for Training

Create a datastore that contains the data for training using `documentGenerationDatastore`. To create the datastore, first save the custom mini-batch datastore `documentGenerationDatastore.m` to the path. For the predictors, this datastore converts the documents into sequences of word indices using a word encoding. The first word index for each document corresponds to a "start of text" token. The "start of text" token is given by the string `"startOfText"`. For the responses, the datastore returns categorical sequences of the words shifted by one.

Tokenize the text data using `tokenizedDocument`.

```
documents = tokenizedDocument(textData);
```

Create a document generation datastore using the tokenized documents.

```
ds = documentGenerationDatastore(documents);
```

To reduce the amount of padding added to the sequences, sort the documents in the datastore by sequence length.

```
ds = sort(ds);
```

Create and Train LSTM Network

Define the LSTM network architecture. To input sequence data into the network, include a sequence input layer and set the input size to 1. Next, include a word embedding layer of dimension 100 and the same number of words as the word encoding. Next, include an LSTM layer and specify the hidden size to be 100. Finally, add a fully connected layer with the same size as the number of classes, a softmax layer, and a classification layer. The number of classes is the number of words in the vocabulary plus an extra class for the "end of text" class.

```
inputSize = 1;
embeddingDimension = 100;
numWords = numel(ds.Encoding.Vocabulary);
numClasses = numWords + 1;

layers = [
    sequenceInputLayer(inputSize)
    wordEmbeddingLayer(embeddingDimension,numWords)
    lstmLayer(100)
    dropoutLayer(0.2)
    fullyConnectedLayer(numClasses)
    softmaxLayer
    classificationLayer];
```

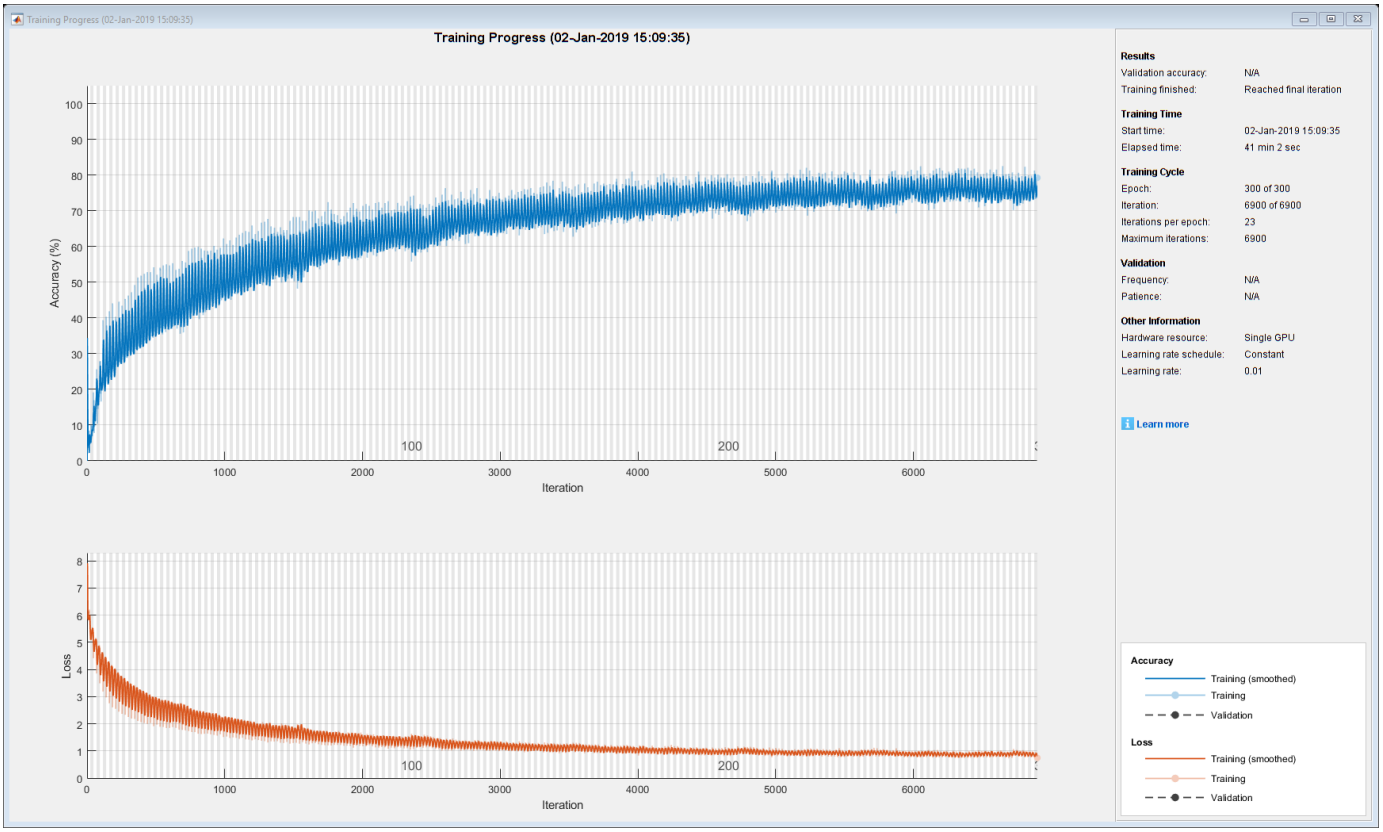
Specify the training options. Specify the solver to be `'adam'`. Train for 300 epochs with learn rate 0.01. Set the mini-batch size to 32. To keep the data sorted by sequence length, set the `'Shuffle'` option to `'never'`. To monitor the training progress, set the `'Plots'` option to `'training-progress'`. To suppress verbose output, set `'Verbose'` to `false`.

```
options = trainingOptions('adam', ...
    'MaxEpochs',300, ...
    'InitialLearnRate',0.01, ...
    'MiniBatchSize',32, ...
    'Shuffle','never', ...
```

```
    'Plots','training-progress', ...
    'Verbose',false);
```

Train the network using `trainNetwork`.

```
net = trainNetwork(ds,layers,options);
```



Generate New Text

Generate the first word of the text by sampling a word from a probability distribution according to the first words of the text in the training data. Generate the remaining words by using the trained LSTM network to predict the next time step using the current sequence of generated text. Keep generating words one-by-one until the network predicts the "end of text" word.

To make the first prediction using the network, input the index that represents the "start of text" token. Find the index by using the `word2ind` function with the word encoding used by the document datastore.

```
enc = ds.Encoding;
wordIndex = word2ind(enc,"startOfText")
```

```
wordIndex = 1
```

For the remaining predictions, sample the next word according to the prediction scores of the network. The prediction scores represent the probability distribution of the next word. Sample the words from the vocabulary given by the class names of the output layer of the network.

```
vocabulary = string(net.Layers(end).Classes);
```

Make predictions word by word using `predictAndUpdateState`. For each prediction, input the index of the previous word. Stop predicting when the network predicts the end of text word or when the generated text is 500 characters long. For large collections of data, long sequences, or large networks, predictions on the GPU are usually faster to compute than predictions on the CPU. Otherwise, predictions on the CPU are usually faster to compute. For single time step predictions, use the CPU. To use the CPU for prediction, set the `'ExecutionEnvironment'` option of `predictAndUpdateState` to `'cpu'`.

```
generatedText = "";
maxLength = 500;
while strlength(generatedText) < maxLength
    % Predict the next word scores.
    [net,wordScores] = predictAndUpdateState(net,wordIndex,'ExecutionEnvironment','cpu');

    % Sample the next word.
    newWord = datasample(vocabulary,1,'Weights',wordScores);

    % Stop predicting at the end of text.
    if newWord == "EndOfText"
        break
    end

    % Add the word to the generated text.
    generatedText = generatedText + " " + newWord;

    % Find the word index for the next input.
    wordIndex = word2ind(enc,newWord);
end
```

The generation process introduces whitespace characters between each prediction, which means that some punctuation characters appear with unnecessary spaces before and after. Reconstruct the generated text by removing the spaces before and after the appropriate punctuation characters.

Remove the spaces that appear before the specified punctuation characters.

```
punctuationCharacters = ["." "," "’" ")" ":" "?" "!"];
generatedText = replace(generatedText," " + punctuationCharacters,punctuationCharacters);
```

Remove the spaces that appear after the specified punctuation characters.

```
punctuationCharacters = ["(" "‘"];
generatedText = replace(generatedText,punctuationCharacters + " ",punctuationCharacters)
```

```
generatedText =
" ‘Sure, it’s a good Turtle!’ said the Queen in a low, weak voice."
```

To generate multiple pieces of text, reset the network state between generations using `resetState`.

```
net = resetState(net);
```

See Also

`doc2sequence` | `extractHTMLText` | `findElement` | `htmlTree` | `lstmLayer` | `sequenceInputLayer` | `tokenizedDocument` | `trainNetwork` | `trainingOptions` | `wordEmbeddingLayer` | `wordcloud`

Related Examples

- “Generate Text Using Deep Learning” (Deep Learning Toolbox)
- “Create Simple Text Model for Classification” on page 2-2
- “Analyze Text Data Using Topic Models” on page 2-13
- “Analyze Text Data Using Multiword Phrases” on page 2-7
- “Train a Sentiment Classifier” on page 2-51
- “Sequence Classification Using Deep Learning” (Deep Learning Toolbox)
- “Deep Learning in MATLAB” (Deep Learning Toolbox)

Generate Text Using Autoencoders

This example shows how to generate text data using autoencoders.

An autoencoder is a type of deep learning network that is trained to replicate its input. An autoencoder consists of two smaller networks: and encoder and a decoder. The encoder maps the input data to a feature vector in some latent space. The decoder reconstructs data using vectors in this latent space.

The training process is unsupervised. In other words, the model does not require labeled data. To generate text, you can use the decoder to reconstruct text from arbitrary input.

This example trains an autoencoder to generate text. The encoder uses a word embedding and an LSTM operation to map the input text into latent vectors. The decoder uses an LSTM operation and the same embedding to reconstruct the text from the latent vectors.

Load Data

The file `sonnets.txt` contains all of Shakespeare's sonnets in a single text file.

Read the Shakespeare's Sonnets data from the file `"sonnets.txt"`.

```
filename = "sonnets.txt";
textData = fileread(filename);
```

The sonnets are indented by two whitespace characters. Remove the indentations using `replace` and split the text into separate lines using the `split` function. Remove the header from the first nine elements and the short sonnet titles.

```
textData = replace(textData,"  ","");
textData = split(textData,newline);
textData(1:9) = [];
textData(strlength(textData)<5) = [];
```

Prepare Data

Create a function that tokenizes and preprocesses the text data. The function `preprocessText`, listed at the end of the example, performs these steps:

1. Prepends and appends each input string with the specified start and stop tokens, respectively.
2. Tokenize the text using `tokenizedDocument`.

Preprocess the text data and specify the start and stop tokens `"<start>"` and `"<stop>"`, respectively.

```
startToken = "<start>";
stopToken = "<stop>";
documents = preprocessText(textData,startToken,stopToken);
```

Create a word encoding object from the tokenized documents.

```
enc = wordEncoding(documents);
```

When training a deep learning model, the input data must be a numeric array containing sequences of a fixed length. Because the documents have different lengths, you must pad the shorter sequences with a padding value.

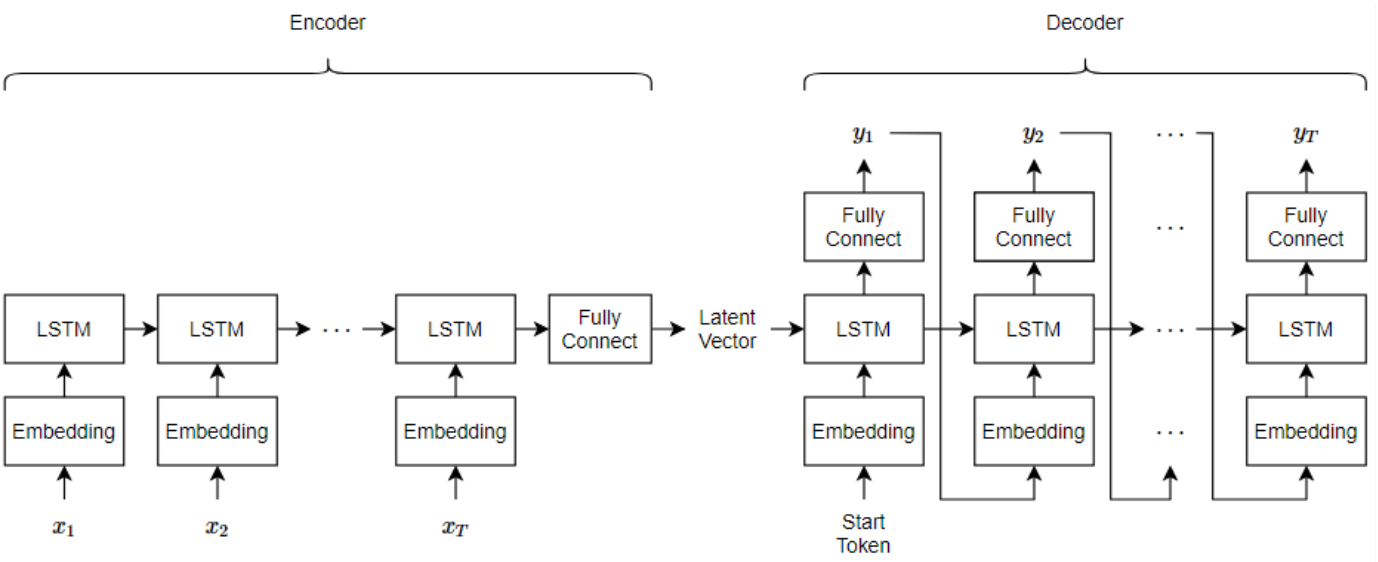
Recreate the word encoding to also include a padding token and determine the index of that token.

```
paddingToken = "<pad>";
newVocabulary = [enc.Vocabulary paddingToken];
enc = wordEncoding(newVocabulary);
paddingIdx = word2ind(enc,paddingToken)
```

```
paddingIdx = 3595
```

Initialize Model Parameters

Initialize the parameters for the following model.



Here, T is the sequence length, $x_1, \cdots, x_T$ is the input sequence of word indices, and $y_1, \cdots, y_T$ is the reconstructed sequence.

The encoder maps sequences of word indices to a latent vector by converting the input to sequences of word vectors using an embedding, inputting the word vector sequences into an LSTM operation, and applying a fully connected operation to the last time step of the LSTM output. The decoder reconstructs the input using an LSTM initialized the encoder output. For each time step, the decoder predicts the next time step and uses the output for the next time-step predictions. Both the encoder and the decoder use the same embedding.

Specify the dimensions of the parameters.

```
embeddingDimension = 100;
numHiddenUnits = 150;
latentDimension = 75;
vocabularySize = enc.NumWords;
```

Create a struct for the parameters.

```
parameters = struct;
```

Initialize the weights of the embedding using the Gaussian using the `initializeGaussian` function which is attached to this example as a supporting file. Specify a mean of 0 and a standard deviation of 0.01. To learn more, see “Gaussian Initialization” (Deep Learning Toolbox).

```
mu = 0;
sigma = 0.01;
parameters.emb.Weights = initializeGaussian([embeddingDimension vocabularySize],mu,sigma);
```

Initialize the learnable parameters for the encoder LSTM operation:

- Initialize the input weights with the Glorot initializer using the `initializeGlorot` function which is attached to this example as a supporting file. To learn more, see “Glorot Initialization” (Deep Learning Toolbox).
- Initialize the recurrent weights with the orthogonal initializer using the `initializeOrthogonal` function which is attached to this example as a supporting file. To learn more, see “Orthogonal Initialization” (Deep Learning Toolbox).
- Initialize the bias with the unit forget gate initializer using the `initializeUnitForgetGate` function which is attached to this example as a supporting file. To learn more, see “Unit Forget Gate Initialization” (Deep Learning Toolbox).

```
sz = [4*numHiddenUnits embeddingDimension];
numOut = 4*numHiddenUnits;
numIn = embeddingDimension;

parameters.lstmEncoder.InputWeights = initializeGlorot(sz,numOut,numIn);
parameters.lstmEncoder.RecurrentWeights = initializeOrthogonal([4*numHiddenUnits numHiddenUnits]
parameters.lstmEncoder.Bias = initializeUnitForgetGate(numHiddenUnits);
```

Initialize the learnable parameters for the encoder fully connected operation:

- Initialize the weights with the Glorot initializer.
- Initialize the bias with zeros using the `initializeZeros` function which is attached to this example as a supporting file. To learn more, see “Zeros Initialization” (Deep Learning Toolbox).

```
sz = [latentDimension numHiddenUnits];
numOut = latentDimension;
numIn = numHiddenUnits;

parameters.fcEncoder.Weights = initializeGlorot(sz,numOut,numIn);
parameters.fcEncoder.Bias = initializeZeros([latentDimension 1]);
```

Initialize the learnable parameters for the decoder LSTM operation:

- Initialize the input weights with the Glorot initializer.
- Initialize the recurrent weights with the orthogonal initializer.
- Initialize the bias with the unit forget gate initializer.

```
sz = [4*latentDimension embeddingDimension];
numOut = 4*latentDimension;
numIn = embeddingDimension;

parameters.lstmDecoder.InputWeights = initializeGlorot(sz,numOut,numIn);
parameters.lstmDecoder.RecurrentWeights = initializeOrthogonal([4*latentDimension latentDimensio
parameters.lstmDecoder.Bias = initializeZeros([4*latentDimension 1]);
```

Initialize the learnable parameters for the decoder fully connected operation:

- Initialize the weights with the Glorot initializer.

- Initialize the bias with zeros.

```
sz = [vocabularySize latentDimension];
numOut = vocabularySize;
numIn = latentDimension;

parameters.fcDecoder.Weights = initializeGlorot(sz,numOut,numIn);
parameters.fcDecoder.Bias = initializeZeros([vocabularySize 1]);
```

To learn more about weight initialization, see “Initialize Learnable Parameters for Model Functions” (Deep Learning Toolbox).

Define Model Encoder Function

Create the function `modelEncoder`, listed in the Encoder Model Function on page 2-0 section of the example, that computes the output of the encoder model. The `modelEncoder` function, takes as input sequences of word indices, the model parameters, and the sequence lengths, and returns the corresponding latent feature vector. To learn more about defining a model encoder function, see “Define Text Encoder Model Function” (Deep Learning Toolbox).

Define Model Decoder Function

Create the function `modelDecoder`, listed in the Decoder Model Function on page 2-0 section of the example, that computes the output of the decoder model. The `modelDecoder` function, takes as input sequences of word indices, the model parameters, and the sequence lengths, and returns the corresponding latent feature vector. To learn more about defining a model decoder function, see “Define Text Decoder Model Function” (Deep Learning Toolbox).

Define Model Gradients Function

The `modelGradients` function, listed in the Model Gradients Function on page 2-0 section of the example, takes as input the model learnable parameters, the input data `dlX`, and a vector of sequence lengths for masking, and returns the gradients of the loss with respect to the learnable parameters and the corresponding loss. To learn more about defining a model gradients function, see “Define Model Gradients Function for Custom Training Loop” (Deep Learning Toolbox).

Specify Training Options

Specify the options for training.

Train for 100 epochs with a mini-batch size of 128.

```
miniBatchSize = 128;
numEpochs = 100;
```

Train with a learning rate of 0.01.

```
learnRate = 0.01;
```

Display the training progress in a plot.

```
plots = "training-progress";
```

Train on a GPU if one is available. Using a GPU requires Parallel Computing Toolbox™ and a CUDA® enabled NVIDIA® GPU with compute capability 3.0 or higher.

```
executionEnvironment = "auto";
```

Train Network

Train the network using a custom training loop.

Initialize the parameters for the Adam optimizer.

```
trailingAvg = [];
trailingAvgSq = [];
```

Initialize the training progress plot. Create an animated line that plots the loss against the corresponding iteration.

```
if plots == "training-progress"
    figure
    lineLossTrain = animatedline('Color',[0.85 0.325 0.098]);
    xlabel("Iteration")
    ylabel("Loss")
    ylim([0 inf])
    grid on
end
```

Train the model. For the first epoch, shuffle the data and loop over mini-batches of data.

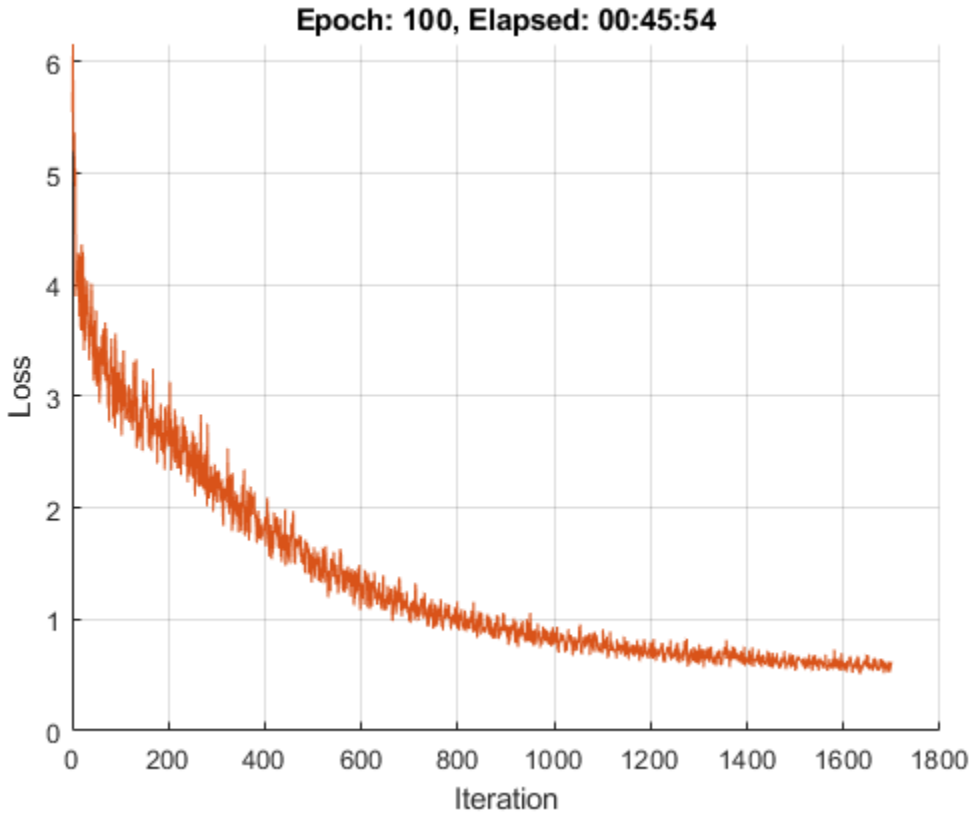
For each mini-batch:

- Convert the text data to sequences of word indices.
- Convert the data to `dlarray`.
- For GPU training, convert the data to `gpuArray` objects.
- Compute loss and gradients.
- Update the learnable parameters using the `adamupdate` function.
- Update the training progress plot.

Training can take some time to run.

```
numObservations = numel(documents);
numIterationsPerEpoch = floor(numObservations / miniBatchSize);

iteration = 0;
start = tic;

for epoch = 1:numEpochs

    % Shuffle.
    idx = randperm(numObservations);
    documents = documents(idx);

    for i = 1:numIterationsPerEpoch
        iteration = iteration + 1;

        % Read mini-batch.
        idx = (i-1)*miniBatchSize+1:i*miniBatchSize;
        documentsBatch = documents(idx);

        % Convert to sequences.
        X = doc2sequence(enc,documentsBatch, ...
            'PaddingDirection','right', ...
```

```
            'PaddingValue',paddingIdx);
        X = cat(1,X{:});

        % Convert to dlarray.
        dlX = dlarray(X,'BTC');

        % If training on a GPU, then convert data to gpuArray.
        if (executionEnvironment == "auto" && canUseGPU) || executionEnvironment == "gpu"
            dlX = gpuArray(dlX);
        end

        % Calculate sequence lengths.
        sequenceLengths = doclength(documentsBatch);

        % Evaluate model gradients.
        [gradients,loss] = dlfeval(@modelGradients, parameters, dlX, sequenceLengths);

        % Update learnable parameters.
        [parameters,trailingAvg,trailingAvgSq] = adamupdate(parameters,gradients, ...
            trailingAvg,trailingAvgSq,iteration,learnRate);

        % Display the training progress.
        if plots == "training-progress"
            D = duration(0,0,toc(start),'Format','hh:mm:ss');
            addpoints(lineLossTrain,iteration,double(gather(extractdata(loss))))
            title("Epoch: " + epoch + ", Elapsed: " + string(D))

            drawnow
        end
    end
end
```



Generate Text

Generate text using closed loop generation by initializing the decoder with different random states. Closed loop generation is when the model generates data one time-step at a time and uses the previous prediction as input for the next prediction.

Specify to generate 3 sequences of length 16.

```
numGenerations = 3;
sequenceLength = 16;
```

Create an array of random values to initialize the decoder state.

```
dlZ = dlarray(randn(latentDimension,numGenerations),'CB');
```

If predicting on a GPU, then convert data to `gpuArray`.

```
if (executionEnvironment == "auto" && canUseGPU) || executionEnvironment == "gpu"
    dlZ = gpuArray(dlZ);
end
```

Make predictions using the `modelPredictions` function, listed at the end of the example. The `modelPredictions` function returns the output scores of the decoder given the model parameters, decoder initial state, maximum sequence length, word encoding, start token, and mini-batch size.

```
dlY = modelDecoderPredictions(parameters,dlZ,sequenceLength,enc,startToken,miniBatchSize);
```

Find the word indices with the highest scores.

```
[~,idx] = max(dlY,[],1);
idx = squeeze(idx);
```

Convert the numeric indices to words and join them using the `join` function.

```
strGenerated = join(enc.Vocabulary(idx));
```

Extract the text before the first stop token using the `extractBefore` function. To prevent the function from returning missing when there are no stop tokens, append a stop token to the end of each sequence.

```
strGenerated = extractBefore(strGenerated+stopToken,stopToken);
```

Remove padding tokens.

```
strGenerated = erase(strGenerated,paddingToken);
```

The generation process introduces whitespace characters between each prediction, which means that some punctuation characters appear with unnecessary spaces before and after. Reconstruct the generated text by removing the spaces before and after the appropriate punctuation characters.

Remove the spaces that appear before the specified punctuation characters.

```
punctuationCharacters = ["." "," "’" ")" ":" ";" "?" "!"];
strGenerated = replace(strGenerated," " + punctuationCharacters,punctuationCharacters);
```

Remove the spaces that appear after the specified punctuation characters.

```
punctuationCharacters = ["(" "‘"];
strGenerated = replace(strGenerated,punctuationCharacters + " ",punctuationCharacters);
```

Remove leading and trailing white space using the `strip` function and view the generated text.

```
strGenerated = strip(strGenerated)
```

```
strGenerated = 3×1 string
    "love's thou rest light best ill mistake show seeing farther cross enough by me"
    "as before his bending sickle's compass come look find."
    "summer's lays? truth once lead mayst take,"
```

Encoder Model Function

The `modelEncoder` function, takes as input the model parameters, sequences of word indices, and the sequence lengths, and returns the corresponding latent feature vector.

Because the input data contains padded sequences of different lengths, the padding can have adverse effects on loss calculations. For the LSTM operation, instead of returning the output of the last time step of the sequence (which likely corresponds to the LSTM state after processing lots of padding values), determine the actual last time step given by the `sequenceLengths` input.

```
function dlZ = modelEncoder(parameters,dlX,sequenceLengths)

% Embedding.
weights = parameters.emb.Weights;
dlZ = embedding(dlX,weights);

% LSTM.
```

```
inputWeights = parameters.lstmEncoder.InputWeights;
recurrentWeights = parameters.lstmEncoder.RecurrentWeights;
bias = parameters.lstmEncoder.Bias;

numHiddenUnits = size(recurrentWeights,2);
hiddenState = zeros(numHiddenUnits,1,'like',dlX);
cellState = zeros(numHiddenUnits,1,'like',dlX);

dlZ1 = lstm(dlZ,hiddenState,cellState,inputWeights,recurrentWeights,bias,'DataFormat','CBT');

% Output mode 'last' with masking.
miniBatchSize = size(dlZ1,2);
dlZ = zeros(numHiddenUnits,miniBatchSize,'like',dlZ1);

for n = 1:miniBatchSize
    t = sequenceLengths(n);
    dlZ(:,n) = dlZ1(:,n,t);
end

% Fully connect.
weights = parameters.fcEncoder.Weights;
bias = parameters.fcEncoder.Bias;
dlZ = fullyconnect(dlZ,weights,bias,'DataFormat','CB');

end
```

Decoder Model Function

The `modelDecoder` function, takes as input the model parameters, sequences of word indices, and the network state, and returns the decoded sequences.

Because the `lstm` function is *stateful* (when given a time series as input, the function propagates and updates the state between each time step) and that the `embedding` and `fullyconnect` functions are time-distributed by default (when given a time series as input, the functions operate on each time step independently), the `modelDecoder` function supports both sequence and single time-step inputs.

```
function [dlY,state] = modelDecoder(parameters,dlX,state)

% Embedding.
weights = parameters.emb.Weights;
dlX = embedding(dlX,weights);

% LSTM.
inputWeights = parameters.lstmDecoder.InputWeights;
recurrentWeights = parameters.lstmDecoder.RecurrentWeights;
bias = parameters.lstmDecoder.Bias;

hiddenState = state.HiddenState;
cellState = state.CellState;

[dlY,hiddenState,cellState] = lstm(dlX,hiddenState,cellState, ...
    inputWeights,recurrentWeights,bias,'DataFormat','CBT');

state.HiddenState = hiddenState;
state.CellState = cellState;

% Fully connect.
```

```
weights = parameters.fcDecoder.Weights;
bias = parameters.fcDecoder.Bias;
dlY = fullyconnect(dlY,weights,bias,'DataFormat','CBT');

% Softmax.
dlY = softmax(dlY,'DataFormat','CBT');

end
```

Model Gradients Function

The `modelGradients` function that takes as input the model learnable parameters, the input data `dlX`, and a vector of sequence lengths for masking, and returns the gradients of the loss with respect to the learnable parameters and the corresponding loss.

To calculate the masked loss, the model gradients function uses the `maskedCrossEntropy` loss function, listed at the end of the example. To train the decoder to predict the next time-step of the sequence, specify the targets to be the input sequences shifted by one time-step.

To learn more about defining a model gradients function, see “Define Model Gradients Function for Custom Training Loop” (Deep Learning Toolbox).

```
function [gradients, loss] = modelGradients(parameters,dlX,sequenceLengths)

% Model encoder.
dlZ = modelEncoder(parameters,dlX,sequenceLengths);

% Initialize LSTM state.
state = struct;
state.HiddenState = dlZ;
state.CellState = zeros(size(dlZ),'like',dlZ);

% Teacher forcing.
dlY = modelDecoder(parameters,dlX,state);

% Loss.
dlYPred = dlY(:,:,1:end-1);
dlT = dlX(:,:,2:end);
loss = mean(maskedCrossEntropy(dlYPred,dlT,sequenceLengths));

% Gradients.
gradients = dlgradient(loss,parameters);

% Normalize loss for plotting.
sequenceLength = size(dlX,3);
loss = loss / sequenceLength;

end
```

Model Predictions Function

The `modelPredictions` function returns the output scores of the decoder given the model parameters, decoder initial state, maximum sequence length, word encoding, start token, and mini-batch size.

```
function dlY = modelDecoderPredictions(parameters,dlZ,maxLength,enc,startToken,miniBatchSize)

numObservations = size(dlZ,2);
```

```
numIterations = ceil(numObservations / miniBatchSize);

startTokenIdx = word2ind(enc,startToken);
vocabularySize = enc.NumWords;

dlY = zeros(vocabularySize,numObservations,maxLength,'like',dlZ);

% Loop over mini-batches.
for i = 1:numIterations
    idxMiniBatch = (i-1)*miniBatchSize+1:min(i*miniBatchSize,numObservations);
    miniBatchSize = numel(idxMiniBatch);

    % Initialize state.
    state = struct;
    state.HiddenState = dlZ(:,idxMiniBatch);
    state.CellState = zeros(size(dlZ(:,idxMiniBatch)),'like',dlZ);

    % Initialize decoder input.
    decoderInput = dlarray(repmat(startTokenIdx,[1 miniBatchSize]),'CBT');

    % Loop over time steps.
    for t = 1:maxLength
        % Predict next time step.
        [dlY(:,idxMiniBatch,t), state] = modelDecoder(parameters,decoderInput,state);

        % Closed loop generation.
        [~,idx] = max(dlY(:,idxMiniBatch,t));
        decoderInput = idx;
    end
end

end
```

Masked Cross Entropy Loss Function

The `maskedCrossEntropy` function calculates the loss between the specified input sequences and target sequences ignoring any time steps containing padding using the specified vector of sequence lengths.

```
function maskedLoss = maskedCrossEntropy(dlY,T,sequenceLengths)

numClasses = size(dlY,1);
miniBatchSize = size(dlY,2);
sequenceLength = size(dlY,3);

maskedLoss = zeros(sequenceLength,miniBatchSize,'like',dlY);

for t = 1:sequenceLength
    T1 = single(oneHot(T(:,:,t),numClasses));

    mask = (t<=sequenceLengths)';

    maskedLoss(t,:) = mask .* crossentropy(dlY(:,:,t),T1,'DataFormat','CBT');
end

maskedLoss = sum(maskedLoss,1);

end
```

Text Preprocessing Function

The function `preprocessText` performs these steps:

1. Prepends and appends each input string with the specified start and stop tokens, respectively.
2. Tokenize the text using `tokenizedDocument`.

```
function documents = preprocessText(textData,startToken,stopToken)

% Add start and stop tokens.
textData = startToken + textData + stopToken;

% Tokenize the text.
documents = tokenizedDocument(textData,'CustomTokens',[startToken stopToken]);

end
```

Embedding Function

The `embedding` function maps sequences of indices to vectors using the given weights.

```
function Z = embedding(X, weights)

% Reshape inputs into a vector.
[N, T] = size(X, 2:3);
X = reshape(X, N*T, 1);

% Index into embedding matrix.
Z = weights(:, X);

% Reshape outputs by separating out batch and sequence dimensions.
Z = reshape(Z, [], N, T);

end
```

One-Hot Encoding Function

The `oneHot` function converts an array of numeric indices to one-hot encoded vectors.

```
function oh = oneHot(idx, outputSize)

miniBatchSize = numel(idx);
oh = zeros(outputSize,miniBatchSize);

for n = 1:miniBatchSize
    c = idx(n);
    oh(c,n) = 1;
end

end
```

See Also

`doc2sequence` | `tokenizedDocument` | `word2ind` | `wordEncoding`

More About

- “Sequence-to-Sequence Translation Using Attention” on page 2-114
- “Generate Text Using Deep Learning” (Deep Learning Toolbox)
- “Define Text Encoder Model Function” on page 2-161
- “Define Text Decoder Model Function” on page 2-168
- “Define Custom Training Loops, Loss Functions, and Networks” (Deep Learning Toolbox)

Define Text Encoder Model Function

This example shows how to define a text encoder model function.

In the context of deep learning, an encoder is the part of a deep learning network that maps the input to some latent space. You can use these vectors for various tasks. For example,

- Classification by applying a softmax operation to the encoded data and using cross entropy loss.
- Sequence-to-sequence translation by using the encoded vector as a context vector.

Load Data

The file `sonnets.txt` contains all of Shakespeare's sonnets in a single text file.

Read the Shakespeare's Sonnets data from the file `"sonnets.txt"`.

```
filename = "sonnets.txt";
textData = fileread(filename);
```

The sonnets are indented by two whitespace characters. Remove the indentations using `replace` and split the text into separate lines using the `split` function. Remove the header from the first nine elements and the short sonnet titles.

```
textData = replace(textData,"  ","");
textData = split(textData,newline);
textData(1:9) = [];
textData(strlength(textData)<5) = [];
```

Prepare Data

Create a function that tokenizes and preprocesses the text data. The function `preprocessText`, listed at the end of the example, performs these steps:

1. Prepends and appends each input string with the specified start and stop tokens, respectively.
2. Tokenize the text using `tokenizedDocument`.

Preprocess the text data and specify the start and stop tokens `"<start>"` and `"<stop>"`, respectively.

```
startToken = "<start>";
stopToken = "<stop>";
documents = preprocessText(textData,startToken,stopToken);
```

Create a word encoding object from the tokenized documents.

```
enc = wordEncoding(documents);
```

When training a deep learning model, the input data must be a numeric array containing sequences of a fixed length. Because the documents have different lengths, you must pad the shorter sequences with a padding value.

Recreate the word encoding to also include a padding token and determine the index of that token.

```
paddingToken = "<pad>";
newVocabulary = [enc.Vocabulary paddingToken];
enc = wordEncoding(newVocabulary);
paddingIdx = word2ind(enc,paddingToken)
```

```
paddingIdx = 3595
```

Initialize Model Parameters

The goal of the encoder is to map sequences of word indices to vectors in some latent space.

Initialize the parameters for the following model.

This model uses three operations:

- The embedding maps word indices in the range 1 though `vocabularySize` to vectors of dimension `embeddingDimension`, where `vocabularySize` is the number of words in the encoding vocabulary and `embeddingDimension` is the number of components learned by the embedding.
- The LSTM operation takes as input sequences of word vectors and outputs 1-by-`numHiddenUnits` vectors, where `numHiddenUnits` is the number of hidden units in the LSTM operation.
- The fully connected operation multiplies the input by a weight matrix adding bias and outputs vectors of size `latentDimension`, where `latentDimension` is the dimension of the latent space.

Specify the dimensions of the parameters.

```
embeddingDimension = 100;
numHiddenUnits = 150;
latentDimension = 50;
vocabularySize = enc.NumWords;
```

Create a struct for the parameters.

```
parameters = struct;
```

Initialize the weights of the embedding using the Gaussian using the `initializeGaussian` function which is attached to this example as a supporting file. Specify a mean of 0 and a standard deviation of 0.01. To learn more, see “Gaussian Initialization” (Deep Learning Toolbox).

```
mu = 0;
sigma = 0.01;
parameters.emb.Weights = initializeGaussian([embeddingDimension vocabularySize],mu,sigma);
```

Initialize the learnable parameters for the encoder LSTM operation:

- Initialize the input weights with the Glorot initializer using the `initializeGlorot` function which is attached to this example as a supporting file. To learn more, see “Glorot Initialization” (Deep Learning Toolbox).
- Initialize the recurrent weights with the orthogonal initializer using the `initializeOrthogonal` function which is attached to this example as a supporting file. To learn more, see “Orthogonal Initialization” (Deep Learning Toolbox).

- Initialize the bias with the unit forget gate initializer using the `initializeUnitForgetGate` function which is attached to this example as a supporting file. To learn more, see "Unit Forget Gate Initialization" (Deep Learning Toolbox).

The sizes of the learnable parameters depend on the size of the input. Because the inputs to the LSTM operation are sequences of word vectors from the embedding operation, the number of input channels is `embeddingDimension`.

- The input weight matrix has size `4*numHiddenUnits`-by-`inputSize`, where `inputSize` is the dimension of the input data.
- The recurrent weight matrix has size `4*numHiddenUnits`-by-`numHiddenUnits`.
- The bias vector has size `4*numHiddenUnits`-by-1.

```
sz = [4*numHiddenUnits embeddingDimension];
numOut = 4*numHiddenUnits;
numIn = embeddingDimension;

parameters.lstmEncoder.InputWeights = initializeGlorot(sz,numOut,numIn);
parameters.lstmEncoder.RecurrentWeights = initializeOrthogonal([4*numHiddenUnits numHiddenUnits])
parameters.lstmEncoder.Bias = initializeUnitForgetGate(numHiddenUnits);
```

Initialize the learnable parameters for the encoder fully connected operation:

- Initialize the weights with the Glorot initializer.
- Initialize the bias with zeros using the `initializeZeros` function which is attached to this example as a supporting file. To learn more, see "Zeros Initialization" (Deep Learning Toolbox).

The sizes of the learnable parameters depend on the size of the input. Because the inputs to the fully connected operation are the outputs of the LSTM operation, the number of input channels is `numHiddenUnits`. To make the fully connected operation output vectors with size `latentDimension`, specify an output size of `latentDimension`.

- The weights matrix has size `outputSize`-by-`inputSize`, where `outputSize` and `inputSize` correspond to the output and input dimensions, respectively.
- The bias vector has size `outputSize`-by-1.

```
sz = [latentDimension numHiddenUnits];
numOut = latentDimension;
numIn = numHiddenUnits;

parameters.fcEncoder.Weights = initializeGlorot(sz,numOut,numIn);
parameters.fcEncoder.Bias = initializeZeros([latentDimension 1]);
```

Define Model Encoder Function

Create the function `modelEncoder`, listed in the Encoder Model Function on page 2-0 section of the example, that computes the output of the encoder model. The `modelEncoder` function, takes as input sequences of word indices, the model parameters, and the sequence lengths, and returns the corresponding latent feature vector.

Prepare Mini-Batch of Data

To train the model using a custom training loop, you must iterate over mini-batches of data and convert it into the format required for the encoder model and the model gradients functions. This

section of the example illustrates the steps needed for preparing a mini-batch of data inside the custom training loop.

Prepare an example mini-batch of data. Select a mini-batch of 32 documents from `documents`. This represents the mini-batch of data used in an iteration of a custom training loop.

```
miniBatchSize = 32;
idx = 1:miniBatchSize;
documentsBatch = documents(idx);
```

Convert the documents to sequences using the `doc2sequence` function and specify to right-pad the sequences with the word index corresponding to the padding token.

```
X = doc2sequence(enc,documentsBatch, ...
    'PaddingDirection','right', ...
    'PaddingValue',paddingIdx);
```

The output of the `doc2sequence` function is a cell array, where each element is a row vector of word indices. Because the encoder model function requires numeric input, concatenate the rows of the data using the `cat` function and specify to concatenate along the first dimension. The output has size `miniBatchSize`-by-`sequenceLength`, where `sequenceLength` is the length of the longest sequence in the mini-batch.

```
X = cat(1,X{:});
size(X)
```

```
ans = 1×2

    32    14
```

Convert the data to a `dlarray` with format `'BTC'` (batch, time, channel). The software automatically rearranges the output to have format `'CTB'` so the output has size `1`-by-`miniBatchSize`-by-`sequenceLength`.

```
dlX = dlarray(X,'BTC');
size(dlX)
```

```
ans = 1×3

     1    32    14
```

For masking, calculate the unpadded sequence lengths of the input data using the `doclength` function with the mini-batch of documents as input.

```
sequenceLengths = doclength(documentsBatch);
```

This code snippet shows an example of preparing a mini-batch in a custom training loop.

```
iteration = 0;

% Loop over epochs.
for epoch = 1:numEpochs

    % Loop over mini-batches.
    for i = 1:numIterationsPerEpoch
```

```
        iteration = iteration + 1;

        % Read mini-batch.
        idx = (i-1)*miniBatchSize+1:i*miniBatchSize;
        documentsBatch = documents(idx);

        % Convert to sequences.
        X = doc2sequence(enc,documentsBatch, ...
            'PaddingDirection','right', ...
            'PaddingValue',paddingIdx);
        X = cat(1,X{:});

        % Convert to dlarray.
        dlX = dlarray(X,'BTC');

        % Calculate sequence lengths.
        sequenceLengths = doclength(documentsBatch);

        % Evaluate model gradients.
        % ...

        % Update learnable parameters.
        % ...
    end
end
```

Use Model Function in Model Gradients Function

When training a deep learning model with a custom training loop, you must calculate the gradients of the loss with respect to the learnable parameters. This calculation depends on the output of a forward pass of the model function.

To perform a forward pass of the encoder, use the `modelEncoder` function directly with the parameters, data, and sequence lengths as input. The output is a `latentDimension`-by-`miniBatchSize` matrix.

```
dlZ = modelEncoder(parameters,dlX,sequenceLengths);
size(dlZ)
```

```
ans = 1×2

    50    32
```

This code snippet shows an example of using a model encoder function inside the model gradients function.

```
function gradients = modelGradients(parameters,dlX,sequenceLengths)

    dlZ = modelEncoder(parameters,dlX,sequenceLengths);

    % Calculate loss.
    % ...

    % Calculate gradients.
    % ...

end
```

This code snippet shows an example of evaluating the model gradients in a custom training loop.

```
iteration = 0;

% Loop over epochs.
for epoch = 1:numEpochs

    % Loop over mini-batches.
    for i = 1:numIterationsPerEpoch
        iteration = iteration + 1;

        % Prepare mini-batch.
        % ...

        % Evaluate model gradients.
        gradients = dlfeval(@modelGradients, parameters, dlX, sequenceLengths);

        % Update learnable parameters.
        [parameters,trailingAvg,trailingAvgSq] = adamupdate(parameters,gradients, ...
            trailingAvg,trailingAvgSq,iteration);
    end
end
```

Encoder Model Function

The `modelEncoder` function, takes as input the model parameters, sequences of word indices, and the sequence lengths, and returns the corresponding latent feature vector.

Because the input data contains padded sequences of different lengths, the padding can have adverse effects on loss calculations. For the LSTM operation, instead of returning the output of the last time step of the sequence (which likely corresponds to the LSTM state after processing lots of padding values), determine the actual last time step given by the `sequenceLengths` input.

```
function dlZ = modelEncoder(parameters,dlX,sequenceLengths)

% Embedding.
weights = parameters.emb.Weights;
dlZ = embed(dlX,weights);

% LSTM.
inputWeights = parameters.lstmEncoder.InputWeights;
recurrentWeights = parameters.lstmEncoder.RecurrentWeights;
bias = parameters.lstmEncoder.Bias;

numHiddenUnits = size(recurrentWeights,2);
hiddenState = zeros(numHiddenUnits,1,'like',dlX);
cellState = zeros(numHiddenUnits,1,'like',dlX);

dlZ1 = lstm(dlZ,hiddenState,cellState,inputWeights,recurrentWeights,bias);

% Output mode 'last' with masking.
miniBatchSize = size(dlZ1,2);
dlZ = zeros(numHiddenUnits,miniBatchSize,'like',dlZ1);
dlZ = dlarray(dlZ,'CB');

for n = 1:miniBatchSize
    t = sequenceLengths(n);
    dlZ(:,n) = dlZ1(:,n,t);
```

```
end

% Fully connect.
weights = parameters.fcEncoder.Weights;
bias = parameters.fcEncoder.Bias;
dlZ = fullyconnect(dlZ,weights,bias);

end
```

Preprocessing Function

The function `preprocessText` performs these steps:

1. Prepends and appends each input string with the specified start and stop tokens, respectively.
2. Tokenize the text using `tokenizedDocument`.

```
function documents = preprocessText(textData,startToken,stopToken)

% Add start and stop tokens.
textData = startToken + textData + stopToken;

% Tokenize the text.
documents = tokenizedDocument(textData,'CustomTokens',[startToken stopToken]);

end
```

See Also

`doc2sequence` | `tokenizedDocument` | `word2ind` | `wordEncoding`

More About

- “Classify Text Data Using Deep Learning” on page 2-65
- “Classify Text Data Using Convolutional Neural Network” on page 2-73
- “Sequence-to-Sequence Translation Using Attention” on page 2-114
- “Generate Text Using Autoencoders” on page 2-148
- “Define Text Decoder Model Function” on page 2-168
- “Define Custom Training Loops, Loss Functions, and Networks” (Deep Learning Toolbox)

Define Text Decoder Model Function

This example shows how to define a text decoder model function.

In the context of deep learning, a decoder is the part of a deep learning network that maps a latent vector to some sample space. You can use decode the vectors for various tasks. For example,

- Text generation by initializing a recurrent network with the encoded vector.
- Sequence-to-sequence translation by using the encoded vector as a context vector.
- Image captioning by using the encoded vector as a context vector.

Load Data

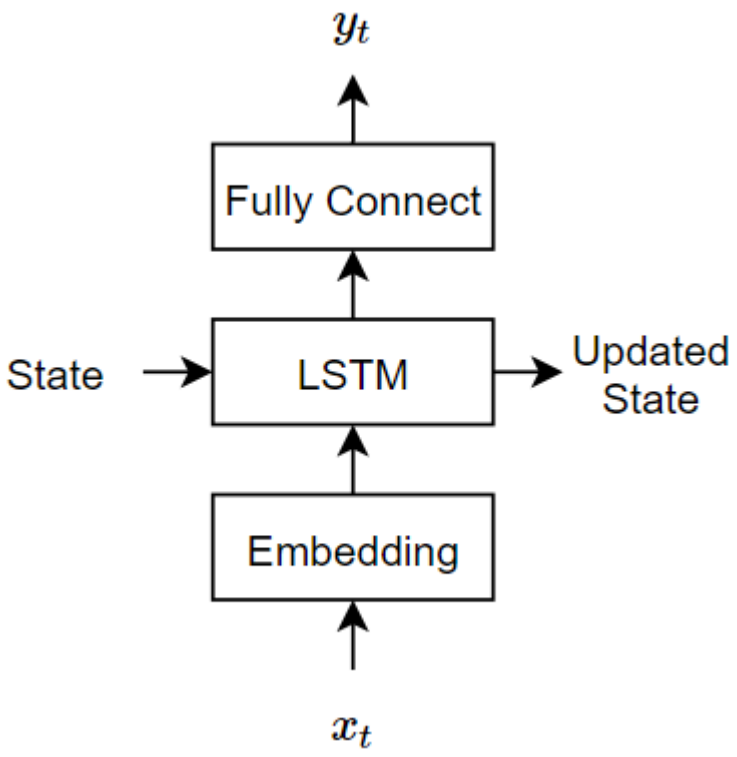
Load the encoded data from `sonnetsEncoded.mat`. This MAT file contains the word encoding, a mini-batch of sequences `dlX`, and the corresponding encoded data `dlZ` output by the encoder used in the example “Define Text Encoder Model Function” (Deep Learning Toolbox).

```
s = load("sonnetsEncoded.mat");
enc = s.enc;
dlX = s.dlX;
dlZ = s.dlZ;

[latentDimension,miniBatchSize] = size(dlZ,1:2);
```

Initialize Model Parameters

The goal of the decoder is to generate sequences given some initial input data and network state.

Initialize the parameters for the following model.



The decoder reconstructs the input using an LSTM initialized the encoder output. For each time step, the decoder predicts the next time step and uses the output for the next time-step predictions. Both the encoder and the decoder use the same embedding.

This model uses three operations:

- The embedding maps word indices in the range 1 though `vocabularySize` to vectors of dimension `embeddingDimension`, where `vocabularySize` is the number of words in the encoding vocabulary and `embeddingDimension` is the number of components learned by the embedding.
- The LSTM operation takes as input a single word vector and outputs 1-by-`numHiddenUnits` vector, where `numHiddenUnits` is the number of hidden units in the LSTM operation. The initial state of the LSTM network (the state at the first time-step) is the encoded vector, so the number of hidden units must match the latent dimension of the encoder.
- The fully connected operation multiplies the input by a weight matrix adding bias and outputs vectors of size `vocabularySize`.

Specify the dimensions of the parameters. The embedding sizes must match the encoder.

```
embeddingDimension = 100;
vocabularySize = enc.NumWords;
numHiddenUnits = latentDimension;
```

Create a struct for the parameters.

```
parameters = struct;
```

Initialize the weights of the embedding using the Gaussian using the `initializeGaussian` function which is attached to this example as a supporting file. Specify a mean of 0 and a standard deviation of 0.01. To learn more, see “Gaussian Initialization” (Deep Learning Toolbox).

```
mu = 0;
sigma = 0.01;
parameters.emb.Weights = initializeGaussian([embeddingDimension vocabularySize],mu,sigma);
```

Initialize the learnable parameters for the decoder LSTM operation:

- Initialize the input weights with the Glorot initializer using the `initializeGlorot` function which is attached to this example as a supporting file. To learn more, see “Glorot Initialization” (Deep Learning Toolbox).
- Initialize the recurrent weights with the orthogonal initializer using the `initializeOrthogonal` function which is attached to this example as a supporting file. To learn more, see “Orthogonal Initialization” (Deep Learning Toolbox).
- Initialize the bias with the unit forget gate initializer using the `initializeUnitForgetGate` function which is attached to this example as a supporting file. To learn more, see “Unit Forget Gate Initialization” (Deep Learning Toolbox).

The sizes of the learnable parameters depend on the size of the input. Because the inputs to the LSTM operation are sequences of word vectors from the embedding operation, the number of input channels is `embeddingDimension`.

- The input weight matrix has size `4*numHiddenUnits`-by-`inputSize`, where `inputSize` is the dimension of the input data.

- The recurrent weight matrix has size `4*numHiddenUnits`-by-`numHiddenUnits`.
- The bias vector has size `4*numHiddenUnits`-by-1.

```
sz = [4*numHiddenUnits embeddingDimension];
numOut = 4*numHiddenUnits;
numIn = embeddingDimension;

parameters.lstmDecoder.InputWeights = initializeGlorot(sz,numOut,numIn);
parameters.lstmDecoder.RecurrentWeights = initializeOrthogonal([4*numHiddenUnits numHiddenUnits]);
parameters.lstmDecoder.Bias = initializeUnitForgetGate(numHiddenUnits);
```

Initialize the learnable parameters for the encoder fully connected operation:

- Initialize the weights with the Glorot initializer.
- Initialize the bias with zeros using the `initializeZeros` function which is attached to this example as a supporting file. To learn more, see “Zeros Initialization” (Deep Learning Toolbox).

The sizes of the learnable parameters depend on the size of the input. Because the inputs to the fully connected operation are the outputs of the LSTM operation, the number of input channels is `numHiddenUnits`. To make the fully connected operation output vectors with size `latentDimension`, specify an output size of `latentDimension`.

- The weights matrix has size `outputSize`-by-`inputSize`, where `outputSize` and `inputSize` correspond to the output and input dimensions, respectively.
- The bias vector has size `outputSize`-by-1.

To make the fully connected operation output vectors with size `vocabularySize`, specify an output size of `vocabularySize`.

```
inputSize = numHiddenUnits;
outputSize = vocabularySize;
parameters.fcDecoder.Weights = dlarray(randn(outputSize,inputSize,'single'));
parameters.fcDecoder.Bias = dlarray(zeros(outputSize,1,'single'));
```

Define Model Decoder Function

Create the function `modelDecoder`, listed in the Decoder Model Function on page 2-0 section of the example, that computes the output of the decoder model. The `modelDecoder` function, takes as input sequences of word indices, the model parameters, and the sequence lengths, and returns the corresponding latent feature vector.

Use Model Function in Model Gradients Function

When training a deep learning model with a custom training loop, you must calculate the gradients of the loss with respect to the learnable parameters. This calculation depends on the output of a forward pass of the model function.

There are two common approaches to generating text data with a decoder:

1. Closed loop — For each time step, make predictions using the previous prediction as input.
2. Open loop — For each time step, make predictions using inputs from an external source (for example, training targets).

Closed Loop Generation

Closed loop generation is when the model generates data one time-step at a time and uses the previous prediction as input for the next prediction. Unlike open loop generation, this process does

not require any input between predictions and is best suited for scenarios without supervision. For example, a language translation model that generates output text in one go.

To use closed loop

Initialize the hidden state of the LSTM network with the encoder output `dlZ`.

```
state = struct;
state.HiddenState = dlZ;
state.CellState = zeros(size(dlZ),'like',dlZ);
```

For the first time step, use an array of start tokens as input for the decoder. For simplicity, extract an array of start tokens from the first time-step of the training data.

```
decoderInput = dlX(:,:,1);
```

Preallocate the decoder output to have size `numClasses`-by-`miniBatchSize`-by-`sequenceLength` with the same datatype as `dlX`, where `sequenceLength` is the desired length of the generation, for example, the length of the training targets. For this example, specify a sequence length of 16.

```
sequenceLength = 16;
dlY = zeros(vocabularySize,miniBatchSize,sequenceLength,'like',dlX);
dlY = dlarray(dlY,'CBT');
```

For each time step, predict the next time step of the sequence using the `modelDecoder` function. After each prediction, find the indices corresponding to the maximum values of the decoder output and use these indices as the decoder input for the next time step.

```
for t = 1:sequenceLength
    [dlY(:,:,t), state] = modelDecoder(parameters,decoderInput,state);

    [~,idx] = max(dlY(:,:,t));
    decoderInput = idx;
end
```

The output is a `vocabularySize`-by-`miniBatchSize`-by-`sequenceLength` array.

```
size(dlY)
```

```
ans = 1×3

        3595          32          16
```

This code snippet shows an example of performing closed loop generation in a model gradients function.

```
function gradients = modelGradients(parameters,dlX,sequenceLengths)

    % Encode input.
    dlZ = modelEncoder(parameters,dlX,sequenceLengths);

    % Initialize LSTM state.
    state = struct;
    state.HiddenState = dlZ;
    state.CellState = zeros(size(dlZ),'like',dlZ);

    % Initialize decoder input.
```

```
    decoderInput = dlX(:,:,1);

    % Closed loop prediction.
    sequenceLength = size(dlX,3);
    dlY = zeros(numClasses,miniBatchSize,sequenceLength,'like',dlX);
    for t = 1:sequenceLength
        [dlY(:,:,t), state] = modelDecoder(parameters,decoderInput,state);

        [~,idx] = max(dlY(:,:,t));
        decoderInput = idx;
    end

    % Calculate loss.
    % ...

    % Calculate gradients.
    % ...

end
```

Open Loop Generation: Teacher Forcing

When training with closed loop generation, predicting the most likely word for each step in the sequence can lead to suboptimal results. For example, in an image captioning workflow, if the decoder predicts the first word of a caption is "a" when given an image of an elephant, then the probability of predicting "elephant" for the next word becomes much more unlikely because of the extremely low probability of the phrase "a elephant" appearing in English text.

To help the network converge faster, you can use *teacher forcing:* use the target values as input to the decoder instead of the previous predictions. Using teacher forcing helps the network to learn characteristics from the later time steps of the sequences without having to wait for the network to correctly generate the earlier time steps of the sequences.

To perform teacher forcing, use the `modelEncoder` function directly with the target sequence as input.

Initialize the hidden state of the LSTM network with the encoder output `dlZ`.

```
state = struct;
state.HiddenState = dlZ;
state.CellState = zeros(size(dlZ),'like',dlZ);
```

Make predictions using the target sequence as input.

```
dlY = modelDecoder(parameters,dlX,state);
```

The output is a `vocabularySize`-by-`miniBatchSize`-by-`sequenceLength` array, where `sequenceLength` is the length of the input sequences.

```
size(dlY)
```

```
ans = 1×3

        3595          32          14
```

This code snippet shows an example of performing teacher forcing in a model gradients function.

```
function gradients = modelGradients(parameters,dlX,sequenceLengths)

    % Encode input.
    dlZ = modelEncoder(parameters,dlX,dlZ);

    % Initialize LSTM state.
    state = struct;
    state.HiddenState = dlZ;
    state.CellState = zeros(size(dlZ),'like',dlZ);

    % Teacher forcing.
    dlY = modelDecoder(parameters,dlX,state);

    % Calculate loss.
    % ...

    % Calculate gradients.
    % ...

end
```

Decoder Model Function

The `modelDecoder` function, takes as input the model parameters, sequences of word indices, and the network state, and returns the decoded sequences.

Because the `lstm` function is *stateful* (when given a time series as input, the function propagates and updates the state between each time step) and that the `embed` and `fullyconnect` functions are time-distributed by default (when given a time series as input, the functions operate on each time step independently), the `modelDecoder` function supports both sequence and single time-step inputs.

```
function [dlY,state] = modelDecoder(parameters,dlX,state)

% Embedding.
weights = parameters.emb.Weights;
dlX = embed(dlX,weights);

% LSTM.
inputWeights = parameters.lstmDecoder.InputWeights;
recurrentWeights = parameters.lstmDecoder.RecurrentWeights;
bias = parameters.lstmDecoder.Bias;

hiddenState = state.HiddenState;
cellState = state.CellState;

[dlY,hiddenState,cellState] = lstm(dlX,hiddenState,cellState, ...
    inputWeights,recurrentWeights,bias);

state.HiddenState = hiddenState;
state.CellState = cellState;

% Fully connect.
weights = parameters.fcDecoder.Weights;
bias = parameters.fcDecoder.Bias;
dlY = fullyconnect(dlY,weights,bias);
```

```
end
```

See Also

`doc2sequence` | `tokenizedDocument` | `word2ind` | `wordEncoding`

More About

- “Classify Text Data Using Deep Learning” on page 2-65
- “Classify Text Data Using Convolutional Neural Network” on page 2-73
- “Sequence-to-Sequence Translation Using Attention” on page 2-114
- “Generate Text Using Autoencoders” on page 2-148
- “Define Text Encoder Model Function” on page 2-161
- “Define Custom Training Loops, Loss Functions, and Networks” (Deep Learning Toolbox)

Classify Out-of-Memory Text Data Using Custom Mini-Batch Datastore

This example shows how to classify out-of-memory text data with a deep learning network using a custom mini-batch datastore.

A mini-batch datastore is an implementation of a datastore with support for reading data in batches. You can use a mini-batch datastore as a source of training, validation, test, and prediction data sets for deep learning applications. Use mini-batch datastores to read out-of-memory data or to perform specific preprocessing operations when reading batches of data.

When training the network, the software creates mini-batches of sequences of the same length by padding, truncating, or splitting the input data. The `trainingOptions` function provides options to pad and truncate input sequences, however, these options are not well suited for sequences of word vectors. Furthermore, this function does not support padding data in a custom datastore. Instead, you must pad and truncate the sequences manually. If you *left-pad* and truncate the sequences of word vectors, then the training might improve.

The “Classify Text Data Using Deep Learning” on page 2-65 example manually truncates and pads all the documents to the same length. This process adds lots of padding to very short documents and discards lots of data from very long documents.

Alternatively, to prevent adding too much padding or discarding too much data, create a custom mini-batch datastore that inputs mini-batches into the network. The custom mini-batch datastore `textDatastore.m` converts mini-batches of documents to sequences or word indices and left-pads each mini-batch to the length of the longest document in the mini-batch. For sorted data, this datastore can help reduce the amount of padding added to the data since documents are not padded to a fixed length. Similarly, the datastore does not discard any data from the documents.

This example uses the custom mini-batch datastore `textDatastore.m`. You can adapt this datastore to your data by customizing the functions. For an example showing how to create your own custom mini-batch datastore, see “Develop Custom Mini-Batch Datastore” (Deep Learning Toolbox).

Load Pretrained Word Embedding

The datastore `textDatastore` requires a word embedding to convert documents to sequences of vectors. Load a pretrained word embedding using `fastTextWordEmbedding`. This function requires Text Analytics Toolbox™ Model *for fastText English 16 Billion Token Word Embedding* support package. If this support package is not installed, then the function provides a download link.

```
emb = fastTextWordEmbedding;
```

Create Mini-Batch Datastore of Documents

Create a datastore that contains the data for training. The custom mini-batch datastore `textDatastore` reads predictors and labels from a CSV file. For the predictors, the datastore converts the documents into sequences of word indices and for the responses, the datastore returns a categorical label for each document.

To create the datastore, first save the custom mini-batch datastore `textDatastore.m` to the path. For more information about creating custom mini-batch datastores, see “Develop Custom Mini-Batch Datastore” (Deep Learning Toolbox).

For the training data, specify the CSV file `"factoryReports.csv"` and that the text and labels are in the columns `"Description"` and `"Category"` respectively.

```
filenameTrain = "factoryReports.csv";
textName = "Description";
labelName = "Category";
dsTrain = textDatastore(filenameTrain,textName,labelName,emb)
```

```
dsTrain =
  textDatastore with properties:

            ClassNames: ["Electronic Failure"    "Leak"    "Mechanical Failure"    "Software Fai
             Datastore: [1×1 matlab.io.datastore.TransformedDatastore]
    EmbeddingDimension: 300
             LabelName: "Category"
         MiniBatchSize: 128
            NumClasses: 4
       NumObservations: 480
```

Create and Train LSTM Network

Define the LSTM network architecture. To input sequence data into the network, include a sequence input layer and set the input size to the embedding dimension. Next, include an LSTM layer with 180 hidden units. To use the LSTM layer for a sequence-to-label classification problem, set the output mode to `'last'`. Finally, add a fully connected layer with output size equal to the number of classes, a softmax layer, and a classification layer.

```
numFeatures = dsTrain.EmbeddingDimension;
numHiddenUnits = 180;
numClasses = dsTrain.NumClasses;

layers = [ ...
    sequenceInputLayer(numFeatures)
    lstmLayer(numHiddenUnits,'OutputMode','last')
    fullyConnectedLayer(numClasses)
    softmaxLayer
    classificationLayer];
```

Specify the training options. Specify the solver to be `'adam'` and the gradient threshold to be 2. The datastore `textDatastore.m` does not support shuffling, so set `'Shuffle'`, to `'never'`. For an example showing how to implement a datastore with support for shuffling, see “Develop Custom Mini-Batch Datastore” (Deep Learning Toolbox). To monitor the training progress, set the `'Plots'` option to `'training-progress'`. To suppress verbose output, set `'Verbose'` to `false`.

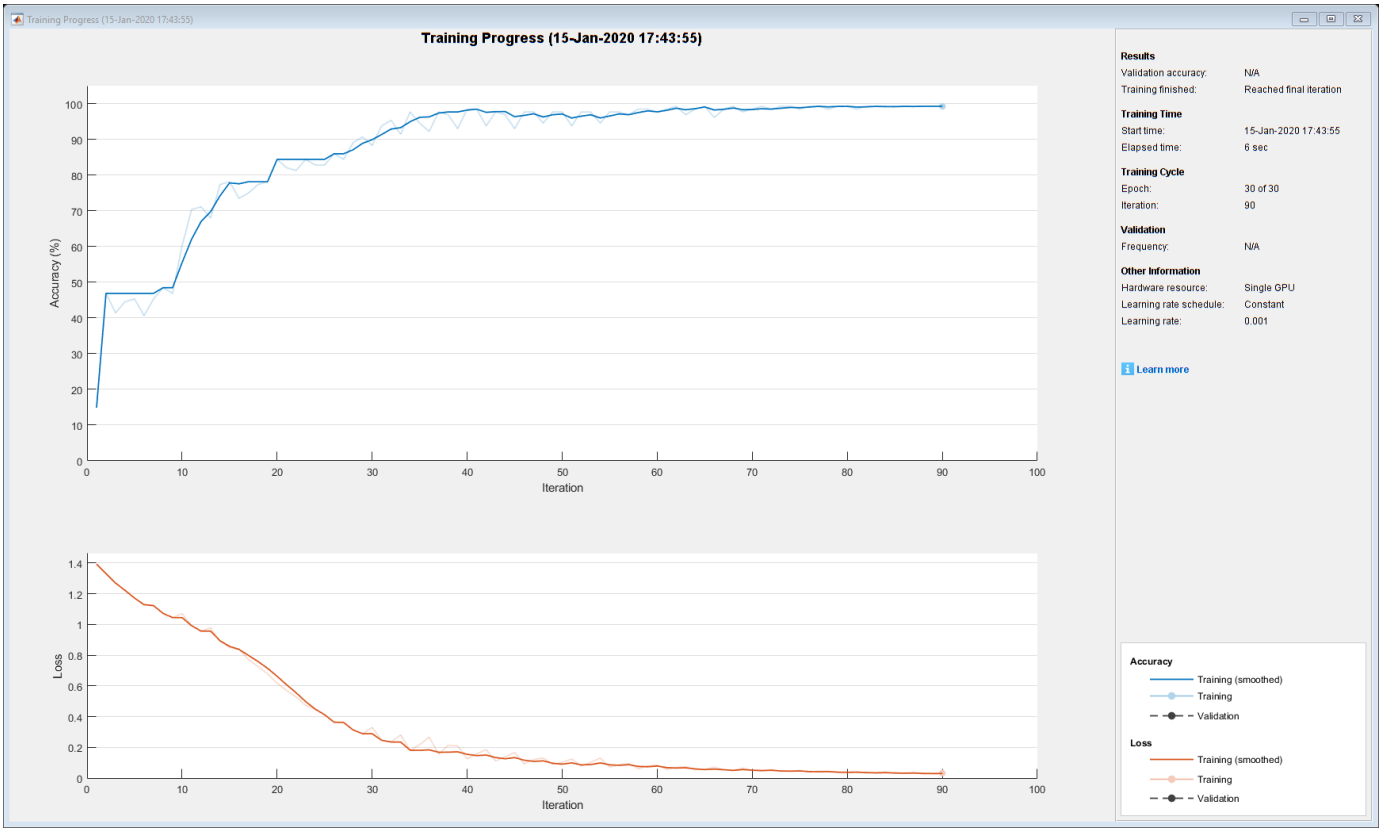
By default, `trainNetwork` uses a GPU if one is available (requires Parallel Computing Toolbox™ and a CUDA® enabled GPU with compute capability 3.0 or higher). Otherwise, it uses the CPU. To specify the execution environment manually, use the `'ExecutionEnvironment'` name-value pair argument of `trainingOptions`. Training on a CPU can take significantly longer than training on a GPU.

```
miniBatchSize = 128;
numObservations = dsTrain.NumObservations;
numIterationsPerEpoch = floor(numObservations / miniBatchSize);

options = trainingOptions('adam', ...
    'MiniBatchSize',miniBatchSize, ...
    'GradientThreshold',2, ...
```

```
    'Shuffle','never', ...
    'Plots','training-progress', ...
    'Verbose',false);
```

Train the LSTM network using the `trainNetwork` function.

```
net = trainNetwork(dsTrain,layers,options);
```



Predict Using New Data

Classify the event type of three new reports. Create a string array containing the new reports.

```
reportsNew = [
    "Coolant is pooling underneath sorter."
    "Sorter blows fuses at start up."
    "There are some very loud rattling sounds coming from the assembler."];
```

Preprocess the text data using the preprocessing steps as the datastore `textDatastore.m`.

```
documents = tokenizedDocument(reportsNew);
documents = lower(documents);
documents = erasePunctuation(documents);
predictors = doc2sequence(emb,documents);
```

Classify the new sequences using the trained LSTM network.

```
labelsNew = classify(net,predictors)
```

```
labelsNew = 3×1 categorical
     Leak
```

```
Electronic Failure
Mechanical Failure
```

See Also

`doc2sequence` | `extractHTMLText` | `findElement` | `htmlTree` | `lstmLayer` | `sequenceInputLayer` | `tokenizedDocument` | `trainNetwork` | `trainingOptions` | `wordEmbeddingLayer` | `wordcloud`

Related Examples

- “Generate Text Using Deep Learning” (Deep Learning Toolbox)
- “Create Simple Text Model for Classification” on page 2-2
- “Analyze Text Data Using Topic Models” on page 2-13
- “Analyze Text Data Using Multiword Phrases” on page 2-7
- “Train a Sentiment Classifier” on page 2-51
- “Sequence Classification Using Deep Learning” (Deep Learning Toolbox)
- “Deep Learning in MATLAB” (Deep Learning Toolbox)

Display and Presentation

Visualize Text Data Using Word Clouds

This example shows how to visualize text data using word clouds.

Text Analytics Toolbox extends the functionality of the `wordcloud` (MATLAB) function. It adds support for creating word clouds directly from string arrays and creating word clouds from bag-of-words models and LDA topics.

Load the example data. The file `factoryReports.csv` contains factory reports, including a text description and categorical labels for each event.

```
filename = "factoryReports.csv";
tbl = readtable(filename,'TextType','string');
```

Extract the text data from the `Description` column.

```
textData = tbl.Description;
textData(1:10)
```

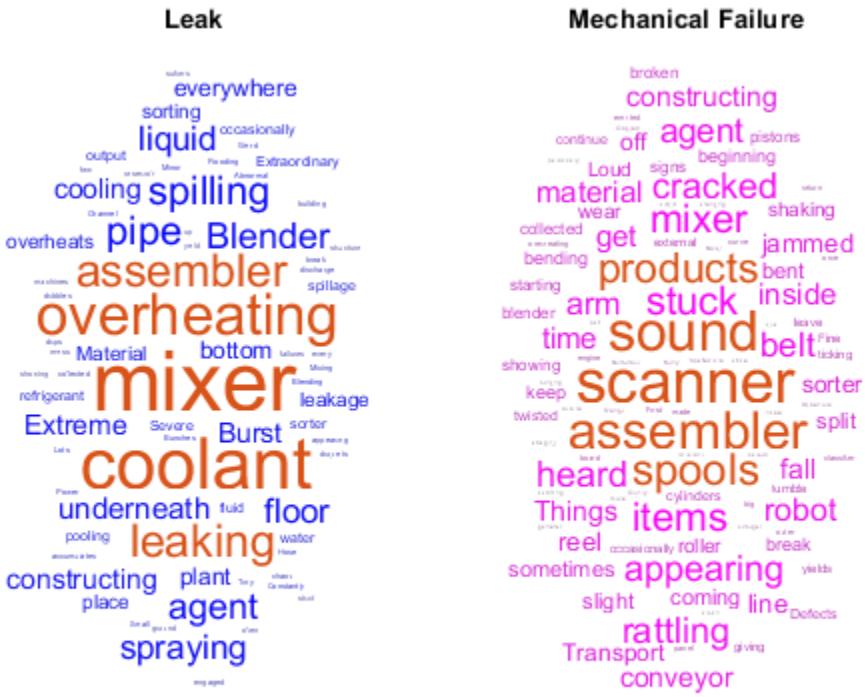
```
ans = 10x1 string
    "Items are occasionally getting stuck in the scanner spools."
    "Loud rattling and banging sounds are coming from assembler pistons."
    "There are cuts to the power when starting the plant."
    "Fried capacitors in the assembler."
    "Mixer tripped the fuses."
    "Burst pipe in the constructing agent is spraying coolant."
    "A fuse is blown in the mixer."
    "Things continue to tumble off of the belt."
    "Falling items from the conveyor belt."
    "The scanner reel is split, it will soon begin to curve."
```

Create a word cloud from the reports.

```
figure
wordcloud(textData);
title("Factory Reports")
```

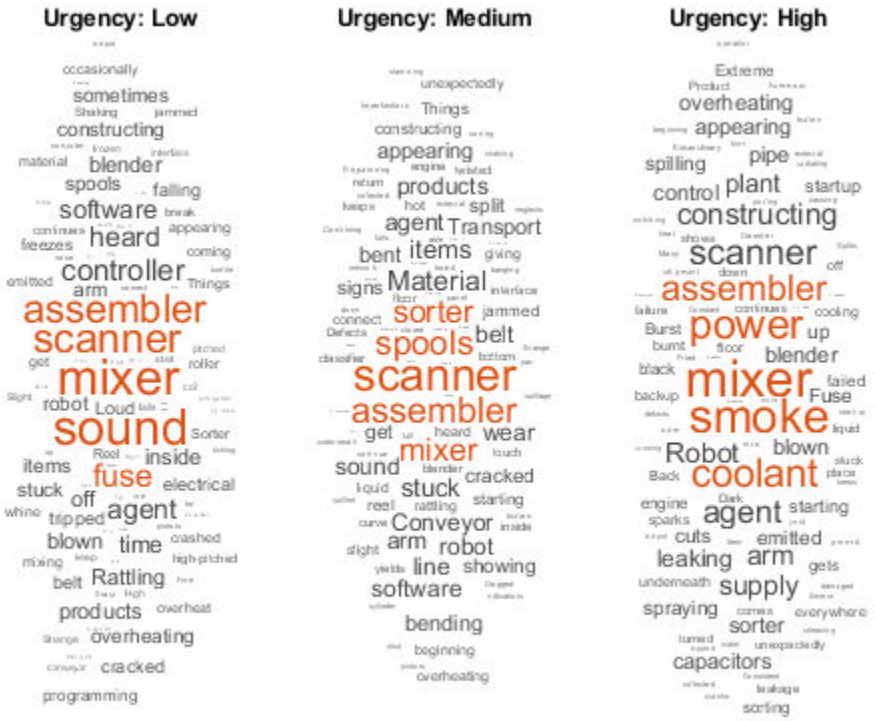
Compare the words in the reports with labels `"Leak"` and `"Mechanical Failure"`. Create word clouds of the reports for each of these labels. Specify the word colors to be blue and magenta for each word cloud respectively.

```
figure
labels = tbl.Category;

subplot(1,2,1)
idx = labels == "Leak";
wordcloud(textData(idx),'Color','blue');
title("Leak")

subplot(1,2,2)
idx = labels == "Mechanical Failure";
wordcloud(textData(idx),'Color','magenta');
title("Mechanical Failure")
```



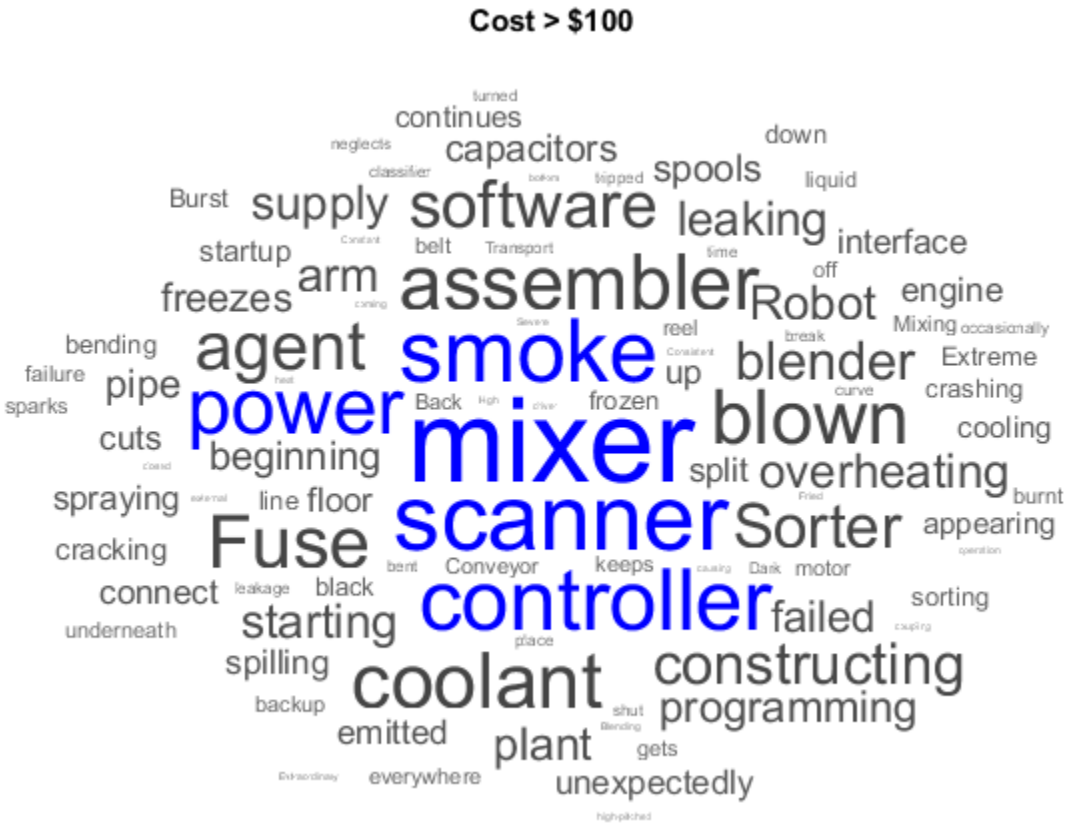
Compare the words in the reports with urgency "Low", "Medium", and "High".

```
figure
urgency = tbl.Urgency;

subplot(1,3,1)
idx = urgency == "Low";
wordcloud(textData(idx));
title("Urgency: Low")

subplot(1,3,2)
idx = urgency == "Medium";
wordcloud(textData(idx));
title("Urgency: Medium")

subplot(1,3,3)
idx = urgency == "High";
wordcloud(textData(idx));
title("Urgency: High")
```

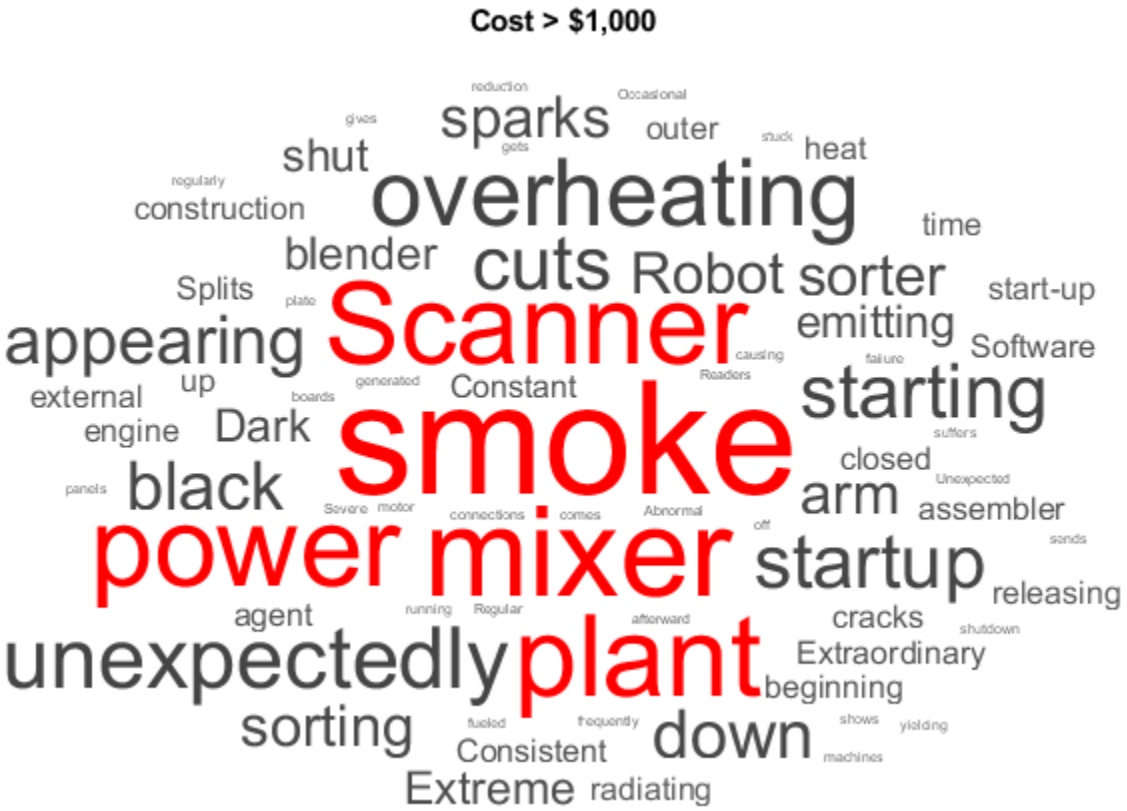


Compare the words in the reports with cost reported in hundreds of dollars to the reports with costs reported in thousands of dollars. Create word clouds of the reports for each of these amounts with highlight color blue and red respectively.

```
cost = tbl.Cost;
idx = cost > 100;
figure
wordcloud(textData(idx),'HighlightColor','blue');
title("Cost > $100")
```




```
idx = cost > 1000;
figure
wordcloud(textData(idx),'HighlightColor','red');
title("Cost > $1,000")
```



See Also

`bagOfWords` | `tokenizedDocument` | `wordcloud`

Related Examples

- “Prepare Text Data for Analysis” on page 1-10
- “Analyze Text Data Using Topic Models” on page 2-13
- “Classify Text Data Using Deep Learning” on page 2-65
- “Visualize Word Embeddings Using Text Scatter Plots” on page 3-8

Visualize Word Embeddings Using Text Scatter Plots

This example shows how to visualize word embeddings using 2-D and 3-D t-SNE and text scatter plots.

Word embeddings map words in a vocabulary to real vectors. The vectors attempt to capture the semantics of the words, so that similar words have similar vectors. Some embeddings also capture relationships between words like "Italy is to France as Rome is to Paris". In vector form, this relationship is *Italy* − *Rome* + *Paris* = *France*.

Load Pretrained Word Embedding

Load a pretrained word embedding using `fastTextWordEmbedding`. This function requires Text Analytics Toolbox™ Model *for fastText English 16 Billion Token Word Embedding* support package. If this support package is not installed, then the function provides a download link.

```
emb = fastTextWordEmbedding
```

```
emb =
  wordEmbedding with properties:

     Dimension: 300
    Vocabulary: [1×999994 string]
```

Explore the word embedding using `word2vec` and `vec2word`. Convert the words *Italy*, *Rome*, and *Paris* to vectors using `word2vec`.

```
italy = word2vec(emb,"Italy");
rome = word2vec(emb,"Rome");
paris = word2vec(emb,"Paris");
```

Compute the vector given by `italy - rome + paris`. This vector encapsulates the semantic meaning of the word *Italy*, without the semantics of the word *Rome*, and also includes the semantics of the word *Paris*.

```
vec = italy - rome + paris
```

```
vec = 1×300 single row vector

    0.1606   -0.0690    0.1183   -0.0349    0.0672    0.0907   -0.1820   -0.0080    0.0320   -0.0
```

Find the closest words in the embedding to `vec` using `vec2word`.

```
word = vec2word(emb,vec)
```

```
word =
"France"
```

Create 2-D Text Scatter Plot

Visualize the word embedding by creating a 2-D text scatter plot using `tsne` and `textscatter`.

Convert the first 5000 words to vectors using `word2vec`. `V` is a matrix of word vectors of length 300.

```
words = emb.Vocabulary(1:5000);
V = word2vec(emb,words);
size(V)
```

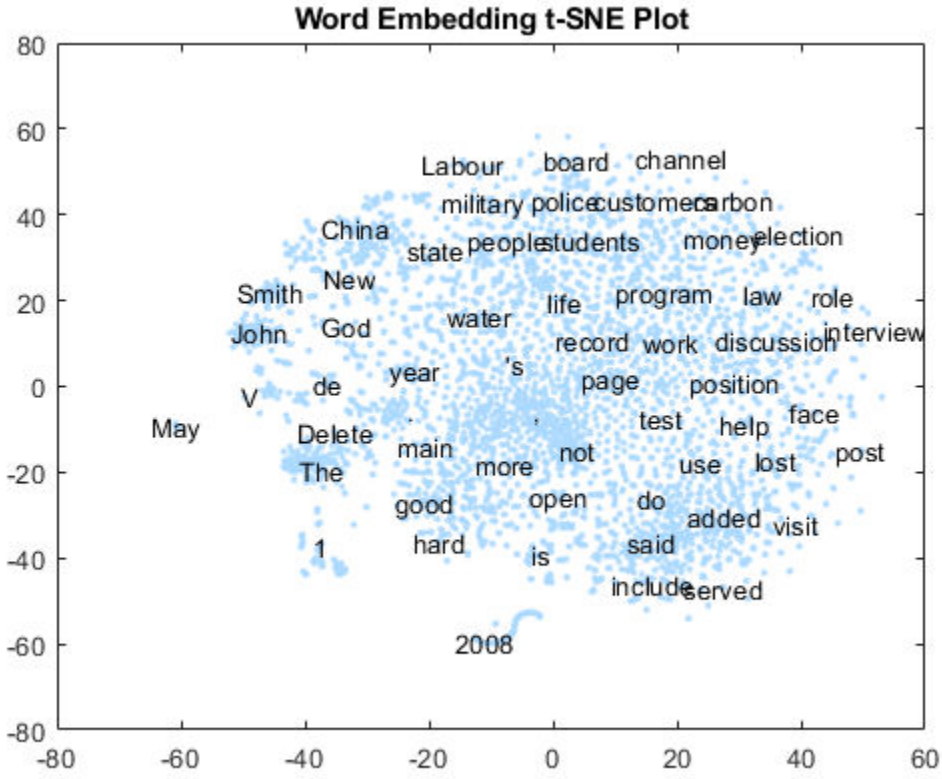
```
ans = 1×2

        5000         300
```

Embed the word vectors in two-dimensional space using `tsne`. This function may take a few minutes to run. If you want to display the convergence information, then set the `'Verbose'` name-value pair to 1.

```
XY = tsne(V);
```

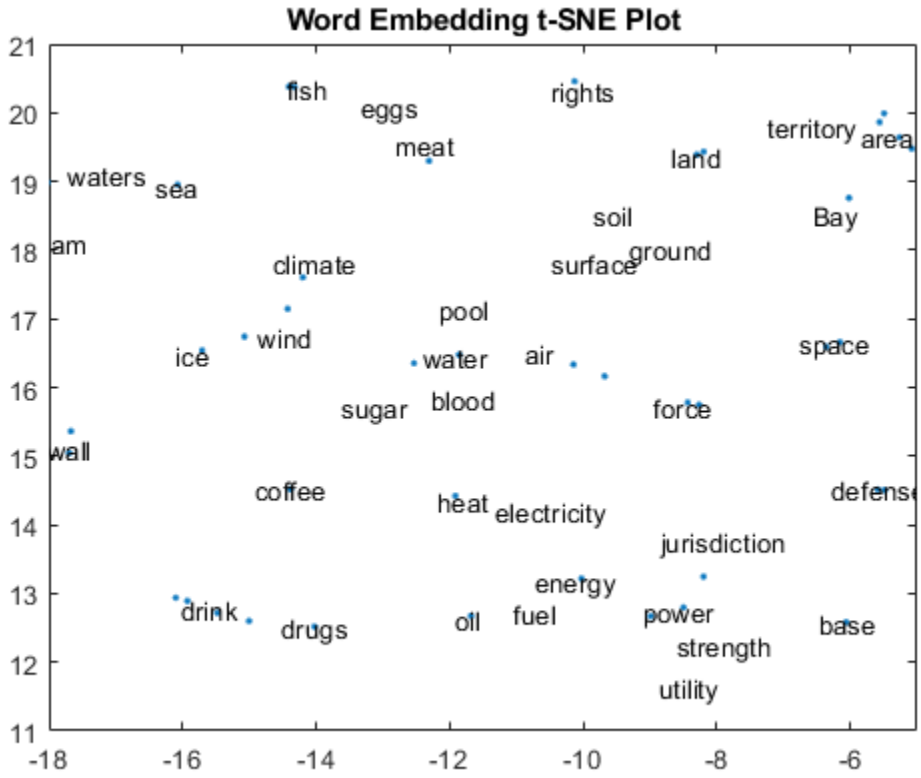
Plot the words at the coordinates specified by XY in a 2-D text scatter plot. For readability, `textscatter`, by default, does not display all of the input words and displays markers instead.

```
figure
textscatter(XY,words)
title("Word Embedding t-SNE Plot")
```



Zoom in on a section of the plot.

```
xlim([-18 -5])
ylim([11 21])
```



Create 3-D Text Scatter Plot

Visualize the word embedding by creating a 3-D text scatter plot using `tsne` and `textscatter`.

Convert the first 5000 words to vectors using `word2vec`. `V` is a matrix of word vectors of length 300.

```
words = emb.Vocabulary(1:5000);
V = word2vec(emb,words);
size(V)
```

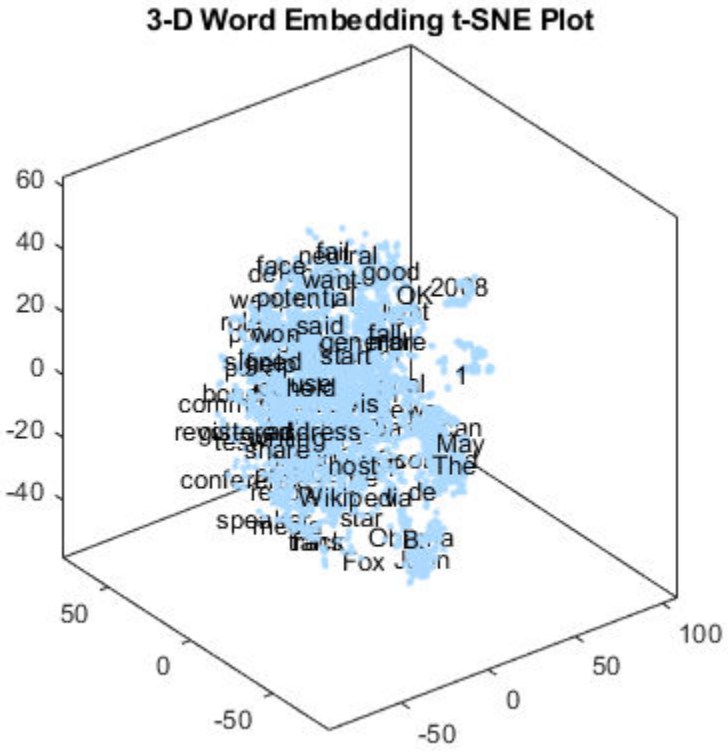
```
ans = 1×2

        5000         300
```

Embed the word vectors in a three-dimensional space using `tsne` by specifying the number of dimensions to be three. This function may take a few minutes to run. If you want to display the convergence information, then you can set the `'Verbose'` name-value pair to 1.

```
XYZ = tsne(V,'NumDimensions',3);
```

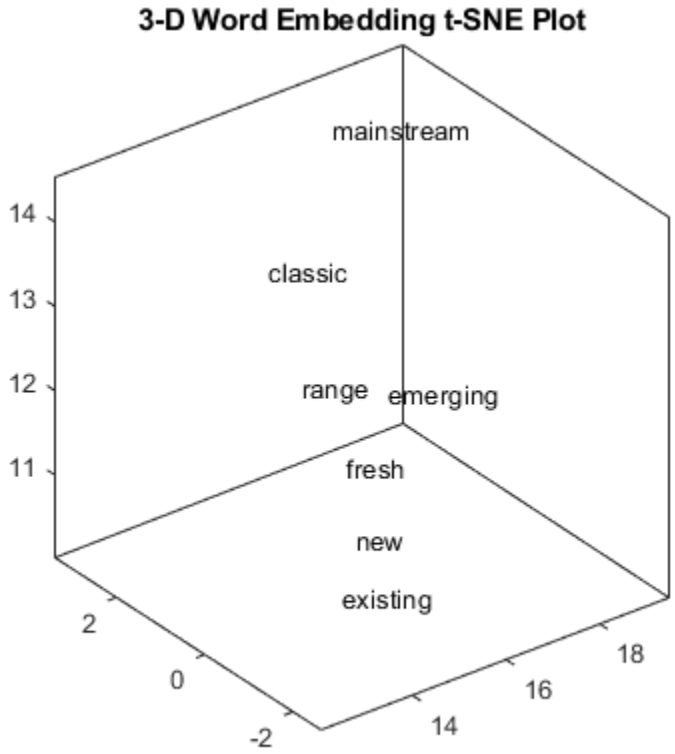
Plot the words at the coordinates specified by XYZ in a 3-D text scatter plot.

```
figure
ts = textscatter3(XYZ,words);
title("3-D Word Embedding t-SNE Plot")
```



Zoom in on a section of the plot.

```
xlim([12.04 19.48])
ylim([-2.66 3.40])
zlim([10.03 14.53])
```



Perform Cluster Analysis

Convert the first 5000 words to vectors using `word2vec`. `V` is a matrix of word vectors of length 300.

```
words = emb.Vocabulary(1:5000);
V = word2vec(emb,words);
size(V)
```

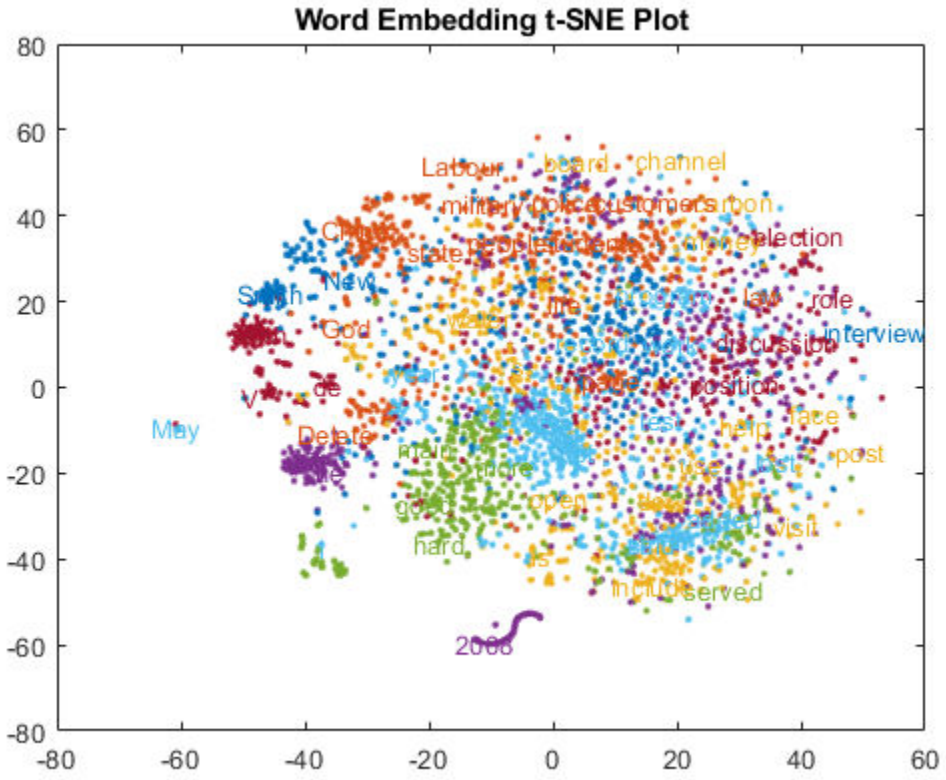
```
ans = 1×2

        5000         300
```

Discover 25 clusters using `kmeans`.

```
cidx = kmeans(V,25,'dist','sqeuclidean');
```

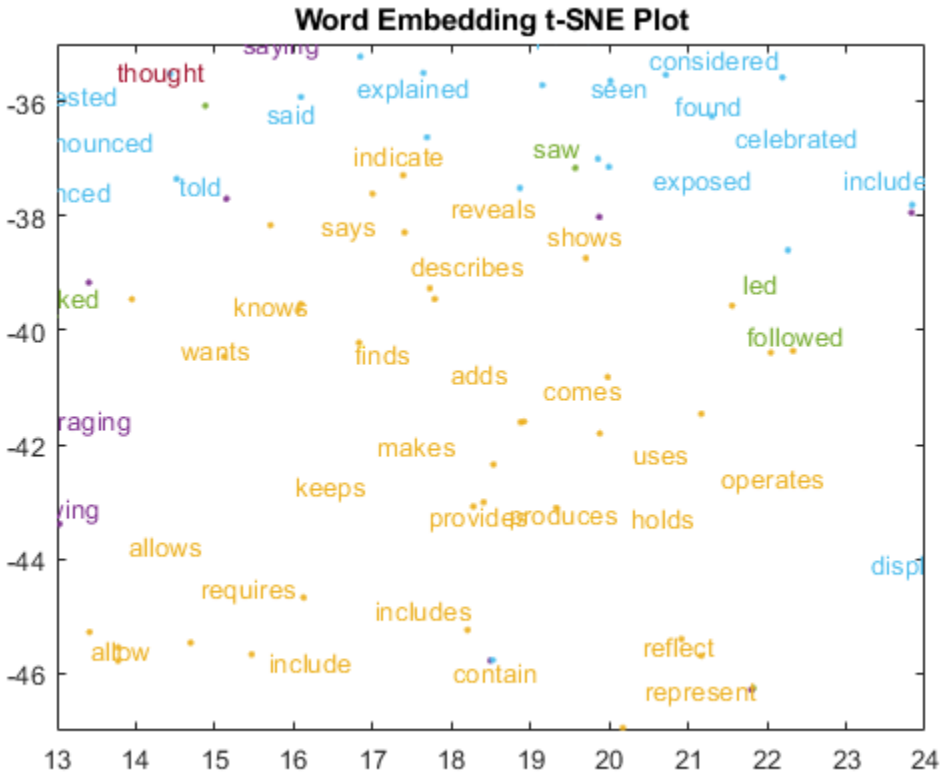
Visualize the clusters in a text scatter plot using the 2-D t-SNE data coordinates calculated earlier.

```
figure
textscatter(XY,words,'ColorData',categorical(cidx));
title("Word Embedding t-SNE Plot")
```



Zoom in on a section of the plot.

```
xlim([13 24])
ylim([-47 -35])
```



See Also

`readWordEmbedding` | `textscatter` | `textscatter3` | `tokenizedDocument` | `vec2word` | `word2vec` | `wordEmbedding`

Related Examples

- “Extract Text Data from Files” on page 1-2
- “Prepare Text Data for Analysis” on page 1-10
- “Visualize Text Data Using Word Clouds” on page 3-2
- “Classify Text Data Using Deep Learning” on page 2-65

Language Support

Language Considerations

Text Analytics Toolbox supports the languages English, Japanese, German, and Korean. Most Text Analytics Toolbox functions also work with text in other languages. This table summarizes how to use Text Analytics Toolbox features for other languages.

Feature	Language Consideration	Workaround
Tokenization	The `tokenizedDocument` function has built-in rules for English, Japanese, German, and Korean only. For English and German text, the `'unicode'` tokenization method of `tokenizedDocument` detects tokens using rules based on Unicode® Standard Annex #29 [1] and the ICU tokenizer [2], modified to better detect complex tokens such as hashtags and URLs. For Japanese and Korean text, the `'mecab'` tokenization method detects tokens using rules based on the MeCab tokenizer [3].	For other languages, you can still try using `tokenizedDocument`. If `tokenizedDocument` does not produce useful results, then try tokenizing the text manually. To create a `tokenizedDocument` array from manually tokenized text, set the `'TokenizeMethod'` option to `'none'`. For more information, see `tokenizedDocument`.
Stop word removal	The `stopWords` and `removeStopWords` functions support English, Japanese, German, and Korean stop words only.	To remove stop words from other languages, use `removeWords` and specify your own stop words to remove.
Sentence detection	The `addSentenceDetails` function detects sentence boundaries based on punctuation characters and line number information. For English and German text, the function also uses a list of abbreviations passed to the function.	For other languages, you might need to specify your own list of abbreviations for sentence detection. To do this, use the `'Abbreviations'` option of `addSentenceDetails`. For more information, see `addSentenceDetails`.

Feature	Language Consideration	Workaround
Word clouds	For string input, the `wordcloud` and `wordCloudCounts` functions use English, Japanese, German, and Korean tokenization, stop word removal, and word normalization.	For other languages, you might need to manually preprocess your text data and specify unique words and corresponding sizes in `wordcloud`. To specify word sizes in `wordcloud`, input your data as a table or arrays containing the unique words and corresponding sizes. For more information, see `wordcloud`.
Word embeddings	File input to the `trainWordEmbedding` function requires words separated by whitespace.	For files containing non-English text, you might need to input a `tokenizedDocument` array to `trainWordEmbedding`. To create a `tokenizedDocument` array from pretokenized text, use the `tokenizedDocument` function and set the `'TokenizeMethod'` option to `'none'`. For more information, see `trainWordEmbedding`.
Keyword extraction	The `rakeKeywords` function supports English, Japanese, German, and Korean text only.	The `rakeKeywords` function extracts keywords using a delimiter-based approach to identify candidate keywords. The function, by default, uses punctuation characters and the stop words given by the `stopWords` with language given by the language details of the input documents as delimiters. For other languages, specify an appropriate set of delimiters using the `'Delimiters'` and `'MergingDelimiters'` options. For more information, see `rakeKeywords`.

Feature	Language Consideration	Workaround
	The `textrankKeywords` function supports English, Japanese, German, and Korean text only.	The `textrankKeywords` function extracts keywords by identifying candidate keywords based on their part-of-speech tag. The function uses part-of-speech tags given by the `addPartOfSpeechDetails` function which supports English, Japanese, German, and Korean text only. For other languages, try using the `rakeKeywords` instead and specify an appropriate set of delimiters using the `'Delimiters'` and `'MergingDelimiters'` options. For more information, see `textrankKeywords`.

Language-Independent Features

Word and N-Gram Counting

The `bagOfWords` and `bagOfNgrams` functions support `tokenizedDocument` input regardless of language. If you have a `tokenizedDocument` array containing your data, then you can use these functions.

Modeling and Prediction

The `fitlda` and `fitlsa` functions support `bagOfWords` and `bagOfNgrams` input regardless of language. If you have a `bagOfWords` or `bagOfNgrams` object containing your data, then you can use these functions.

The `trainWordEmbedding` function supports `tokenizedDocument` or file input regardless of language. If you have a `tokenizedDocument` array or a file containing your data in the correct format, then you can use this function.

References

[1] *Unicode Text Segmentation*. https://www.unicode.org/reports/tr29/

[2] *Boundary Analysis*. http://userguide.icu-project.org/boundaryanalysis

[3] *MeCab: Yet Another Part-of-Speech and Morphological Analyzer*. https://taku910.github.io/mecab/

See Also

`addLanguageDetails` | `addSentenceDetails` | `bagOfNgrams` | `bagOfWords` | `fitlda` | `fitlsa` | `normalizeWords` | `removeWords` | `stopWords` | `tokenizedDocument` | `wordcloud`

More About

- “Text Data Preparation”
- “Modeling and Prediction”
- “Display and Presentation”
- “Japanese Language Support” on page 4-6
- “Analyze Japanese Text Data” on page 4-11
- “German Language Support” on page 4-21
- “Analyze German Text Data” on page 4-26

Japanese Language Support

This topic summarizes the Text Analytics Toolbox features that support Japanese text. For an example showing how to analyze Japanese text data, see “Analyze Japanese Text Data” on page 4-11.

Tokenization

The `tokenizedDocument` function automatically detects Japanese input. Alternatively, set the `'Language'` option in `tokenizedDocument` to `'ja'`. This option specifies the language details of the tokens. To view the language details of the tokens, use `tokenDetails`. These language details determine the behavior of the `removeStopWords`, `addPartOfSpeechDetails`, `normalizeWords`, `addSentenceDetails`, and `addEntityDetails` functions on the tokens.

To specify additional MeCab options for tokenization, create a `mecabOptions` object. To tokenize using the specified MeCab tokenization options, use the `'TokenizeMethod'` option of `tokenizedDocument`.

Tokenize Japanese Text

Tokenize Japanese text using `tokenizedDocument`. The function automatically detects Japanese text.

```
str = [
    "恋に悩み、苦しむ。"
    "恋の悩みで苦しむ。"
    "空に星が輝き、瞬いている。"
    "空の星が輝きを増している。"];
documents = tokenizedDocument(str)
```

```
documents =
  4x1 tokenizedDocument:

     6 tokens: 恋 に 悩み 、 苦しむ 。
     6 tokens: 恋 の 悩み で 苦しむ 。
    10 tokens: 空 に 星 が 輝き 、 瞬い て いる 。
    10 tokens: 空 の 星 が 輝き を 増し て いる 。
```

Part of Speech Details

The `tokenDetails` function, by default, includes part of speech details with the token details.

Get Part of Speech Details of Japanese Text

Tokenize Japanese text using `tokenizedDocument`.

```
str = [
    "恋に悩み、苦しむ。"
    "恋の悩みで 苦しむ。"
    "空に星が輝き、瞬いている。"
    "空の星が輝きを増している。"
    "駅までは遠くて、歩けない。"
    "遠くの駅まで歩けない。"
```

```
    "すももももももものうち。"];
documents = tokenizedDocument(str);
```

For Japanese text, you can get the part-of-speech details using `tokenDetails`. For English text, you must first use `addPartOfSpeechDetails`.

```
tdetails = tokenDetails(documents);
head(tdetails)
```

```
ans=8×8 table
     Token      DocumentNumber    LineNumber       Type        Language    PartOfSpeech      Lemma
    ________    ______________    __________    ___________    ________    ____________    ________

    "恋"              1               1         letters          ja        noun            "恋"
    "に"              1               1         letters          ja        adposition      "に"
    "悩み"            1               1         letters          ja        verb            "悩む"
    "、"              1               1         punctuation      ja        punctuation     "、"
    "苦しむ"          1               1         letters          ja        verb            "苦しむ"
    "。"              1               1         punctuation      ja        punctuation     "。"
    "恋"              2               1         letters          ja        noun            "恋"
    "の"              2               1         letters          ja        adposition      "の"
```

Named Entity Recognition

The `tokenDetails` function, by default, includes entity details with the token details.

Add Named Entity Tags to Japanese Text

Tokenize Japanese text using `tokenizedDocument`.

```
str = [
    "マリーさんはボストンからニューヨークに引っ越しました。"
    "駅で鈴木さんに迎えに行きます。"
    "東京は大阪より大きいですか？"
    "東京に行った時、新宿や渋谷などいろいろな所に訪れました。"];
documents = tokenizedDocument(str);
```

For Japanese text, the software automatically adds named entity tags, so you do not need to use the `addEntityDetails` function. This software detects person names, locations, organizations, and other named entities. To view the entity details, use the `tokenDetails` function.

```
tdetails = tokenDetails(documents);
head(tdetails)
```

```
ans=8×8 table
       Token        DocumentNumber    LineNumber      Type       Language    PartOfSpeech       Le
    ____________    ______________    __________    _______    ________    ____________    _____

    "マリー"            1               1          letters       ja        proper-noun     "マリー
    "さん"              1               1          letters       ja        noun            "さん"
    "は"                1               1          letters       ja        adposition      "は"
    "ボストン"          1               1          letters       ja        proper-noun     "ボスト
    "から"              1               1          letters       ja        adposition      "から"
    "ニューヨーク"      1               1          letters       ja        proper-noun     "ニュー
    "に"                1               1          letters       ja        adposition      "に"
```

```
    "引っ越し"            1              1         letters       ja       verb            "引っ越
```

View the words tagged with entity `"person"`, `"location"`, `"organization"`, or `"other"`. These words are the words not tagged `"non-entity"`.

```
idx = tdetails.Entity ~= "non-entity";
tdetails(idx,:).Token
```

```
ans = 11x1 string
    "マリー"
    "さん"
    "ボストン"
    "ニューヨーク"
    "鈴木"
    "さん"
    "東京"
    "大阪"
    "東京"
    "新宿"
    "渋谷"
```

Stop Words

To remove stop words from documents according to the token language details, use `removeStopWords`. For a list of Japanese stop words set the `'Language'` option in `stopWords` to `'ja'`.

Remove Japanese Stop Words

Tokenize Japanese text using `tokenizedDocument`. The function automatically detects Japanese text.

```
str = [
    "ここは静かなので、とても穏やかです"
    "企業内の顧客データを利用し、今年の売り上げを調べることが出来た。"
    "私は先生です。私は英語を教えています。"];
documents = tokenizedDocument(str);
```

Remove stop words using `removeStopWords`. The function uses the language details from `documents` to determine which language stop words to remove.

```
documents = removeStopWords(documents)
```

```
documents =
  3x1 tokenizedDocument:

     4 tokens: 静か 、 とても 穏やか
    10 tokens: 企業 顧客 データ 利用 、 今年 売り上げ 調べる 出来 。
     5 tokens: 先生 。 英語 教え 。
```

Lemmatization

To lemmatize tokens according to the token language details, use `normalizeWords` and set the `'Style'` option to `'lemma'`.

Lemmatize Japanese Text

Tokenize Japanese text using the `tokenizedDocument` function. The function automatically detects Japanese text.

```
str = [
    "空に星が輝き、瞬いている。"
    "空の星が輝きを増している。"
    "駅までは遠くて、歩けない。"
    "遠くの駅まで歩けない。"];
documents = tokenizedDocument(str);
```

Lemmatize the tokens using `normalizeWords`.

```
documents = normalizeWords(documents)
```

```
documents =
  4x1 tokenizedDocument:

    10 tokens: 空 に 星 が 輝く 、 瞬く て いる 。
    10 tokens: 空 の 星 が 輝き を 増す て いる 。
     9 tokens: 駅 まで は 遠い て 、 歩ける ない 。
     7 tokens: 遠く の 駅 まで 歩ける ない 。
```

Language-Independent Features

Word and N-Gram Counting

The `bagOfWords` and `bagOfNgrams` functions support `tokenizedDocument` input regardless of language. If you have a `tokenizedDocument` array containing your data, then you can use these functions.

Modeling and Prediction

The `fitlda` and `fitlsa` functions support `bagOfWords` and `bagOfNgrams` input regardless of language. If you have a `bagOfWords` or `bagOfNgrams` object containing your data, then you can use these functions.

The `trainWordEmbedding` function supports `tokenizedDocument` or file input regardless of language. If you have a `tokenizedDocument` array or a file containing your data in the correct format, then you can use this function.

See Also

`addEntityDetails` | `addLanguageDetails` | `addPartOfSpeechDetails` | `normalizeWords` | `removeStopWords` | `stopWords` | `tokenDetails` | `tokenizedDocument`

More About

- “Language Considerations” on page 4-2
- “Analyze Japanese Text Data” on page 4-11

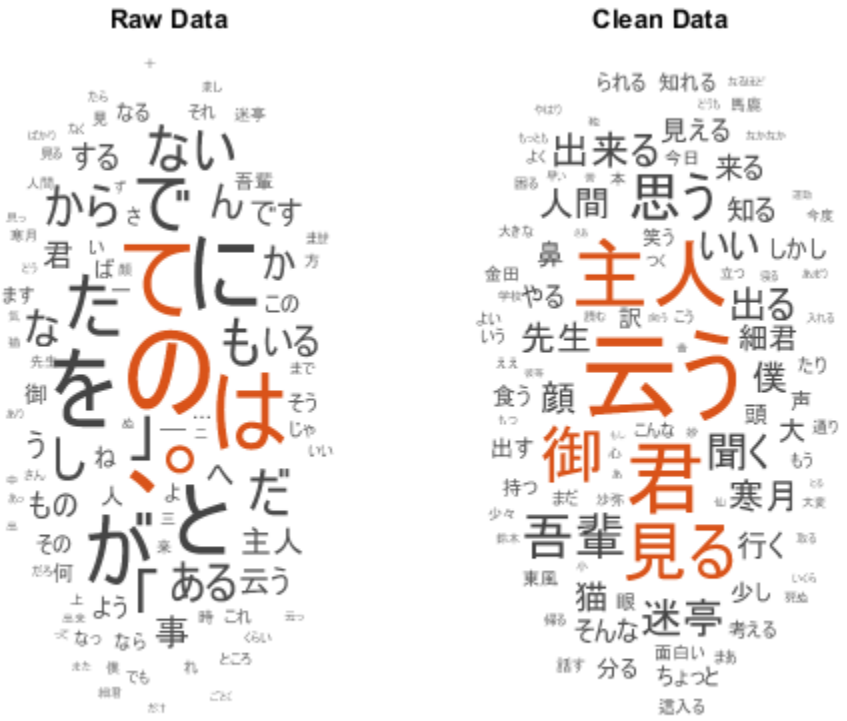
Analyze Japanese Text Data

This example shows how to import, prepare, and analyze Japanese text data using a topic model.

Japanese text data can be large and can contain lots of noise that negatively affects statistical analysis. For example, the text data can contain the following:

- Variations in word forms. For example, "難しい" ("is difficult") and "難しかった" ("was difficult")
- Words that add noise. For example, stop words such as "あそこ" ("over there"), "あたり" ("around"), and "あちら" ("there")
- Punctuation and special characters

These word clouds illustrate word frequency analysis applied to some raw text data from "吾輩は猫である" by 夏目漱石, and a preprocessed version of the same text data.



This example first shows how to import and prepare Japanese text data, and then it shows how to analyze the text data using a Latent Dirichlet Allocation (LDA) model. An LDA model is a topic model that discovers underlying topics in a collection of documents and infers the word probabilities in topics. Use these steps in preparing the text data and fitting the model:

- Read HTML code from a website.
- Parse the HTML code and extract the relevant data.
- Prepare the text data for analysis using standard preprocessing techniques.

- Fit a topic model and visualize the results.

Import Data

Read the data from "吾輩は猫である" by 夏目漱石 from https://www.aozora.gr.jp/cards/000148/files/789_14547.html using the `webread` function.

Specify the character encoding of the text using the `weboptions` function. To find the correct character encoding for an HTML, look in the header of the HTML code. For this file, specify the character encoding to be "`Shift_JIS`".

```
url = "https://www.aozora.gr.jp/cards/000148/files/789_14547.html";
options = weboptions('CharacterEncoding','Shift_JIS');
code = webread(url,options);
```

View the first few lines of the HTML code.

```
extractBefore(code,"<script")
```

```
ans =
    '<?xml version="1.0" encoding="Shift_JIS"?>
     <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
         "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
     <html xml:lang="ja" >
     <head>
         <meta http-equiv="Content-Type" content="text/html;charset=Shift_JIS" />
         <meta http-equiv="content-style-type" content="text/css" />
         <link rel="stylesheet" type="text/css" href="../../aozora.css" />
         <title>夏目漱石 吾輩は猫である</title>
         '
```

Extract the text data from the HTML using `extractHTMLText`. Split the text by `newline` characters.

```
textData = extractHTMLText(code);
textData = string(split(textData,newline));
textData(1:10)
```

```
ans = 10×1 string array
    "吾輩は猫である"
    ""
    "夏目漱石"
    ""
    " "
    ""
    "―"
    ""
    " 吾輩は猫である。名前はまだ無い。"
    " どこで生れたかとんと見当がつかぬ。何でも薄暗いじめじめした所でニャーニャー泣いていた事だけは記憶している。吾輩
```

Remove the empty lines of text.

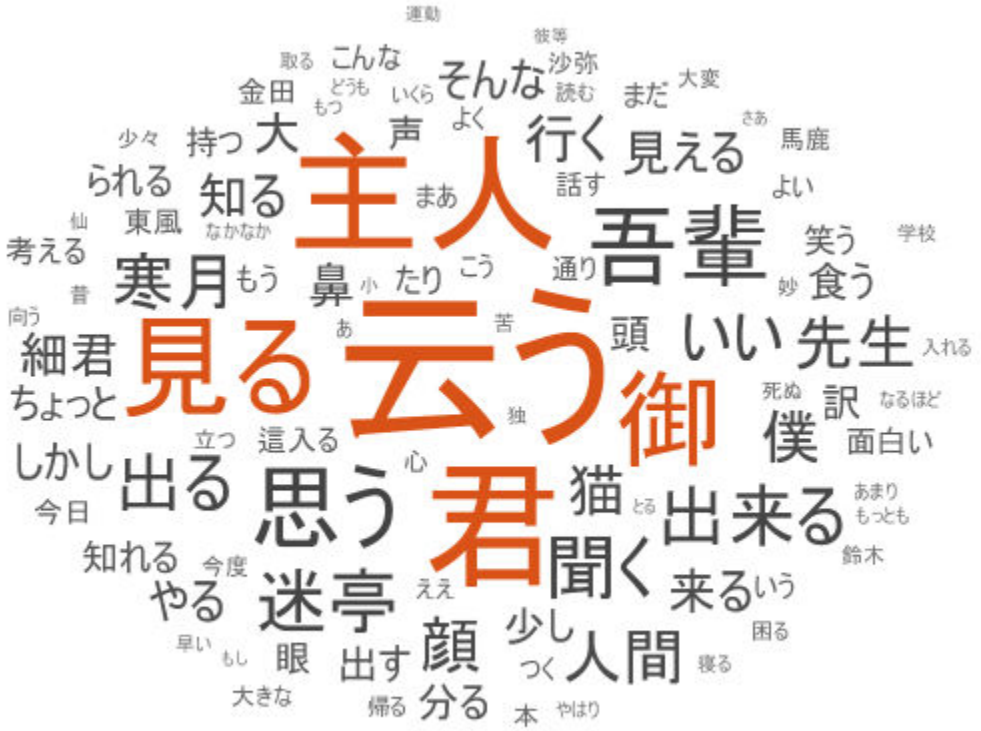
```
idx = textData == "";
textData(idx) = [];
textData(1:10)
```

```
ans = 10×1 string array
    "吾輩は猫である"
```

```
"夏目漱石"
" "
"一"
" 吾輩は猫である。名前はまだ無い。"
" どこで生れたかとんと見当がつかぬ。何でも薄暗いじめじめした所でニャーニャー泣いていた事だけは記憶している。吾輩
" この書生の掌の裏でしばらくはよい心持に坐っておったが、しばらくすると非常な速力で運転し始めた。書生が動くのか自
" ふと気が付いて見ると書生はいない。たくさんおった兄弟が一疋も見えぬ。肝心の母親さえ姿を隠してしまった。その上今
" ようやくの思いで笹原を這い出すと向うに大きな池がある。吾輩は池の前に坐ってどうしたらよかろうと考えて見た。別に
" 吾輩の主人は滅多に吾輩と顔を合せる事がない。職業は教師だそうだ。学校から帰ると終日書斎に這入ったぎりほとんど出
```

Visualize the text data in a word cloud.

```
figure
wordcloud(textData);
```



Tokenize Documents

Tokenize the text using `tokenizedDocument` and view the first few documents.

```
documents = tokenizedDocument(textData);
documents(1:10)
```

```
ans =
  10×1 tokenizedDocument:

      5 tokens: 吾輩 は 猫 で ある
      2 tokens: 夏目 漱石
```

```
  0 tokens:
  1 tokens: 一
 11 tokens: 吾輩 は 猫 で ある 。 名前 は まだ 無い 。
264 tokens: どこ で 生れ た か とんと 見当 が つか ぬ 。 何 でも 薄暗い じめじめ し た 所 で ニャーニャー
100 tokens: この 書生 の 掌 の 裏 で しばらく は よい 心持 に 坐っ て おっ た が 、 しばらく する と 非常
 92 tokens: ふと 気 が 付い て 見る と 書生 は い ない 。 たくさん おっ た 兄弟 が 一疋 も 見え ぬ 。 肝
693 tokens: ようやく の 思い で 笹原 を 這い出す と 向う に 大きな 池 が ある 。 吾輩 は 池 の 前 に 坐っ
276 tokens: 吾輩 の 主人 は 滅多 に 吾輩 と 顔 を 合せる 事 が ない 。 職業 は 教師 だ そう だ 。 学校 か
```

Get Part-of-Speech Tags

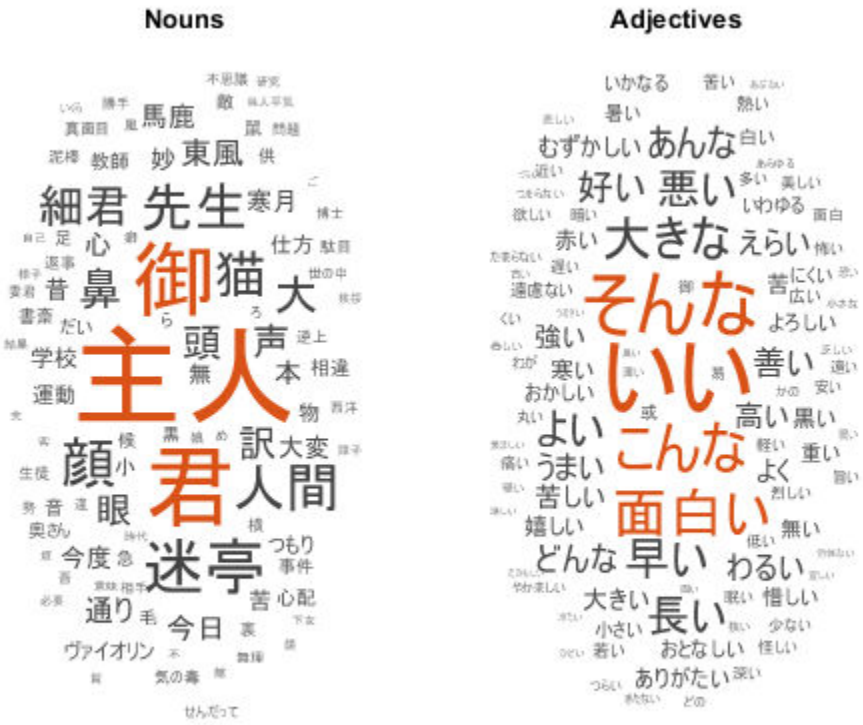
Get the token details and then view the details of the first few tokens.

```
tdetails = tokenDetails(documents);
head(tdetails)
```

```
ans=8×8 table
    Token      DocumentNumber    LineNumber      Type       Language      PartOfSpeech      Lemma
    _______    ______________    __________    _______    ________    _________________    ______

    "吾輩"           1               1         letters       ja       pronoun              "吾輩"
    "は"             1               1         letters       ja       adposition           "は"
    "猫"             1               1         letters       ja       noun                 "猫"
    "で"             1               1         letters       ja       auxiliary-verb       "だ"
    "ある"           1               1         letters       ja       auxiliary-verb       "ある"
    "夏目"           2               1         letters       ja       proper-noun          "夏目"
    "漱石"           2               1         letters       ja       proper-noun          "漱石"
    "一"             4               1         letters       ja       numeral              "一"
```

The `PartOfSpeech` variable in the table contains the part-of-speech tags of the tokens. Create word clouds of all the nouns and adjectives, respectively.

```
figure
idx = tdetails.PartOfSpeech == "noun";
tokens = tdetails.Token(idx);
subplot(1,2,1)
wordcloud(tokens);
title("Nouns")

idx = tdetails.PartOfSpeech == "adjective";
tokens = tdetails.Token(idx);
subplot(1,2,2)
wordcloud(tokens);
title("Adjectives")
```



Prepare Text Data for Analysis

Remove the stop words.

```
documents = removeStopWords(documents);
documents(1:10)
```

```
ans =
  10×1 tokenizedDocument:

     2 tokens: 吾輩 猫
     2 tokens: 夏目 漱石
     0 tokens:
     0 tokens:
     6 tokens: 吾輩 猫 。 まだ 無い 。
   117 tokens: 生れ とんと 見当 つか ぬ 。 薄暗い じめじめ ニャーニャー 泣い いた事 記憶 。 吾輩 始め 人間 と
    43 tokens: 書生 掌 裏 しばらく よい 心持 坐っ おっ 、 しばらく 非常 速力 運転 始め 。 書生 動く 動く 分ら
    46 tokens: ふと 付い 見る 書生 。 おっ 兄弟 一疋 見え ぬ 。 肝心 母親 姿 隠し しまっ 。 上今 違っ 無 暗い
   323 tokens: ようやく 思い 笹原 這い出す 向う 大きな 池 。 吾輩 池 坐っ たら よかろ 考え 。 別に という 分
   122 tokens: 吾輩 主人 滅多 吾輩 顔 合せる 。 職業 教師 。 学校 帰る 終日 書斎 這入っ ぎりほとんど 出 来る
```

Erase the punctuation.

```
documents = erasePunctuation(documents);
documents(1:10)
```

```
ans =
  10×1 tokenizedDocument:
```

```
     2 tokens: 吾輩 猫
     2 tokens: 夏目 漱石
     0 tokens:
     0 tokens:
     4 tokens: 吾輩 猫 まだ 無い
   102 tokens: 生れ とんと 見当 つか ぬ 薄暗い じめじめ ニャーニャー 泣い いた事 記憶 吾輩 始め 人間 という
    36 tokens: 書生 掌 裏 しばらく よい 心持 坐っ おっ しばらく 非常 速力 運転 始め 書生 動く 動く 分ら 無 暗
    38 tokens: ふと 付い 見る 書生 おっ 兄弟 一疋 見え ぬ 肝心 母親 姿 隠し しまっ 上今 違っ 無 暗に 明るい
   274 tokens: ようやく 思い 笹原 這い出す 向う 大きな 池 吾輩 池 坐っ たら よかろ 考え 別に という 分別 出
   101 tokens: 吾輩 主人 滅多 吾輩 顔 合せる 職業 教師 学校 帰る 終日 書斎 這入っ ぎりほとんど 出 来る 大変
```

Lemmatize the text using `normalizeWords`.

```
documents = normalizeWords(documents);
documents(1:10)
```

```
ans =
  10×1 tokenizedDocument:

     2 tokens: 吾輩 猫
     2 tokens: 夏目 漱石
     0 tokens:
     0 tokens:
     4 tokens: 吾輩 猫 まだ 無い
   102 tokens: 生れる とんと 見当 つく ぬ 薄暗い じめじめ ニャーニャー 泣く いた事 記憶 吾輩 始める 人間 とい
    36 tokens: 書生 掌 裏 しばらく よい 心持 坐る おる しばらく 非常 速力 運転 始める 書生 動く 動く 分る 無
    38 tokens: ふと 付く 見る 書生 おる 兄弟 一疋 見える ぬ 肝心 母親 姿 隠す しまう 上今 違う 無 暗に 明るい
   274 tokens: ようやく 思い 笹原 這い出す 向う 大きな 池 吾輩 池 坐る た よい 考える 別に という 分別 出る
   101 tokens: 吾輩 主人 滅多 吾輩 顔 合せる 職業 教師 学校 帰る 終日 書斎 這入る ぎりほとんど 出る 来る 大変
```

Some preprocessing steps, such as removing stop words and erasing punctuation, return empty documents. Remove the empty documents using the `removeEmptyDocuments` function.

```
documents = removeEmptyDocuments(documents);
```

Create Preprocessing Function

Creating a function that performs preprocessing can be useful to prepare different collections of text data in the same way. For example, you can use a function to preprocess new data using the same steps as the training data.

Create a function which tokenizes and preprocesses the text data to use for analysis. The function `preprocessJapaneseText`, performs these steps:

1. Tokenize the text using `tokenizedDocument`.
2. Erase punctuation using `erasePunctuation`.
3. Remove a list of stop words (such as "あそこ", "あたり", and "あちら") using `removeStopWords`.
4. Lemmatize the words using `normalizeWords`.

Remove the empty documents after preprocessing using the `removeEmptyDocuments` function. Removing documents after using a preprocessing function makes it easier to remove corresponding data such as labels from other sources.

In this example, use the preprocessing function `preprocessJapaneseText`, listed at the end of the example, to prepare the text data.

```
documents = preprocessJapaneseText(textData);
documents(1:5)
```

```
ans =
  5×1 tokenizedDocument:

    2 tokens: 吾輩 猫
    2 tokens: 夏目 漱石
    0 tokens:
    0 tokens:
    4 tokens: 吾輩 猫 まだ 無い
```

Remove the empty documents.

```
documents = removeEmptyDocuments(documents);
```

Fit Topic Model

Fit a latent Dirichlet allocation (LDA) topic model to the data. An LDA model discovers underlying topics in a collection of documents and infers word probabilities in topics.

To fit an LDA model to the data, you first must create a bag-of-words model. A bag-of-words model (also known as a term-frequency counter) records the number of times that words appear in each document of a collection. Create a bag-of-words model using `bagOfWords`.

```
bag = bagOfWords(documents);
```

Remove the empty documents from the bag-of-words model.

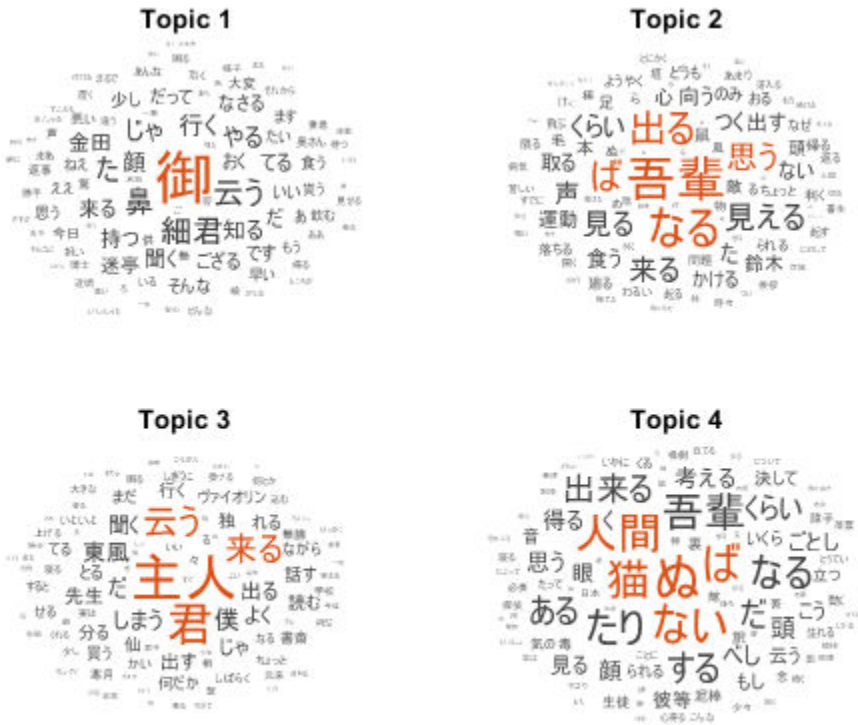
```
bag = removeEmptyDocuments(bag);
```

Fit an LDA model with seven topics using `fitlda`. To suppress the verbose output, set `'Verbose'` to `0`.

```
numTopics = 7;
mdl = fitlda(bag,numTopics,'Verbose',0);
```

Visualize the first four topics using word clouds.

```
figure
for i = 1:4
    subplot(2,2,i)
    wordcloud(mdl,i);
    title("Topic " + i)
end
```

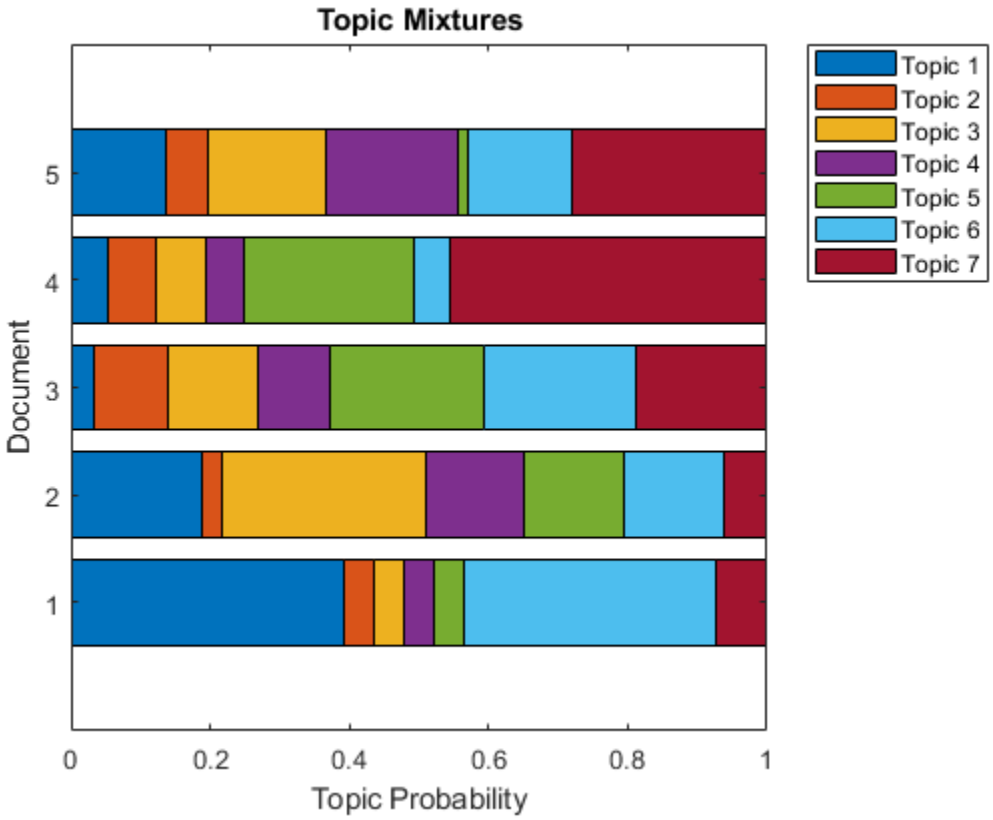


Visualize multiple topic mixtures using stacked bar charts. View five input documents at random and visualize the corresponding topic mixtures.

```
numDocuments = numel(documents);
idx = randperm(numDocuments,5);
documents(idx)
```

```
ans =
  5×1 tokenizedDocument:

     4 tokens: 細君 細君 なかなか さばける
     7 tokens: 進行 せる 山々 どうしても 暮れる くれる 困る
    13 tokens: 来る そんな 仙骨 相手 少々 骨 折れる 過ぎる 宛然 たり 仙 伝 人物
     3 tokens: 先生 譜 下さる
    23 tokens: 立つ 月給 上がる いくら 勉強 褒める くれる 郎 君 独 寂寞 中学 時代 覚える 詩 句 細君 朗吟 細君
```

```
topicMixtures = transform(mdl,documents(idx));
figure
barh(topicMixtures(1:5,:),'stacked')
xlim([0 1])
title("Topic Mixtures")
xlabel("Topic Probability")
ylabel("Document")
legend("Topic " + string(1:numTopics),'Location','northeastoutside')
```



Example Preprocessing Function

The function `preprocessJapaneseText`, performs these steps:

1. Tokenize the text using `tokenizedDocument`.
2. Erase punctuation using `erasePunctuation`.
3. Remove a list of stop words (such as "あそこ", "あたり", and "あちら") using `removeStopWords`.
4. Lemmatize the words using `normalizeWords`.

```
function documents = preprocessJapaneseText(textData)

% Tokenize the text.
documents = tokenizedDocument(textData);

% Erase the punctuation.
documents = erasePunctuation(documents);

% Remove a list of stop words.
documents = removeStopWords(documents);

% Lemmatize the words.
```

```
documents = normalizeWords(documents,'Style','lemma');
end
```

See Also

`addPartOfSpeechDetails` | `normalizeWords` | `removeStopWords` | `stopWords` | `tokenDetails` | `tokenizedDocument`

More About

- “Language Considerations” on page 4-2
- “Create Simple Text Model for Classification” on page 2-2
- “Analyze Text Data Using Topic Models” on page 2-13
- “Analyze Text Data Using Multiword Phrases” on page 2-7
- “Analyze Text Data Containing Emojis” on page 2-32
- “Train a Sentiment Classifier” on page 2-51
- “Classify Text Data Using Deep Learning” on page 2-65
- “Generate Text Using Deep Learning” (Deep Learning Toolbox)

See Also

German Language Support

This topic summarizes the Text Analytics Toolbox features that support German text. For an example showing how to analyze German text data, see “Analyze German Text Data” on page 4-26.

Tokenization

The `tokenizedDocument` function automatically detects German input. Alternatively, set the `'Language'` option in `tokenizedDocument` to `'de'`. This option specifies the language details of the tokens. To view the language details of the tokens, use `tokenDetails`. These language details determine the behavior of the `removeStopWords`, `addPartOfSpeechDetails`, `normalizeWords`, `addSentenceDetails`, and `addEntityDetails` functions on the tokens.

Tokenize German Text

Tokenize German text using `tokenizedDocument`. The function automatically detects German text.

```
str = [
    "Guten Morgen. Wie geht es dir?"
    "Heute wird ein guter Tag."];
documents = tokenizedDocument(str)
```

```
documents =
  2x1 tokenizedDocument:

    8 tokens: Guten Morgen . Wie geht es dir ?
    6 tokens: Heute wird ein guter Tag .
```

Sentence Detection

To detect sentence structure in documents, use the `addSentenceDetails`. You can use the `abbreviations` function to help create custom lists of abbreviations to detect.

Add Sentence Details to German Documents

Tokenize German text using `tokenizedDocument`.

```
str = [
    "Guten Morgen, Dr. Schmidt. Geht es Ihnen wieder besser?"
    "Heute wird ein guter Tag."];
documents = tokenizedDocument(str);
```

Add sentence details to the documents using `addSentenceDetails`. This function adds the sentence numbers to the table returned by `tokenDetails`. View the updated token details of the first few tokens.

```
documents = addSentenceDetails(documents);
tdetails = tokenDetails(documents);
head(tdetails,10)
```

```
ans=10×6 table
      Token      DocumentNumber    SentenceNumber    LineNumber       Type        Language
    _________    ______________    ______________    __________    ___________    ________
```

```
    "Guten"           1                1                1          letters           de
    "Morgen"          1                1                1          letters           de
    ","               1                1                1          punctuation       de
    "Dr"              1                1                1          letters           de
    "."               1                1                1          punctuation       de
    "Schmidt"         1                1                1          letters           de
    "."               1                1                1          punctuation       de
    "Geht"            1                2                1          letters           de
    "es"              1                2                1          letters           de
    "Ihnen"           1                2                1          letters           de
```

Table of German Abbreviations

View a table of German abbreviations. Use this table to help create custom tables of abbreviations for sentence detection when using `addSentenceDetails`.

```
tbl = abbreviations('Language','de');
head(tbl)
```

```
ans=8×2 table
    Abbreviation     Usage
    ____________    _______

       "A.T"        regular
       "ABl"        regular
       "Abb"        regular
       "Abdr"       regular
       "Abf"        regular
       "Abfl"       regular
       "Abh"        regular
       "Abk"        regular
```

Part of Speech Details

To add German part of speech details to documents, use the `addPartOfSpeechDetails` function.

Get Part of Speech Details of German Text

Tokenize German text using `tokenizedDocument`.

```
str = [
    "Guten Morgen. Wie geht es dir?"
    "Heute wird ein guter Tag."];
documents = tokenizedDocument(str)
```

```
documents =
  2x1 tokenizedDocument:

    8 tokens: Guten Morgen . Wie geht es dir ?
    6 tokens: Heute wird ein guter Tag .
```

To get the part of speech details for German text, first use `addPartOfSpeechDetails`.

```
documents = addPartOfSpeechDetails(documents);
```

To view the part of speech details, use the `tokenDetails` function.

```
tdetails = tokenDetails(documents);
head(tdetails)
```

```
ans=8×7 table
     Token      DocumentNumber    SentenceNumber    LineNumber       Type        Language    Part
    ________    ______________    ______________    __________    ___________    ________    ____

    "Guten"           1                 1               1         letters          de       adje
    "Morgen"          1                 1               1         letters          de       noun
    "."               1                 1               1         punctuation      de       punc
    "Wie"             1                 2               1         letters          de       adve
    "geht"            1                 2               1         letters          de       verb
    "es"              1                 2               1         letters          de       pron
    "dir"             1                 2               1         letters          de       pron
    "?"               1                 2               1         punctuation      de       punc
```

Named Entity Recognition

To add entity tags to documents, use the `addEntityDetails` function.

Add Named Entity Tags to German Text

Tokenize German text using `tokenizedDocument`.

```
str = [
    "Ernst zog von Frankfurt nach Berlin."
    "Besuchen Sie Volkswagen in Wolfsburg."];
documents = tokenizedDocument(str);
```

To add entity tags to German text, use the `addEntityDetails` function. This function detects person names, locations, organizations, and other named entities.

```
documents = addEntityDetails(documents);
```

To view the entity details, use the `tokenDetails` function.

```
tdetails = tokenDetails(documents);
head(tdetails)
```

```
ans=8×8 table
       Token       DocumentNumber    SentenceNumber    LineNumber       Type        Language    E
    ___________    ______________    ______________    __________    ___________    ________    _

    "Ernst"            1                 1               1         letters          de       p
    "zog"              1                 1               1         letters          de       v
    "von"              1                 1               1         letters          de       a
    "Frankfurt"        1                 1               1         letters          de       p
    "nach"             1                 1               1         letters          de       a
    "Berlin"           1                 1               1         letters          de       p
    "."                1                 1               1         punctuation      de       p
    "Besuchen"         2                 1               1         letters          de       v
```

View the words tagged with entity `"person"`, `"location"`, `"organization"`, or `"other"`. These words are the words not tagged with `"non-entity"`.

```
idx = tdetails.Entity ~= "non-entity";
tdetails(idx,:)
```

```
ans=5×8 table
       Token        DocumentNumber    SentenceNumber    LineNumber     Type      Language    Part
    ____________    ______________    ______________    __________    _______    ________    ____

    "Ernst"              1                 1                1         letters       de       prop
    "Frankfurt"          1                 1                1         letters       de       prop
    "Berlin"             1                 1                1         letters       de       prop
    "Volkswagen"         2                 1                1         letters       de       noun
    "Wolfsburg"          2                 1                1         letters       de       prop
```

Stop Words

To remove stop words from documents according to the token language details, use `removeStopWords`. For a list of German stop words set the `'Language'` option in `stopWords` to `'de'`.

Remove German Stop Words from Documents

Tokenize German text using `tokenizedDocument`. The function automatically detects German text.

```
str = [
    "Guten Morgen. Wie geht es dir?"
    "Heute wird ein guter Tag."];
documents = tokenizedDocument(str)
```

```
documents =
  2x1 tokenizedDocument:

    8 tokens: Guten Morgen . Wie geht es dir ?
    6 tokens: Heute wird ein guter Tag .
```

Remove stop words using the `removeStopWords` function. The function uses the language details from documents to determine which language stop words to remove.

```
documents = removeStopWords(documents)
```

```
documents =
  2x1 tokenizedDocument:

    5 tokens: Guten Morgen . geht ?
    5 tokens: Heute wird guter Tag .
```

Stemming

To stem tokens according to the token language details, use `normalizeWords`.

Stem German Text

Tokenize German text using the `tokenizedDocument` function. The function automatically detects German text.

```
str = [
    "Guten Morgen. Wie geht es dir?"
    "Heute wird ein guter Tag."];
documents = tokenizedDocument(str);
```

Stem the tokens using `normalizeWords`.

```
documents = normalizeWords(documents)
```

```
documents =
  2x1 tokenizedDocument:

    8 tokens: gut morg . wie geht es dir ?
    6 tokens: heut wird ein gut tag .
```

Language-Independent Features

Word and N-Gram Counting

The `bagOfWords` and `bagOfNgrams` functions support `tokenizedDocument` input regardless of language. If you have a `tokenizedDocument` array containing your data, then you can use these functions.

Modeling and Prediction

The `fitlda` and `fitlsa` functions support `bagOfWords` and `bagOfNgrams` input regardless of language. If you have a `bagOfWords` or `bagOfNgrams` object containing your data, then you can use these functions.

The `trainWordEmbedding` function supports `tokenizedDocument` or file input regardless of language. If you have a `tokenizedDocument` array or a file containing your data in the correct format, then you can use this function.

See Also

`addLanguageDetails` | `addPartOfSpeechDetails` | `normalizeWords` | `removeStopWords` | `stopWords` | `tokenDetails` | `tokenizedDocument`

More About

- “Language Considerations” on page 4-2
- “Analyze German Text Data” on page 4-26

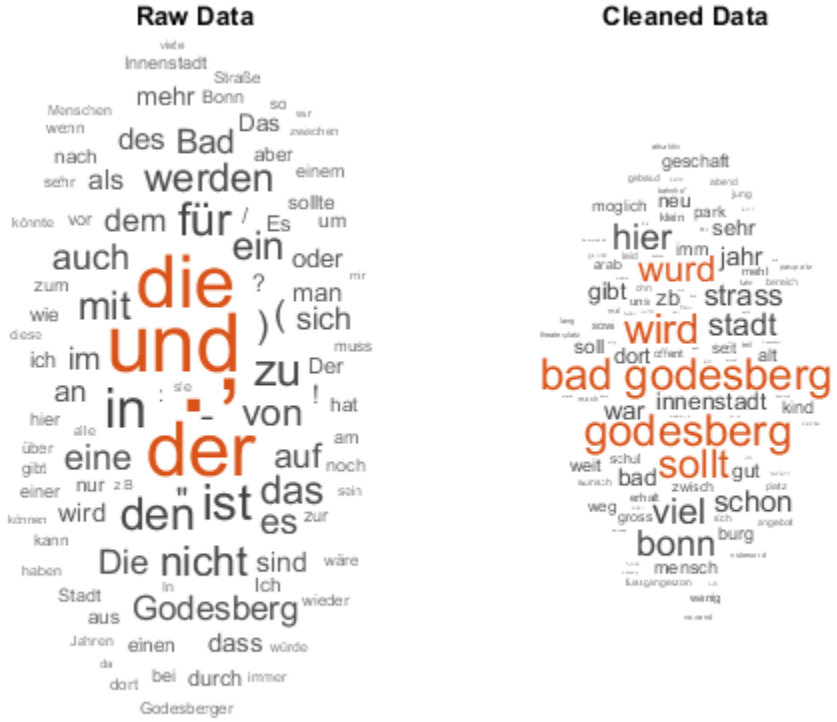
Analyze German Text Data

This example shows how to import, prepare, and analyze German text data using a topic model.

German text data can be large and can contain lots of noise that negatively affects statistical analysis. For example, the text data can contain the following:

- Variations in word forms. For example, „rot“, „rote“, and „roten“.
- Words that add noise. For example, stop words such as „der“, „die“, and „das“.
- Punctuation and special characters.

These word clouds illustrate word frequency analysis applied to some raw text data and a preprocessed version of the same text data.



This example first shows how to import and prepare German text data, and then it shows how to analyze the text data using a Latent Dirichlet Allocation (LDA) model. An LDA model is a topic model that discovers underlying topics in a collection of documents and infers the word probabilities in topics. Use these steps in preparing the text data and fitting the model:

- Import the text data from a CSV file and extract the relevant data.
- Prepare the text data for analysis using standard preprocessing techniques.
- Fit a topic model and visualize the results.

Import Data

Download the data `vorhaben.csv` from https://opendata.bonn.de/dataset/vorhabenliste-b%C3%BCrgerbeteiligungen-planungen-und-projekte. This file can change over time, so the results in the example can vary.

Use `detectImportOptions` to determine the format of the CSV file and set the text type to string. Set the `'Encoding'` option to `'ISO-8859-15'`. Read the data using the `readtable` function and view the first few rows.

```
filename = "vorhaben.csv";
options = detectImportOptions(filename,'TextType','string','Encoding','ISO-8859-15');
data = readtable(filename,options);
head(data)
```

```
ans=8×19 table
                                                                Titel
    ____________________________________________________________________________________________________

    "Bauleitplanverfahren zur Aufstellung des vorhabenbezogenen↵Bebauungsplans Nr. 6620-1 ?Bundes
    "Bauleitplanverfahren zur Aufstellung des vorhabenbezogenen↵Bebauungsplans Nr. 6522-1 "Didinh
    "Bauleitplanverfahren zur Aufstellung des Bebauungsplans↵Nr. 7621-56 ?Sebastianstraße?↵"
    "EPICURO - European Partnership for Innovative Cities within and Urban Resilience Outlook"
    "Bauleitplanverfahren zur Aufstellung des Bebauungsplanes Nr. 6719-3 "Schwimmbad Wasserland""
    "Bürgerbeteiligung an der Konzepterstellung für den Neubau eines Schwimmbades in Bonn-Dottend
    "Integriertes Handlungskonzept Grüne Infrastruktur (InHK GI) zur↵zukünftigen Freiraumsicherun
    "Verlängerung des Teufelsbachweges bis zur L 83n"
```

Extract the text data from the variable `InhaltlicheBeschreibungUndZielsetzung` (the description of the content and the goal).

```
textData = data.InhaltlicheBeschreibungUndZielsetzung;
```

Visualize the text data in a word cloud.

```
figure
wordcloud(textData);
```

Tokenize Text Data

Create an array of tokenized documents using the `tokenizedDocument` function.

```
documents = tokenizedDocument(textData);
documents(1:10)
```

```
ans =
  10×1 tokenizedDocument:

     50 tokens: Für das Gebiet zwischen Reuterstraße , Bundeskanzlerplatz , Willy-Brandt-Allee ,
     46 tokens: Für den vorhabenbezogenen Bebauungsplan Nr . 6522-1 ? Didinkirica ? der Bundesst
     41 tokens: Für das Gebiet zwischen Alfred-Bucherer-Straße , Sebastianstraße und dem Fußweg z
    134 tokens: In den vergangenen Jahren führte der Klimawandel zu einer Vielzahl von Folgen fü
     24 tokens: Schaffung von Planungsrecht für den Bau eines neuen Familien - , Schul - und Spo
     80 tokens: Für die begleitende Bürgerbeteiligung bei der Konzepterstellung für das neue Schw
     60 tokens: In der Gebietskulisse des Grünen C sollen die Freiräume auch zukünftig im Sinne v
     51 tokens: Zur Entlastung von Pützchen / Bechlinghoven vor Durchgangsverkehr und in Verbindu
     29 tokens: Für das Areal der ehemaligen Landwirtschaftskammer sowie einer angrenzenden städt
     37 tokens: Für das Areal Herbert-Rabius-Straße im Stadtbezirk Beuel , Ortsteil Beuel-Mitte s
```

Get Part-of-Speech Tags

Add the part of speech details using the `addPartOfSpeechDetails` function.

```
documents = addPartOfSpeechDetails(documents);
```

Get the token details and then view the details of the first few tokens.

```
tdetails = tokenDetails(documents);
head(tdetails)
```

```
ans=8×7 table
           Token            DocumentNumber    SentenceNumber    LineNumber       Type        Lang
    ____________________    ______________    ______________    __________    ___________    ____

    "Für"                          1                 1              1         letters          d
    "das"                          1                 1              1         letters          d
    "Gebiet"                       1                 1              1         letters          d
    "zwischen"                     1                 1              1         letters          d
    "Reuterstraße"                 1                 1              1         letters          d
    ","                            1                 1              1         punctuation      d
    "Bundeskanzlerplatz"           1                 1              1         letters          d
    ","                            1                 1              1         punctuation      d
```

The `PartOfSpeech` variable in the table contains the part-of-speech tags of the tokens. Create word clouds of all the nouns and adjectives, respectively.

```
figure
idx = tdetails.PartOfSpeech == "noun";
tokens = tdetails.Token(idx);
subplot(1,2,1)
wordcloud(tokens);
title("Nouns")

idx = tdetails.PartOfSpeech == "adjective";
tokens = tdetails.Token(idx);
subplot(1,2,2)
wordcloud(tokens);
title("Adjectives")
```

Prepare Text Data for Analysis

Tokenize the text using `tokenizedDocument` and view the first few documents.

```
documentsRaw = tokenizedDocument(textData);
documents = documentsRaw;
documents(1:10)
```

```
ans =
  10×1 tokenizedDocument:

     50 tokens: Für das Gebiet zwischen Reuterstraße , Bundeskanzlerplatz , Willy-Brandt-Allee ,
     46 tokens: Für den vorhabenbezogenen Bebauungsplan Nr . 6522-1 ? Didinkirica ? der Bundesst
     41 tokens: Für das Gebiet zwischen Alfred-Bucherer-Straße , Sebastianstraße und dem Fußweg z
    134 tokens: In den vergangenen Jahren führte der Klimawandel zu einer Vielzahl von Folgen fü
     24 tokens: Schaffung von Planungsrecht für den Bau eines neuen Familien - , Schul - und Spo
     80 tokens: Für die begleitende Bürgerbeteiligung bei der Konzepterstellung für das neue Schw
     60 tokens: In der Gebietskulisse des Grünen C sollen die Freiräume auch zukünftig im Sinne v
     51 tokens: Zur Entlastung von Pützchen / Bechlinghoven vor Durchgangsverkehr und in Verbindu
     29 tokens: Für das Areal der ehemaligen Landwirtschaftskammer sowie einer angrenzenden städt
     37 tokens: Für das Areal Herbert-Rabius-Straße im Stadtbezirk Beuel , Ortsteil Beuel-Mitte s
```

Replace common phrases (n-grams) with a single token and remove the stop words.

```
old = ["Bad" "Godesberg"];
new = "Bad Godesberg";
documents = replaceNgrams(documents,old,new);
```

```
documents = removeStopWords(documents);
documents(1:10)

ans =
  10×1 tokenizedDocument:

    35 tokens: Gebiet zwischen Reuterstraße , Bundeskanzlerplatz , Willy-Brandt-Allee , Eduard-Pf
    33 tokens: vorhabenbezogenen Bebauungsplan Nr . 6522-1 ? Didinkirica ? Bundesstadt Bonn , Sta
    27 tokens: Gebiet zwischen Alfred-Bucherer-Straße , Sebastianstraße Fußweg zwischen Röckumstr
    81 tokens: vergangenen Jahren führte Klimawandel Vielzahl Folgen Umwelt , Wirtschaft Menschen
    15 tokens: Schaffung Planungsrecht Bau neuen Familien - , Schul - Sportschwimmbades Flächen n
    57 tokens: begleitende Bürgerbeteiligung Konzepterstellung neue Schwimmbad soll folgenden The
    40 tokens: Gebietskulisse Grünen C sollen Freiräume zukünftig Sinne Naherholung , Landwirtsch
    32 tokens: Entlastung Pützchen / Bechlinghoven Durchgangsverkehr Verbindung geplanten Anschlu
    19 tokens: Areal ehemaligen Landwirtschaftskammer sowie angrenzenden städtischen Fläche Stadt
    25 tokens: Areal Herbert-Rabius-Straße Stadtbezirk Beuel , Ortsteil Beuel-Mitte soll vorhaben
```

Normalize the text using the `normalizeWords` function.

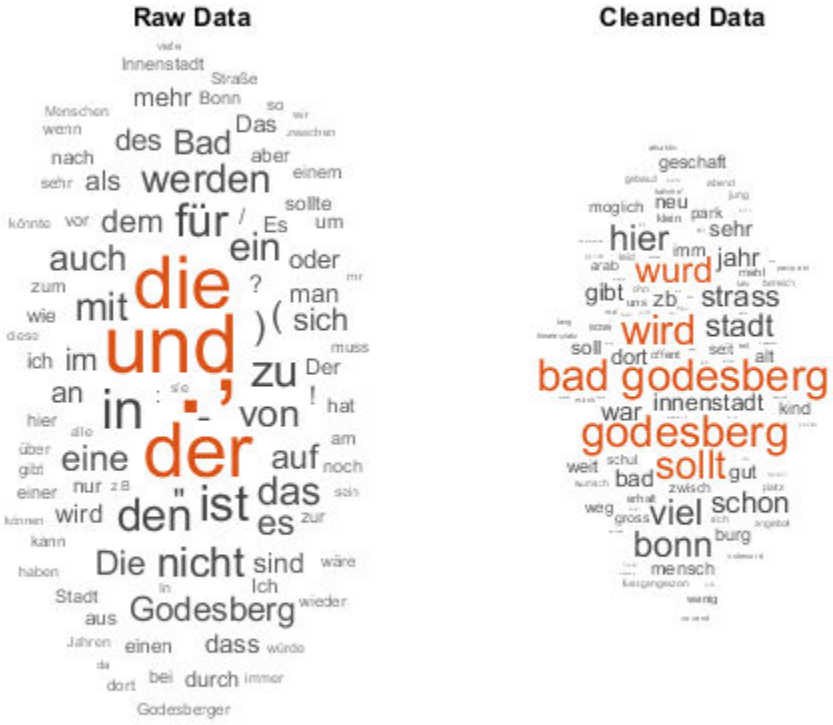
```
documents = normalizeWords(documents);
documents(1:10)

ans =
  10×1 tokenizedDocument:

    35 tokens: gebiet zwisch reuterstrass , bundeskanzlerplatz , willy-brandt-alle , eduard-pflug
    33 tokens: vorhabenbezog bebauungsplan nr . 6522-1 ? didinkirica ? bundesstadt bonn , stadtbe
    27 tokens: gebiet zwisch alfred-bucherer-strass , sebastianstrass fussweg zwisch rockumstrass
    81 tokens: vergang jahr fuhrt klimawandel vielzahl folg umwelt , wirtschaft mensch . stadt ge
    15 tokens: schaffung planungsrecht bau neu famili - , schul - sportschwimmbad flach nordlich
    57 tokens: begleit burgerbeteil konzepterstell neu schwimmbad soll folgend them beteil geb :
    40 tokens: gebietskuliss grun c soll freiraum zukunft sinn naherhol , landwirtschaft natursch
    32 tokens: entlast putzch / bechlinghov durchgangsverkehr verbind geplant anschlussstell maar
    19 tokens: areal ehemal landwirtschaftskamm sowi angrenz stadtisch flach stadtbezirk beuel ,
    25 tokens: areal herbert-rabius-strass stadtbezirk beuel , ortsteil beuel-mitt soll vorhabenb
```

Erase the punctuation using the `erasePunctuation` function.

```
documents = erasePunctuation(documents);
documents(1:10)

ans =
  10×1 tokenizedDocument:

    27 tokens: gebiet zwisch reuterstrass bundeskanzlerplatz willybrandtalle eduardpflugerstrass
    25 tokens: vorhabenbezog bebauungsplan nr 65221 didinkirica bundesstadt bonn stadtbezirk bonn
    22 tokens: gebiet zwisch alfredbuchererstrass sebastianstrass fussweg zwisch rockumstrass end
    64 tokens: vergang jahr fuhrt klimawandel vielzahl folg umwelt wirtschaft mensch stadt gemein
    11 tokens: schaffung planungsrecht bau neu famili schul sportschwimmbad flach nordlich heizk
    41 tokens: begleit burgerbeteil konzepterstell neu schwimmbad soll folgend them beteil geb di
    31 tokens: gebietskuliss grun c soll freiraum zukunft sinn naherhol landwirtschaft naturschut
    27 tokens: entlast putzch bechlinghov durchgangsverkehr verbind geplant anschlussstell maarst
    18 tokens: areal ehemal landwirtschaftskamm sowi angrenz stadtisch flach stadtbezirk beuel or
    19 tokens: areal herbertrabiusstrass stadtbezirk beuel ortsteil beuelmitt soll vorhabenbezog
```

Visualize the raw and cleaned data in word clouds.

```
figure
subplot(1,2,1)
wordcloud(documentsRaw);
title("Raw Data")

subplot(1,2,2)
wordcloud(documents);
title("Cleaned Data")
```



Create Preprocessing Function

Creating a function that performs preprocessing can be useful to prepare different collections of text data in the same way. For example, you can use a function to preprocess new data using the same steps as the training data.

Create a function which tokenizes and preprocesses the text data to use for analysis. The function `preprocessGermanText`, listed at the end of the example, performs these steps:

1. Tokenize the text using `tokenizedDocument`.
2. Replace the multiword phrase `["Bad" "Godesberg"]` with `"Bad Godesberg"`.
3. Remove a list of stop words (such as „der“, „die“, and „das“) using `removeStopWords`.
4. Normalize the words using `normalizeWords`.
5. Erase punctuation using `erasePunctuation`.

Remove the empty documents after preprocessing using the `removeEmptyDocuments` function. Removing documents after using a preprocessing function makes it easier to remove corresponding data such as labels from other sources.

In this example, use the preprocessing function `preprocessGermanText`, listed at the end of the example, to prepare the text data.

```
documents = preprocessGermanText(textData);
documents(1:5)
```

```
ans =
  5×1 tokenizedDocument:

    27 tokens: gebiet zwisch reuterstrass bundeskanzlerplatz willybrandtalle eduardpflugerstrass
    25 tokens: vorhabenbezog bebauungsplan nr 65221 didinkirica bundesstadt bonn stadtbezirk bonn
    22 tokens: gebiet zwisch alfredbucherstrass sebastianstrass fussweg zwisch rockumstrass end
    64 tokens: vergang jahr fuhrt klimawandel vielzahl folg umwelt wirtschaft mensch stadt gemein
    11 tokens: schaffung planungsrecht bau neu famili schul sportschwimmbad flach nordlich heizk
```

Remove the empty documents using the `removeEmptyDocuments` function.

```
documents = removeEmptyDocuments(documents);
```

Fit Topic Model

Fit a latent Dirichlet allocation (LDA) topic model to the data. An LDA model discovers underlying topics in a collection of documents and infers word probabilities in topics.

To fit an LDA model to the data, you first must create a bag-of-words model. A bag-of-words model (also known as a term-frequency counter) records the number of times that words appear in each document of a collection. Create a bag-of-words model using `bagOfWords`.

```
bag = bagOfWords(documents);
```

Remove the empty documents from the bag-of-words model.

```
bag = removeEmptyDocuments(bag);
```

Fit an LDA model with seven topics using `fitlda`. To suppress the verbose output, set `'Verbose'` to `0`.

```
numTopics = 7;
mdl = fitlda(bag,numTopics,'Verbose',0);
```

Visualize the first four topics using word clouds.

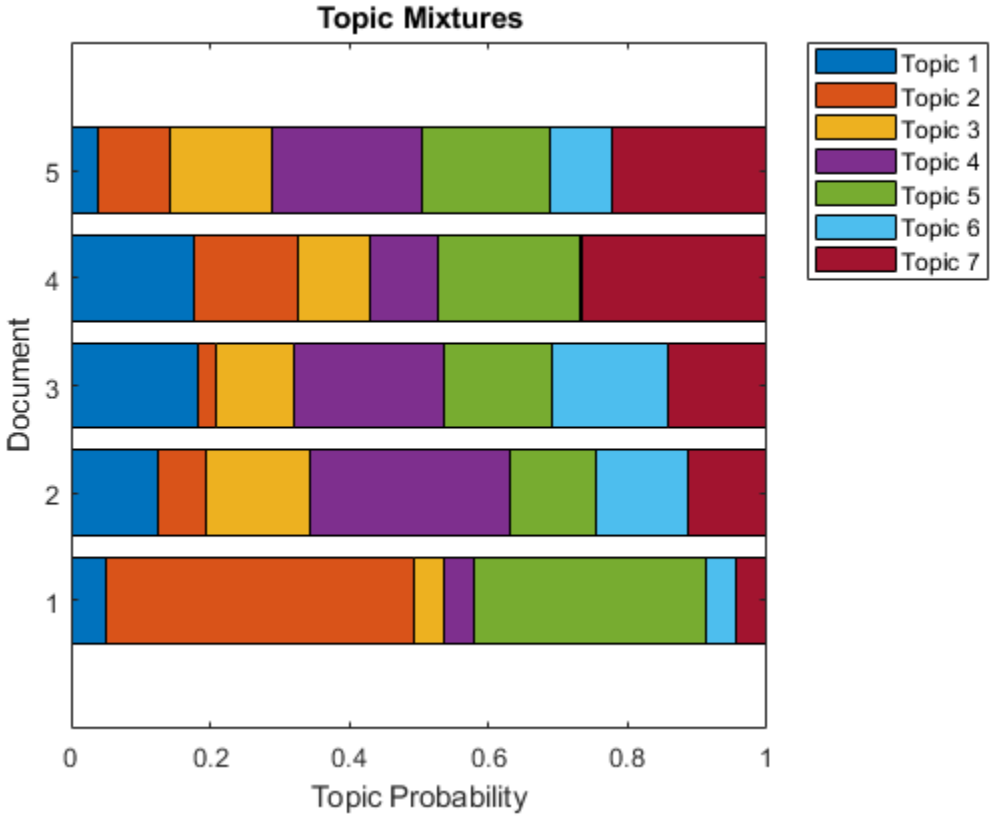
```
figure
for i = 1:4
    subplot(2,2,i)
    wordcloud(mdl,i);
    title("Topic " + i)
end
```

Visualize multiple topic mixtures using stacked bar charts. View five input documents at random and visualize the corresponding topic mixtures.

```
numDocuments = numel(documents);
idx = randperm(numDocuments,5);
documents(idx)
```

```
ans =
  5×1 tokenizedDocument:

      4 tokens: gastronom angebot sollt verbessert
     82 tokens: grunflach dietrichglaunerstrass rand dorfplatz entlang fussweg mehlem bach entlan
    116 tokens: sportplatz plittersdorf kommt leid regelmass unschon vorfall einsehbar umfeld her
     64 tokens: mainz strass bereich geschaft kirch uberwieg beidseit zugeparkt unschon sond fuss
     50 tokens: "1" "bezirksverodnet" "sollt" "kulturburgermeist" "gewahlt" "hatt" "aufgab" "vert
```

```
topicMixtures = transform(mdl,documents(idx));

figure
barh(topicMixtures(1:5,:),'stacked')
xlim([0 1])
title("Topic Mixtures")
xlabel("Topic Probability")
ylabel("Document")
legend("Topic " + string(1:numTopics),'Location','northeastoutside')
```



Example Preprocessing Function

The function `preprocessGermanText`, performs these steps:

1. Tokenize the text using `tokenizedDocument`.
2. Replace the multiword phrase `["Bad" "Godesberg"]` with `"Bad Godesberg"`.
3. Remove a list of stop words (such as „der“, „die“, and „das“) using `removeStopWords`.
4. Normalize the words using `normalizeWords`.
5. Erase punctuation using `erasePunctuation`.

```
function documents = preprocessGermanText(textData)

% Tokenize the text.
documents = tokenizedDocument(textData);

% Replace multiword phrases
old = ["Bad" "Godesberg"];
new = "Bad Godesberg";
documents = replaceNgrams(documents,old,new);

% Remove a list of stop words.
documents = removeStopWords(documents);

% Normalize the words.
documents = normalizeWords(documents);
```

```
% Erase the punctuation.
documents = erasePunctuation(documents);

end
```

See Also

`addPartOfSpeechDetails` | `normalizeWords` | `removeStopWords` | `stopWords` | `tokenDetails` | `tokenizedDocument`

More About

- “Language Considerations” on page 4-2
- “Create Simple Text Model for Classification” on page 2-2
- “Analyze Text Data Using Topic Models” on page 2-13
- “Analyze Text Data Using Multiword Phrases” on page 2-7
- “Analyze Text Data Containing Emojis” on page 2-32
- “Train a Sentiment Classifier” on page 2-51
- “Classify Text Data Using Deep Learning” on page 2-65
- “Generate Text Using Deep Learning” (Deep Learning Toolbox)

Korean Language Support

This topic summarizes the Text Analytics Toolbox features that support Korean text.

Tokenization

The `tokenizedDocument` function automatically detects Korean input. Alternatively, set the `'Language'` option in `tokenizedDocument` to `'ko'`. This option specifies the language details of the tokens. To view the language details of the tokens, use `tokenDetails`. These language details determine the behavior of the `removeStopWords`, `addPartOfSpeechDetails`, `normalizeWords`, `addSentenceDetails`, and `addEntityDetails` functions on the tokens.

To specify additional MeCab options for tokenization, create a `mecabOptions` object. To tokenize using the specified MeCab tokenization options, use the `'TokenizeMethod'` option of `tokenizedDocument`.

Part of Speech Details

The `tokenDetails` function, by default, includes part of speech details with the token details.

Named Entity Recognition

The `tokenDetails` function, by default, includes entity details with the token details.

Stop Words

To remove stop words from documents according to the token language details, use `removeStopWords`. For a list of Korean stop words set the `'Language'` option in `stopWords` to `'ko'`.

Lemmatization

To lemmatize tokens according to the token language details, use `normalizeWords` and set the `'Style'` option to `'lemma'`.

Language-Independent Features

Word and N-Gram Counting

The `bagOfWords` and `bagOfNgrams` functions support `tokenizedDocument` input regardless of language. If you have a `tokenizedDocument` array containing your data, then you can use these functions.

Modeling and Prediction

The `fitlda` and `fitlsa` functions support `bagOfWords` and `bagOfNgrams` input regardless of language. If you have a `bagOfWords` or `bagOfNgrams` object containing your data, then you can use these functions.

The `trainWordEmbedding` function supports `tokenizedDocument` or file input regardless of language. If you have a `tokenizedDocument` array or a file containing your data in the correct format, then you can use this function.

See Also

`addEntityDetails` | `addLanguageDetails` | `addPartOfSpeechDetails` | `normalizeWords` | `removeStopWords` | `stopWords` | `tokenDetails` | `tokenizedDocument`

More About

- “Language Considerations” on page 4-2

Language-Independent Features

Word and N-Gram Counting

The `bagOfWords` and `bagOfNgrams` functions support `tokenizedDocument` input regardless of language. If you have a `tokenizedDocument` array containing your data, then you can use these functions.

Modeling and Prediction

The `fitlda` and `fitlsa` functions support `bagOfWords` and `bagOfNgrams` input regardless of language. If you have a `bagOfWords` or `bagOfNgrams` object containing your data, then you can use these functions.

The `trainWordEmbedding` function supports `tokenizedDocument` or file input regardless of language. If you have a `tokenizedDocument` array or a file containing your data in the correct format, then you can use this function.

See Also

`addLanguageDetails` | `addSentenceDetails` | `bagOfNgrams` | `bagOfWords` | `fitlda` | `fitlsa` | `normalizeWords` | `removeWords` | `stopWords` | `tokenizedDocument` | `wordcloud`

More About

- “Text Data Preparation”
- “Modeling and Prediction”
- “Display and Presentation”
- “Japanese Language Support” on page 4-6
- “Analyze Japanese Text Data” on page 4-11
- “German Language Support” on page 4-21
- “Analyze German Text Data” on page 4-26

Glossary

Text Analytics Glossary

This section provides a list of terms used in text analytics.

Documents and Tokens

Term	Definition	More Information
Bigram	Two tokens in succession. For example, `["New" "York"]`.	`bagOfNgrams`
Complex token	A token with complex structure. For example, an email address or a hash tag.	`tokenDetails`
Context	Tokens or characters that surround a given token.	`context`
Corpus	A collection of documents.	`tokenizedDocument`
Document	A single observation of text data. For example, a report, a tweet, or an article.	`tokenizedDocument`
Grapheme	A human readable character. A grapheme can consist of multiple Unicode code points. For example, "a", "□□", or "語".	`splitGraphemes`
N-gram	*N* tokens in succession.	`bagOfNgrams`
Part of speech	Categories of words used in grammatical structure. For example, "noun", "verb", and "adjective".	`addPartOfSpeechDetails`
Token	A string of characters representing a unit of text data, also known as a "unigram". For example, a word, number, or email address.	`tokenizedDocument`
Token details	Information about the token. For example, type, language, or part-of-speech details.	`tokenDetails`
Token types	The category of the token. For example, "letters", "punctuation", or "email address".	`tokenDetails`
Tokenized document	A document split into tokens.	`tokenizedDocument`
Trigram	Three tokens in succession. For example, `["The" "United" "States"]`	`bagOfNgrams`
Vocabulary	Unique words or tokens in a corpus or model.	`tokenizedDocument`

Preprocessing

Term	Definition	More Information
Normalize	Reduce words to a root form. For example, reduce the word "walking" to "walk" using stemming or lemmatization.	`normalizeWords`
Lemmatize	Reduce words to a dictionary word (the lemma form). For example, reduce the words "running" and "ran" to "run".	`normalizeWords`
Stem	Reduce words by removing inflections. The reduced word is not necessarily a real word. For example, the Porter stemmer reduces the words "happy" and "happiest" to "happi".	`normalizeWords`
Stop words	Words commonly removed before analysis. For example "and", "of", and "the".	`removeStopWords`

Modeling and Prediction

Bag-of-Words

Term	Definition	More Information
Bag-of-n-grams model	A model that records the number of times that n-grams appear in each document of a corpus.	`bagOfNgrams`
Bag-of-words model	A model that records the number of times that words appear in each document of a collection.	`bagOfWords`
Term frequency count matrix	A matrix of the frequency counts of words occurring in a collection of documents corresponding to a given vocabulary. This matrix is the underlying data of a bag-of-words model.	`bagOfWords`
Term Frequency–Inverse Document Frequency (tf-idf) matrix	A statistical measure based on the word frequency counts in documents and the proportion of documents containing the words in the corpus.	`tfidf`

Latent Dirichlet Allocation

Term	Definition	More Information
Corpus topic probabilities	The probabilities of observing each topic in the corpus used to fit the LDA model.	`ldaModel`
Document topic probabilities	The probabilities of observing each topic in each document used to fit the LDA model. Equivalently, the topic mixtures of the training documents.	`ldaModel`
Latent Dirichlet allocation (LDA)	A generative statistical topic model that infers topic probabilities in documents and word probabilities in topics.	`fitlda`
Perplexity	A statistical measure of how well a model describes the given data. A lower perplexity indicates a better fit.	`logp`
Topic	A distribution of words, characterized by the "topic word probabilities".	`ldaModel`
Topic concentration	The concentration parameter of the underlying Dirichlet distribution of the corpus topics mixtures.	`ldaModel`
Topic mixture	The probabilities of topics in a given document.	`transform`
Topic word probabilities	The probabilities of words in a given topic.	`ldaModel`
Word concentration	The concentration parameter of the underlying Dirichlet distribution of the topics.	`ldaModel`

Latent Semantic Analysis

Term	Definition	More Information
Component weights	The singular values of the decomposition, squared.	`lsaModel`
Document scores	The score vectors in lower dimensional space of the documents used to fit the LSA model.	`transform`
Latent semantic analysis (LSA)	A dimension reducing technique based on principal component analysis (PCA).	`fitlsa`

Term	Definition	More Information
Word scores	The scores of each word in each component of the LSA model.	`lsaModel`

Word Embeddings

Term	Definition	More Information
Word embedding	A model, popularized by the word2vec, GloVe, and fastText libraries, that maps words in a vocabulary to real vectors.	`wordEmbedding`
Word embedding layer	A deep learning network layer that learns a word embedding during training.	`wordEmbeddingLayer`
Word encoding	A model that maps words to numeric indices.	`wordEncoding`

Visualization

Term	Definition	More Information
Text scatter plot	A scatter plot with words plotted at specified coordinates instead of markers.	`textscatter`
Word cloud	A chart that displays words with sizes corresponding to numeric data, usually frequency counts.	`wordcloud`

See Also

`addPartOfSpeechDetails` | `bagOfNgrams` | `bagOfWords` | `fitlda` | `normalizeWords` | `removeStopWords` | `textscatter` | `tokenDetails` | `tokenizedDocument` | `wordEmbedding` | `wordEmbeddingLayer` | `wordEncoding` | `wordcloud`

More About

- “Try Text Analytics in 10 Lines of Code”
- “Import Text Data into MATLAB”
- “Create Simple Preprocessing Function”
- “Get Started with Topic Modeling”
- “Visualize Text Data Using Word Clouds” on page 3-2