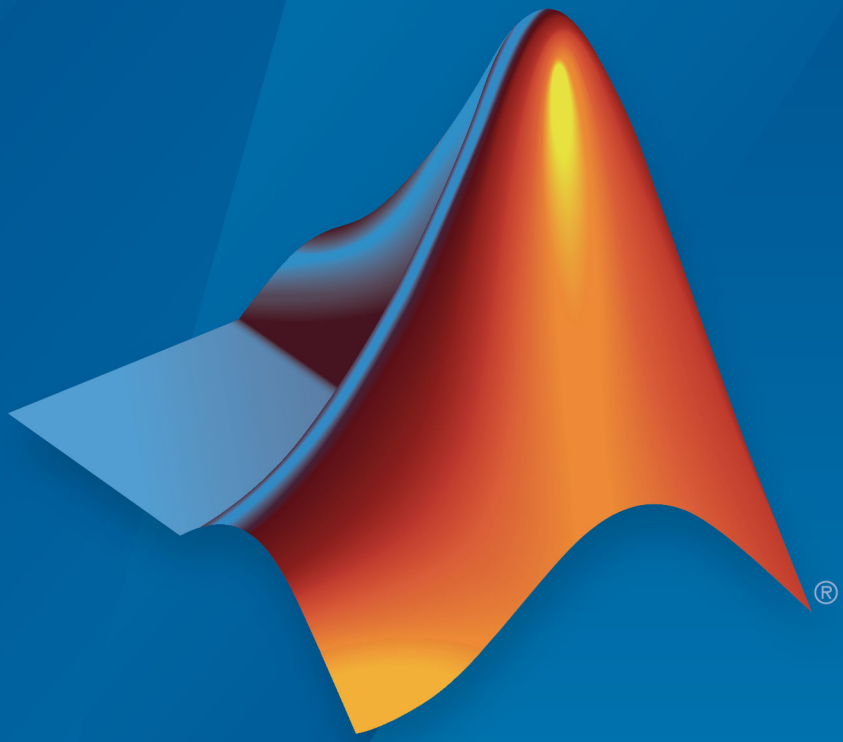
Text Analytics Toolbox™

User's Guide



MATLAB®

R2019b

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)

Contents

1 Text Data Preparation

2 Modeling and Prediction

3 Display and Presentation

4 Language Support

5 Glossary

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.

      "
```

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,

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

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

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="svg_link navbar-brand" href="https://www.mathworks.com?s_tid=gn_logo"><IM
    <A class="mwa-nav_login" href="https://www.mathworks.com/login?uri=http://www.mathw
    <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">
    <A href="https://www.mathworks.com/store?s_cid=store_top_nav&amp;s_tid=gn_store">Ho
```

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

View the extracted text of the first 10 hyperlinks.

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

```
ans = 10×1 string array
    ""
```

```
    "Sign In"
    "Products"
    "Solutions"
    "Academia"
    "Support"
    "Community"
    "Events"
    "Contact Us"
    "How to Buy"
```

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 array
    "https://www.mathworks.com?s_tid=gn_logo"
    "https://www.mathworks.com/login?uri=http://www.mathworks.com/help/textanalytics/ir
    "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/store?s_cid=store_top_nav&s_tid=gn_store"
```

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 using the `readtable` function and view the first few rows of the table.

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

```
ans=8×16 table
            Time             event_id          state              event_type          da
    _____________________    __________    _______________    ____________________    __
```

```
    22-Jul-2016 16:10:00    6.4433e+05    "MISSISSIPPI"       "Thunderstorm Wind"
    15-Jul-2016 17:15:00    6.5182e+05    "SOUTH CAROLINA"    "Heavy Rain"
    15-Jul-2016 17:25:00    6.5183e+05    "SOUTH CAROLINA"    "Thunderstorm Wind"
    16-Jul-2016 12:46:00    6.5183e+05    "NORTH CAROLINA"    "Thunderstorm Wind"
    15-Jul-2016 14:28:00    6.4332e+05    "MISSOURI"          "Hail"
    15-Jul-2016 16:31:00    6.4332e+05    "ARKANSAS"          "Thunderstorm Wind"
    15-Jul-2016 16:03:00    6.4343e+05    "TENNESSEE"         "Thunderstorm Wind"
    15-Jul-2016 17:27:00    6.4344e+05    "TENNESSEE"         "Hail"
```

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

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

```
ans = 10×1 string array
    "Large tree down between Plantersville and Nettleton."
    "One to two feet of deep standing water developed on a street on the Winthrop Unive
    "NWS Columbia relayed a report of trees blown down along Tom Hall St."
    "Media reported two trees blown down along I-40 in the Old Fort area."
    ""
    "A few tree limbs greater than 6 inches down on HWY 18 in Roseland."
    "Awning blown off a building on Lamar Avenue. Multiple trees down near the intersec
    "Quarter size hail near Rosemark."
    "Tin roof ripped off house on Old Memphis Road near Billings Drive. Several large t
    "Powerlines down at Walnut Grove and Cherry Lane roads."
```

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 examples 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.

```
fds = fileDatastore('exampleSonnet*.txt','ReadFcn',@extractFileText)
```

```
fds =
  FileDatastore with properties:

                      Files: {
                             ' ...\Documents\MATLAB\examples\textanalytics-ex15735454
```

```
                                ' ...\Documents\MATLAB\examples\textanalytics-ex15735454\
                                ' ...\Documents\MATLAB\examples\textanalytics-ex15735454\
                                 ... and 1 more
                                }
                    UniformRead: 0
                        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 = 4×1 string array
    "  From fairest creatures we desire increase,↵  That thereby beauty's rose might ne
    "  When forty winters shall besiege thy brow,↵  And dig deep trenches in thy beauty
    "  Look in thy glass and tell the face thou viewest↵  Now is the time that face sho
    "  Unthrifty loveliness, why dost thou spend↵  Upon thy self thy beauty's legacy?↵
```

See Also

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

Related Examples

- “Prepare Text Data for Analysis” on page 1-12
- “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 2-35
- “Analyze Text Data Using Topic Models” on page 2-18
- “Analyze Text Data Using Multiword Phrases” on page 2-9

- “Classify Text Data Using Deep Learning” on page 2-57
- “Train a Sentiment Classifier” on page 2-47

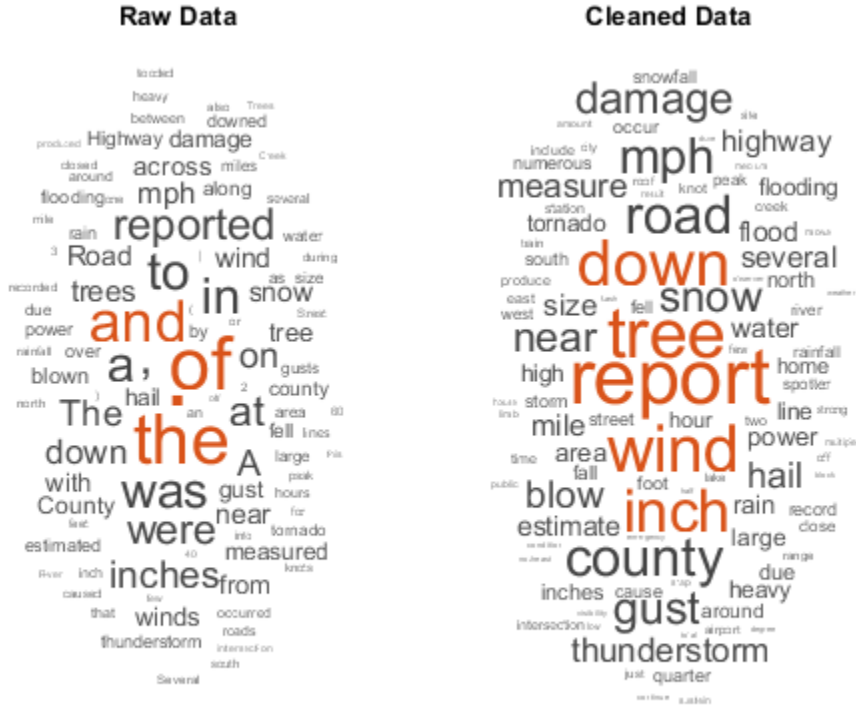
Prepare Text Data for Analysis

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 weather reports, and a preprocessed version of the same text data.



Load and Extract Text Data

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

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

Extract the text data from the field `event_narrative`, and the label data from the field `event_type`.

```
textData = data.event_narrative;
labels = data.event_type;
textData(1:10)
```

```
ans = 10×1 string array
    "Large tree down between Plantersville and Nettleton."
    "One to two feet of deep standing water developed on a street on the Winthrop Unive
    "NWS Columbia relayed a report of trees blown down along Tom Hall St."
    "Media reported two trees blown down along I-40 in the Old Fort area."
    ""
    "A few tree limbs greater than 6 inches down on HWY 18 in Roseland."
    "Awning blown off a building on Lamar Avenue. Multiple trees down near the intersec
    "Quarter size hail near Rosemark."
    "Tin roof ripped off house on Old Memphis Road near Billings Drive. Several large t
    "Powerlines down at Walnut Grove and Cherry Lane roads."
```

Create Tokenized Documents

Create an array of tokenized documents.

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

```
ans =
  10×1 tokenizedDocument:

     8 tokens: Large tree down between Plantersville and Nettleton .
    39 tokens: One to two feet of deep standing water developed on a street on the Wint
    14 tokens: NWS Columbia relayed a report of trees blown down along Tom Hall St .
    14 tokens: Media reported two trees blown down along I-40 in the Old Fort area .
     0 tokens:
    15 tokens: A few tree limbs greater than 6 inches down on HWY 18 in Roseland .
    20 tokens: Awning blown off a building on Lamar Avenue . Multiple trees down near t
     6 tokens: Quarter size hail near Rosemark .
    21 tokens: Tin roof ripped off house on Old Memphis Road near Billings Drive . Seve
    10 tokens: Powerlines down at Walnut Grove and Cherry Lane roads .
```

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:

     6 tokens: Large tree down Plantersville Nettleton .
    20 tokens: two feet deep standing water developed street Winthrop University campus
    11 tokens: NWS Columbia relayed report trees blown down Tom Hall St .
    11 tokens: Media reported two trees blown down I-40 Old Fort area .
     0 tokens:
    11 tokens: few tree limbs greater 6 inches down HWY 18 Roseland .
    15 tokens: Awning blown off building Lamar Avenue . Multiple trees down near inters
     6 tokens: Quarter size hail near Rosemark .
    18 tokens: Tin roof ripped off house Old Memphis Road near Billings Drive . Several
     8 tokens: Powerlines down Walnut Grove Cherry Lane roads .
```

Lemmatize the words using `normalizeWords`.

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

ans =
  10×1 tokenizedDocument:

     6 tokens: large tree down plantersville nettleton .
    20 tokens: two foot deep standing water develop street winthrop university campus
    11 tokens: nws columbia relay report tree blow down tom hall st .
    11 tokens: medium report two tree blow down i-40 old fort area .
     0 tokens:
    11 tokens: few tree limb great 6 inch down hwy 18 roseland .
    15 tokens: awning blow off building lamar avenue . multiple tree down near intersec
     6 tokens: quarter size hail near rosemark .
    18 tokens: tin roof rip off house old memphis road near billings drive . several la
     8 tokens: powerlines down walnut grove cherry lane road .
```

Erase the punctuation from the documents.

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

ans =
  10×1 tokenizedDocument:
```

```
     5 tokens: large tree down plantersville nettleton
    18 tokens: two foot deep standing water develop street winthrop university campus i
    10 tokens: nws columbia relay report tree blow down tom hall st
    10 tokens: medium report two tree blow down i40 old fort area
     0 tokens:
    10 tokens: few tree limb great 6 inch down hwy 18 roseland
    13 tokens: awning blow off building lamar avenue multiple tree down near intersecti
     5 tokens: quarter size hail near rosemark
    16 tokens: tin roof rip off house old memphis road near billings drive several larg
     7 tokens: powerlines down walnut grove cherry lane road
```

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:

     5 tokens: large tree down plantersville nettleton
    18 tokens: two foot deep standing water develop street winthrop university campus i
     9 tokens: nws columbia relay report tree blow down tom hall
    10 tokens: medium report two tree blow down i40 old fort area
     0 tokens:
     8 tokens: few tree limb great inch down hwy roseland
    13 tokens: awning blow off building lamar avenue multiple tree down near intersecti
     5 tokens: quarter size hail near rosemark
    16 tokens: tin roof rip off house old memphis road near billings drive several larg
     7 tokens: powerlines down walnut grove cherry lane road
```

Create Bag-of-Words Model

Create a bag-of-words model.

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

```
cleanedBag =
  bagOfWords with properties:

          Counts: [36176×18478 double]
      Vocabulary: [1×18478 string]
```

```
        NumWords: 18478
    NumDocuments: 36176
```

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: [36176×6978 double]
      Vocabulary: [1×6978 string]
        NumWords: 6978
    NumDocuments: 36176
```

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: [28137×6978 double]
      Vocabulary: [1×6978 string]
        NumWords: 6978
    NumDocuments: 28137
```

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 `preprocessWeatherNarratives`, 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 `preprocessWeatherNarratives` to prepare the text data.

```
newText = "A tree is downed outside Apple Hill Drive, Natick";
newDocuments = preprocessWeatherNarratives(newText)
```

```
newDocuments =
  tokenizedDocument:

   7 tokens: tree down outside apple hill drive natick
```

Compare with Raw Data

Compare the preprocessed data with the raw data.

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

```
rawBag =
  bagOfWords with properties:

          Counts: [36176×23302 double]
      Vocabulary: [1×23302 string]
        NumWords: 23302
    NumDocuments: 36176
```

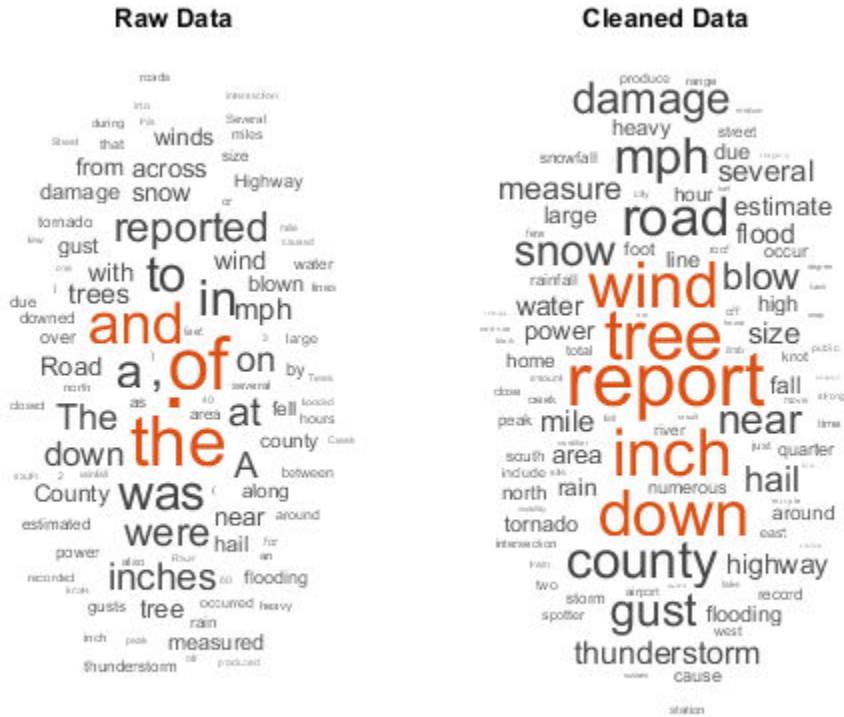
Calculate the reduction in data.

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

```
reduction = 0.7005
```

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 `preprocessWeatherNarratives`, 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 = preprocessWeatherNarratives(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 2-35
- “Analyze Text Data Using Topic Models” on page 2-18
- “Analyze Text Data Using Multiword Phrases” on page 2-9
- “Classify Text Data Using Deep Learning” on page 2-57
- “Train a Sentiment Classifier” on page 2-47

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"/><ME
    " "
    <BODY id="responsive_offcanvas"><!-- Mobile TopNav: Start --><DIV class="header vis
```

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,
```

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

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

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"><IM
    <A class="mwa-nav_login" href="https://www.mathworks.com/login?uri=http://www.mathw
    <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">
    <A href="https://www.mathworks.com/store?s_cid=store_top_nav&amp;s_tid=gn_store">Ho
    <A href="https://www.mathworks.com/company/aboutus/contact_us.html?s_tid=gn_cntus">
    <A href="https://www.mathworks.com/store?s_cid=store_top_nav&amp;s_tid=gn_store">Ho
    <A class="mwa-nav_login" href="https://www.mathworks.com/login?uri=http://www.mathw
    <A class="svg_link pull-left" href="https://www.mathworks.com?s_tid=gn_logo"><IMG a
    <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 1-12
- “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 2-18

- “Analyze Text Data Using Multiword Phrases” on page 2-9
- “Classify Text Data Using Deep Learning” on page 2-57
- “Train a Sentiment Classifier” on page 2-47

Correct Spelling Using Edit Distance Searchers

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

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.

Lemmatization with `normalizeWords` and `word2vec` requires correctly spelled words to work. 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 the Spell Checking Oriented Word Lists (SCOWL) from https://sourceforge.net/projects/wordlist/. Import the words from the downloaded data using the supporting function `scowlWordList`.

```
folderName = "scowl-2018.04.16";
maxSize = 60;
vocabulary = scowlWordList(folderName,'english',maxSize);
```

View the number of words in the vocabulary.

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

```
numWords = 98129
```

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 array
    "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 array
    "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 array
    "an"    "example"    "document"
```

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

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

```
words = 1×8 string array
    "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);
documentsCorrected = docfun(func,documents)
```

```
documentsCorrected =
  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);
documentsCorrected = docfun(func,documentsLower)
```

```
documentsCorrected =
  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

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

More About

- “Prepare Text Data for Analysis” on page 1-12
- “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 2-18
- “Analyze Text Data Using Multiword Phrases” on page 2-9
- “Classify Text Data Using Deep Learning” on page 2-57
- “Train a Sentiment Classifier” on page 2-47

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 event type of weather reports using text descriptions.

To reproduce the results of this example, set `rng` to `'default'`.

```
rng('default')
```

Load and Extract Text Data

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

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

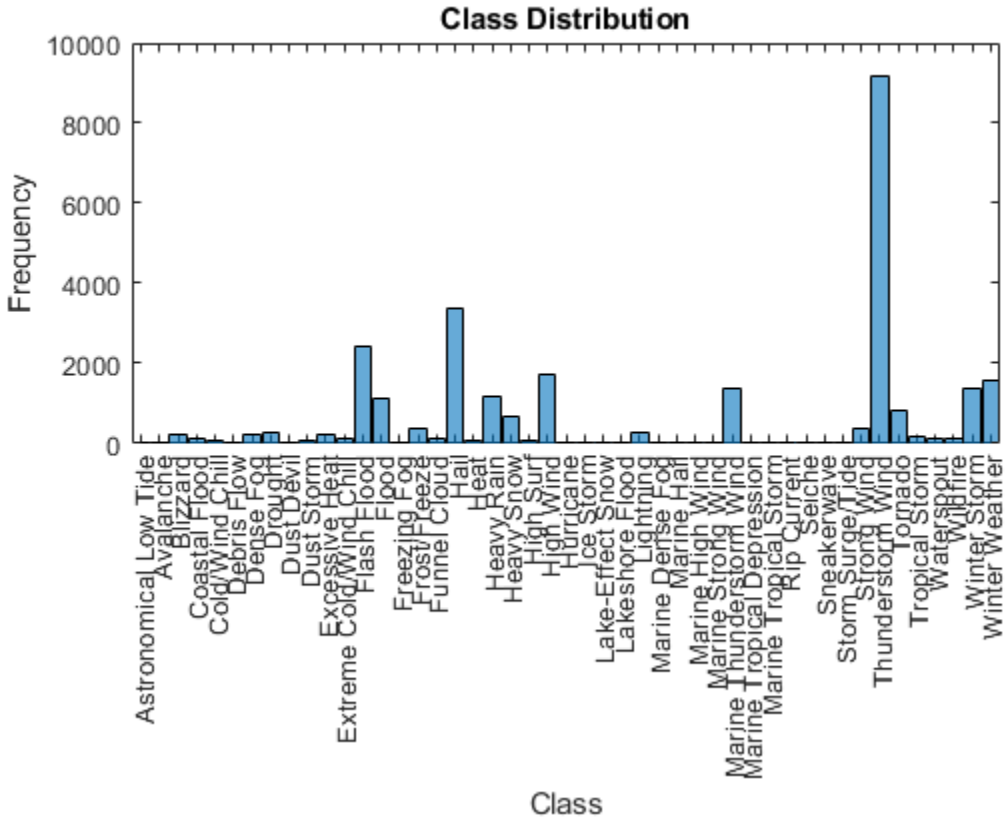
```
ans=8×16 table
            Time             event_id          state              event_type          da
    _____________________    _________    _______________    ____________________    __

    22-Jul-2016 16:10:00    6.4433e+05    "MISSISSIPPI"      "Thunderstorm Wind"
    15-Jul-2016 17:15:00    6.5182e+05    "SOUTH CAROLINA"   "Heavy Rain"
    15-Jul-2016 17:25:00    6.5183e+05    "SOUTH CAROLINA"   "Thunderstorm Wind"
    16-Jul-2016 12:46:00    6.5183e+05    "NORTH CAROLINA"   "Thunderstorm Wind"
    15-Jul-2016 14:28:00    6.4332e+05    "MISSOURI"         "Hail"
    15-Jul-2016 16:31:00    6.4332e+05    "ARKANSAS"         "Thunderstorm Wind"
    15-Jul-2016 16:03:00    6.4343e+05    "TENNESSEE"        "Thunderstorm Wind"
    15-Jul-2016 17:27:00    6.4344e+05    "TENNESSEE"        "Hail"
```

Remove rows with empty reports.

```
idx = strlength(data.event_narrative) == 0;
data(idx,:) = [];
```

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

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



The classes of the data are imbalanced, with several classes containing few observations. To ensure that you can partition the data so that the partitions contain observations for each class, remove any classes which appear fewer than ten times.

Get the frequency counts of the classes and their names from the histogram.

```
classCounts = h.BinCounts;
classNames = h.Categories;
```

Find the classes containing fewer than ten observations and remove these infrequent classes from the data.

```
idxLowCounts = classCounts < 10;
infrequentClasses = classNames(idxLowCounts);
idxInfrequent = ismember(data.event_type,infrequentClasses);
data(idxInfrequent,:) = [];
```

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

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

Extract the text data and labels from the tables.

```
textDataTrain = dataTrain.event_narrative;
textDataTest = dataTest.event_narrative;
YTrain = dataTrain.event_type;
YTest = dataTest.event_type;
```

Prepare Text Data for Analysis

Create a function which tokenizes and preprocesses the text data so it can be used for analysis. The function `preprocessWeatherNarratives`, 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 `preprocessWeatherNarratives` to prepare the text data.

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

```
ans =
  5×1 tokenizedDocument:

     5 tokens: large tree down plantersville nettleton
    18 tokens: two foot deep standing water develop street winthrop university campus
     9 tokens: nws columbia relay report tree blow down tom hall
    10 tokens: medium report two tree blow down i40 old fort area
     8 tokens: few tree limb great inch down hwy roseland
```

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

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

```
bag =
  bagOfWords with properties:

          Counts: [25316×17533 double]
      Vocabulary: [1×17533 string]
        NumWords: 17533
    NumDocuments: 25316
```

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: [25315×6535 double]
      Vocabulary: [1×6535 string]
        NumWords: 6535
    NumDocuments: 25315
```

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 =
  classreg.learning.classif.CompactClassificationECOC
      ResponseName: 'Y'
        ClassNames: [1×39 categorical]
    ScoreTransform: 'none'
    BinaryLearners: {741×1 cell}
      CodingMatrix: [39×741 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 = preprocessWeatherNarratives(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 = 0.8851
```

Predict Using New Data

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

```
str = [ ...
    "A large tree is downed and blocking traffic outside Apple Hill."
    "Damage to many car windshields in parking lot."
    "Lots of water damage to computer equipment inside the office."];
documentsNew = preprocessWeatherNarratives(str);
XNew = encode(bag,documentsNew);
labelsNew = predict(mdl,XNew)
```

```
labelsNew = 3×1 categorical array
     Thunderstorm Wind
     Thunderstorm Wind
     Flash Flood
```

Example Preprocessing Function

The function `preprocessWeatherNarratives`, 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 = preprocessWeatherNarratives(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 2-18
- “Analyze Text Data Using Multiword Phrases” on page 2-9
- “Analyze Text Data Containing Emojis” on page 2-35
- “Train a Sentiment Classifier” on page 2-47
- “Classify Text Data Using Deep Learning” on page 2-57
- “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.

N-Grams

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

To reproduce the results of this example, set `rng` to `'default'`.

```
rng('default')
```

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

```
filename = "weatherReports.csv";
data = readtable(filename,'TextType','String');
idx = strlength(data.event_narrative) == 0;
data(idx,:) = [];
```

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

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

```
ans = 5×1 string array
    "Large tree down between Plantersville and Nettleton."
    "One to two feet of deep standing water developed on a street on the Winthrop Unive
    "NWS Columbia relayed a report of trees blown down along Tom Hall St."
```

```
    "Media reported two trees blown down along I-40 in the Old Fort area."
    "A few tree limbs greater than 6 inches down on HWY 18 in Roseland."
```

Prepare Text Data for Analysis

Create a function which tokenizes and preprocesses the text data so it can be used for analysis. The function `preprocessWeatherNarratives` 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 `preprocessWeatherNarratives` to prepare the text data.

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

```
ans =
  5×1 tokenizedDocument:

     5 tokens: large tree down plantersville nettleton
    18 tokens: two foot deep standing water develop street winthrop university campus
     9 tokens: nws columbia relayed report tree blow down tom hall
    10 tokens: medium report two tree blow down i40 old fort area
     8 tokens: few tree limb great inch down hwy roseland
```

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`.

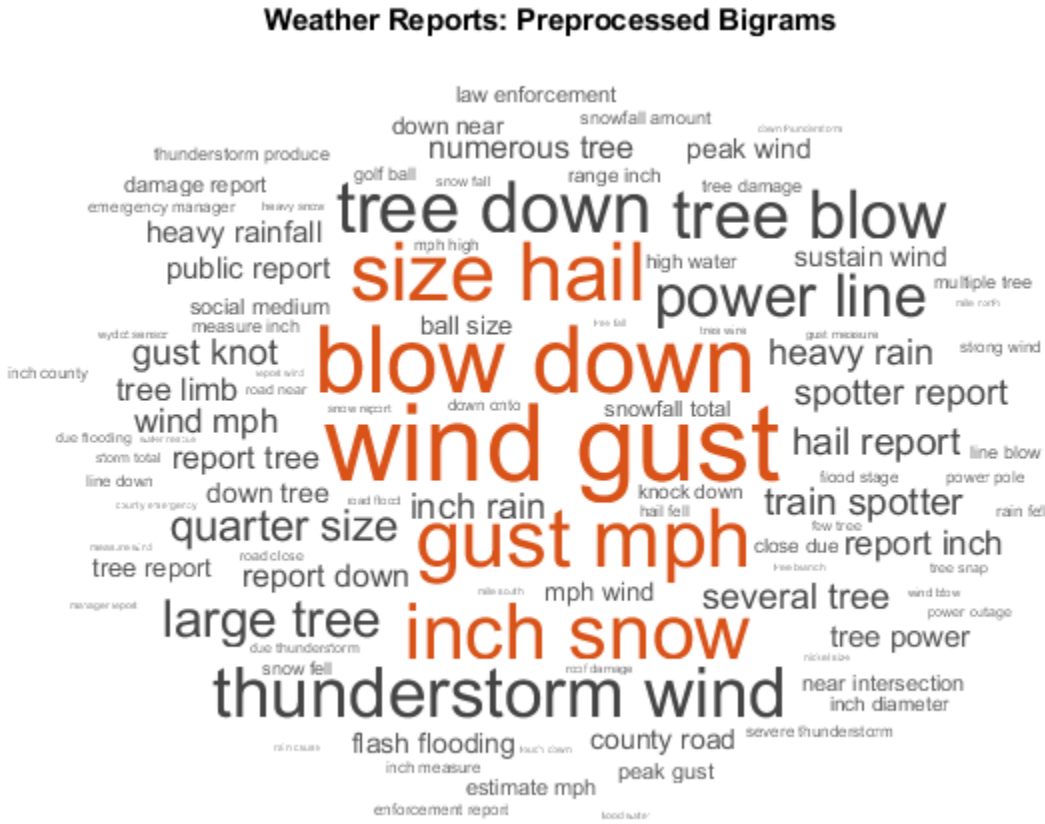
To count the n-grams of length 2 (bigrams), use `bagOfNgrams` with the default options.

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

```
bag =
  bagOfNgrams with properties:

          Counts: [28138×116734 double]
      Vocabulary: [1×18352 string]
          Ngrams: [116734×2 string]
    NgramLengths: 2
       NumNgrams: 116734
    NumDocuments: 28138
```

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

```
figure
wordcloud(bag);
title("Weather Reports: 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);
```

```
Initial topic assignments sampled in 0.892068 seconds.
=================================================================================
| Iteration  |  Time per  |  Relative  |  Training  |     Topic     |     Topic     |
|            | iteration  | change in  | perplexity | concentration | concentration |
```

```
|            | (seconds)  |   log(L)   |           |               |   iterations  |
=============================================================================================
|          0 |       2.66 |            |  2.022e+04 |         2.500 |             0 |
|          1 |       3.23 | 6.8447e-02 |  1.071e+04 |         2.500 |             0 |
|          2 |       3.23 | 2.1061e-03 |  1.051e+04 |         2.500 |             0 |
|          3 |       3.32 | 4.0862e-04 |  1.047e+04 |         2.500 |             0 |
|          4 |       3.25 | 2.7706e-04 |  1.044e+04 |         2.500 |             0 |
|          5 |       3.63 | 2.4722e-04 |  1.042e+04 |         2.500 |             0 |
|          6 |       3.40 | 2.5610e-04 |  1.039e+04 |         2.500 |             0 |
|          7 |       3.71 | 2.0857e-04 |  1.037e+04 |         2.500 |             0 |
|          8 |       3.49 | 7.9013e-05 |  1.036e+04 |         2.500 |             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-occuring 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`.

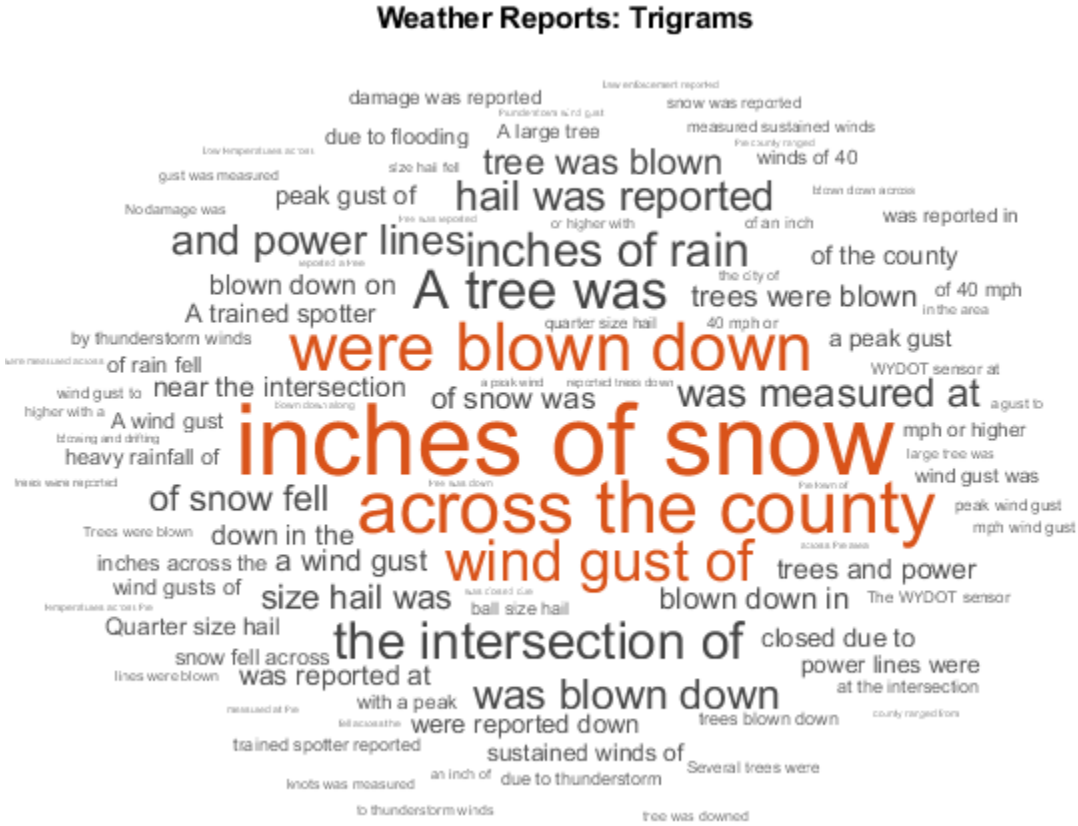
```
cleanTextData = erasePunctuation(textData);
documents = tokenizedDocument(cleanTextData);
```

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("Weather Reports: Trigrams")
```



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

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

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

    "inches"    "of"              "snow"           2075          3
    "across"    "the"             "county"         1318          3
    "were"      "blown"           "down"           1189          3
    "wind"      "gust"            "of"              934          3
    "A"         "tree"            "was"             860          3
    "the"       "intersection"    "of"              812          3
    "inches"    "of"              "rain"            739          3
    "hail"      "was"             "reported"        648          3
    "was"       "blown"           "down"            638          3
    "and"       "power"           "lines"           631          3
```

Example Preprocessing Function

The function `preprocessWeatherNarratives` 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] = preprocessWeatherNarratives(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 2-35
- “Analyze Text Data Using Topic Models” on page 2-18
- “Train a Sentiment Classifier” on page 2-47
- “Classify Text Data Using Deep Learning” on page 2-57
- “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.

To reproduce the results of this example, set `rng` to `'default'`.

```
rng('default')
```

Load and Extract Text Data

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

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

```
ans=8×16 table
            Time             event_id          state              event_type          da
    _____________________    _________    _______________    ____________________    __

    22-Jul-2016 16:10:00    6.4433e+05    "MISSISSIPPI"       "Thunderstorm Wind"
    15-Jul-2016 17:15:00    6.5182e+05    "SOUTH CAROLINA"    "Heavy Rain"
    15-Jul-2016 17:25:00    6.5183e+05    "SOUTH CAROLINA"    "Thunderstorm Wind"
    16-Jul-2016 12:46:00    6.5183e+05    "NORTH CAROLINA"    "Thunderstorm Wind"
    15-Jul-2016 14:28:00    6.4332e+05    "MISSOURI"          "Hail"
    15-Jul-2016 16:31:00    6.4332e+05    "ARKANSAS"          "Thunderstorm Wind"
    15-Jul-2016 16:03:00    6.4343e+05    "TENNESSEE"         "Thunderstorm Wind"
    15-Jul-2016 17:27:00    6.4344e+05    "TENNESSEE"         "Hail"
```

Extract the text data from the field `event_narrative`.

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

```
ans = 10×1 string array
    "Large tree down between Plantersville and Nettleton."
    "One to two feet of deep standing water developed on a street on the Winthrop Unive
    "NWS Columbia relayed a report of trees blown down along Tom Hall St."
    "Media reported two trees blown down along I-40 in the Old Fort area."
```

```
    ""
    "A few tree limbs greater than 6 inches down on HWY 18 in Roseland."
    "Awning blown off a building on Lamar Avenue. Multiple trees down near the intersec
    "Quarter size hail near Rosemark."
    "Tin roof ripped off house on Old Memphis Road near Billings Drive. Several large t
    "Powerlines down at Walnut Grove and Cherry Lane roads."
```

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:

     5 tokens: large tree down plantersville nettleton
    18 tokens: two foot deep standing water develop street winthrop university campus i
     9 tokens: nws columbia relay report tree blow down tom hall
    10 tokens: medium report two tree blow down i40 old fort area
     0 tokens:
```

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

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

```
bag =
  bagOfWords with properties:
```

```
          Counts: [36176×18469 double]
      Vocabulary: [1×18469 string]
        NumWords: 18469
    NumDocuments: 36176
```

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: [28137×6974 double]
      Vocabulary: [1×6974 string]
        NumWords: 6974
    NumDocuments: 28137
```

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 2-25. To suppress verbose output, set `'Verbose'` to 0.

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

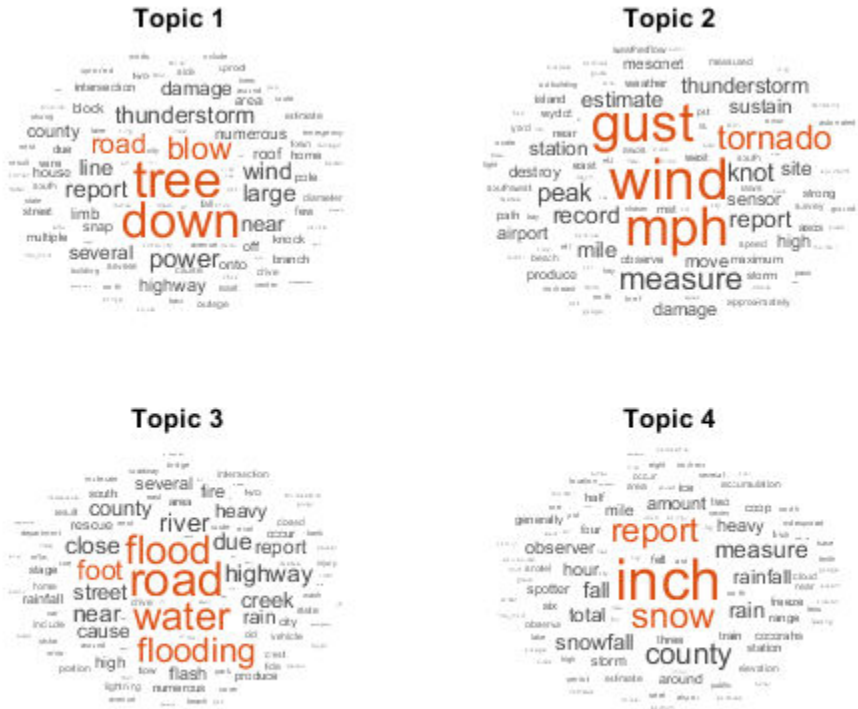
If you have a large dataset, then the stochastic approximate variational Bayes solver is usually better suited as it can fit a good model in fewer passes of the data. The default solver for `fitlda` (collapsed Gibbs sampling) can be more accurate at the cost of taking longer to run. To use stochastic approximate variational Bayes, set the `'Solver'` option to `'savb'`. For an example showing how to compare LDA solvers, see “Compare LDA Solvers” on page 2-30.

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
```

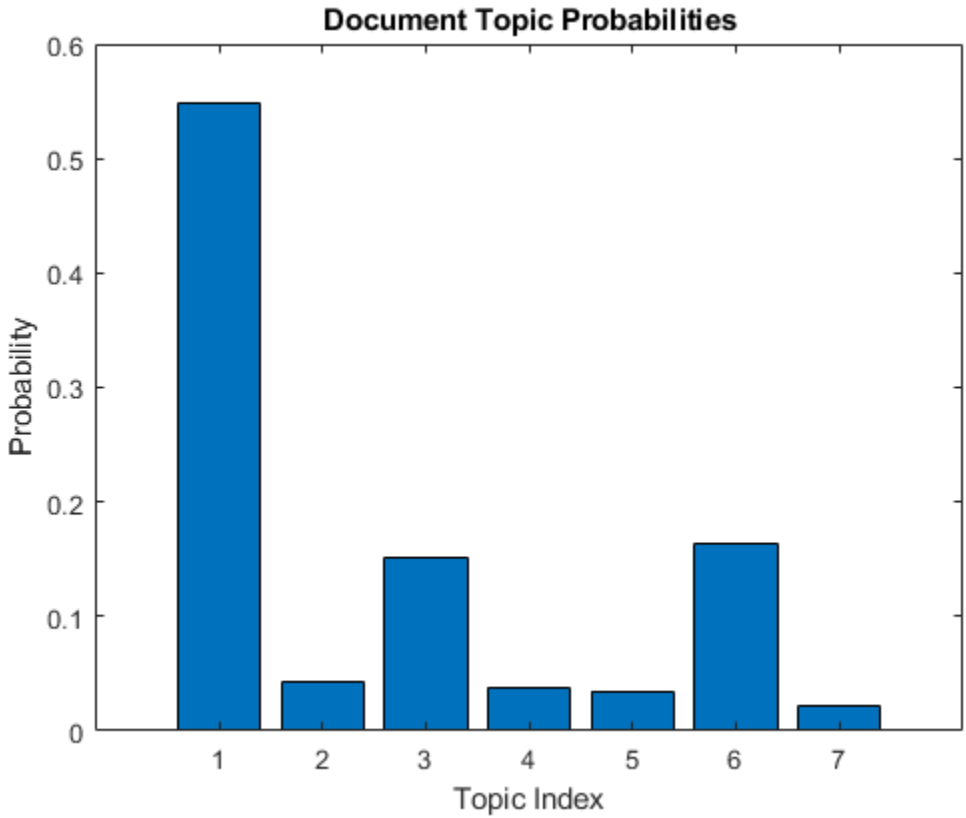


View Mixtures of Topics in Documents

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

```
newDocument = tokenizedDocument("A tree is downed outside Apple Hill Drive, Natick");
topicMixture = transform(mdl,newDocument);

figure
bar(topicMixture)
```

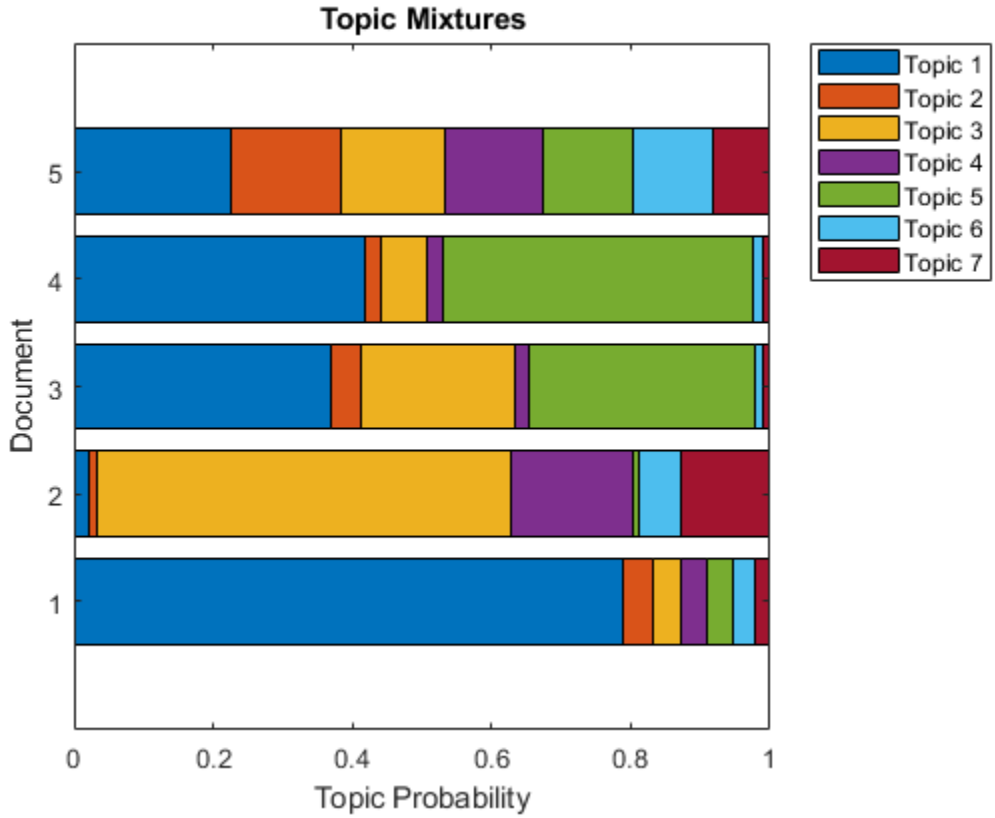
```
xlabel("Topic Index")
ylabel("Probability")
title("Document Topic Probabilities")
```



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 2-35
- “Analyze Text Data Using Multiword Phrases” on page 2-9
- “Train a Sentiment Classifier” on page 2-47
- “Classify Text Data Using Deep Learning” on page 2-57
- “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.

To reproduce the results of this example, set `rng` to `'default'`.

```
rng('default')
```

Extract and Preprocess Text Data

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

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

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

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

```
ans =
  5×1 tokenizedDocument:

     5 tokens: large tree down plantersville nettleton
    18 tokens: two foot deep standing water develop street winthrop university campus
     9 tokens: nws columbia relayed report tree blow down tom hall
    10 tokens: medium report two tree blow down i40 old fort area
     0 tokens:
```

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 is lowest 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.

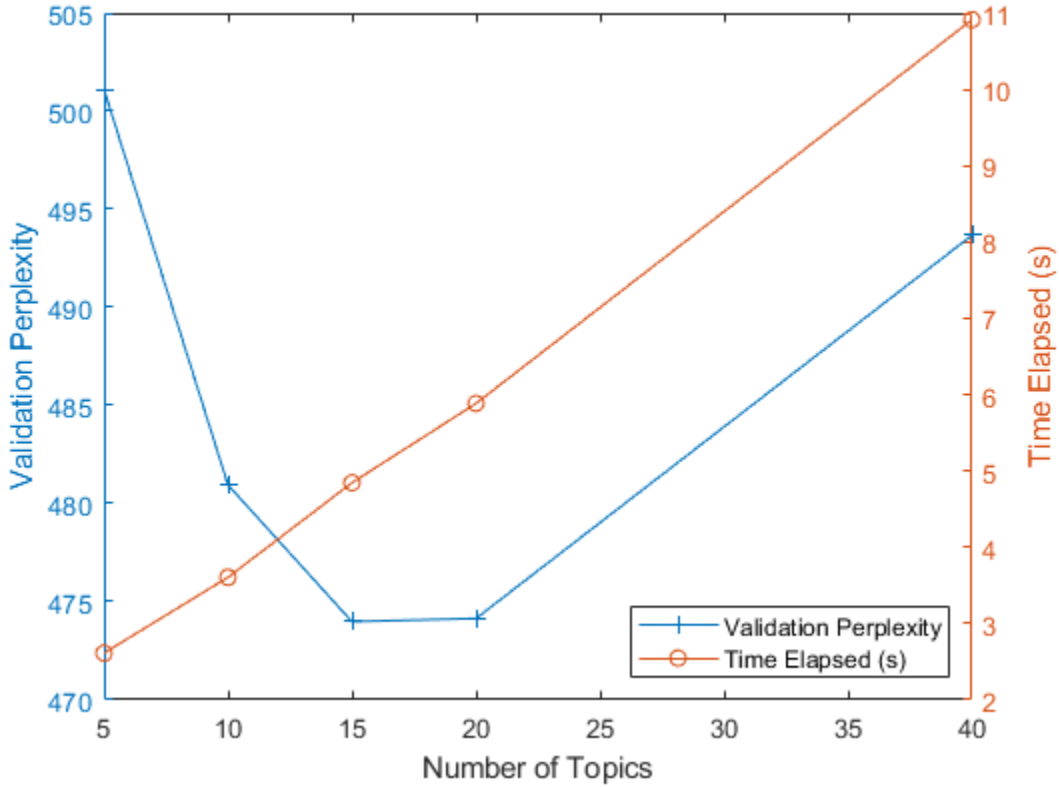
For a quicker fit, specify `'Solver'` to be `'savb'`. To suppress verbose output, set `'Verbose'` to `0`. This may take a few minutes to run.

```
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 `preprocessWeatherNarratives`, 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] = preprocessWeatherNarratives(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 2-18
- “Compare LDA Solvers” on page 2-30

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.

To reproduce the results of this example, set `rng` to `'default'`.

```
rng('default')
```

Extract and Preprocess Text Data

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

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

Set aside 10% 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);
```

Fit and Compare Models

For each of the LDA solvers, fit an LDA model with 60 topics. To distinguish the solvers when plotting the results on the same axes, specify different line properties for each solver.

```
numTopics = 60;
solvers = ["cgs" "avb" "cvb0" "savb"];
lineSpecs = ["+-" "*-" "x-" "o-"];
```

For the validation data, create a bag-of-words model from the validation documents.

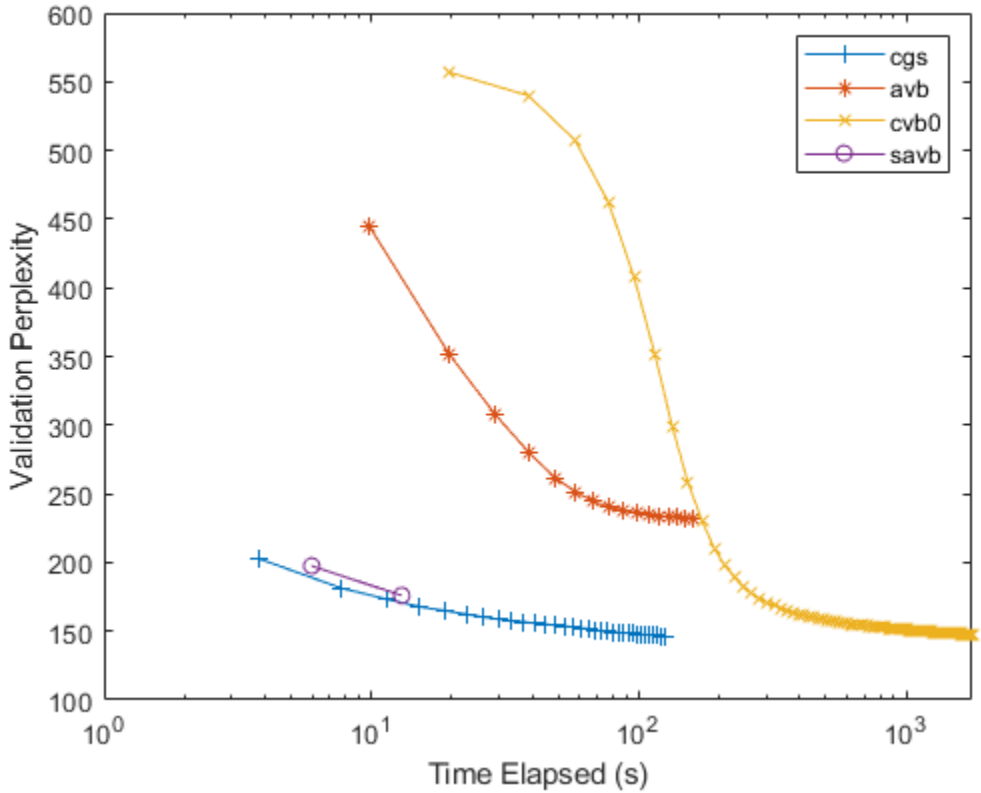
```
validationData = bagOfWords(documentsValidation);
```

For each of the LDA solvers, fit the model, set the initial topic concentration to 1, and specify not to 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. Plot the time elapsed in a logarithmic scale. This can take up to an hour to run.

The code for removing NaNs is necessary because of a quirk of the stochastic solver `'savb'`. For this solver, the function evaluates the validation perplexity after each pass of the data. The function does not evaluate the validation perplexity for each iteration (mini-batch) and reports NaNs in the `FitInfo` property. To plot the validation perplexity, remove the NaNs from the reported values.

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

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

    history = mdl.FitInfo.History;

    timeElapsed = history.TimeSinceStart;
    validationPerplexity = history.ValidationPerplexity;

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

    semilogx(timeElapsed,validationPerplexity,lineSpec)
    hold on
end
```

```
hold off
xlabel("Time Elapsed (s)")
ylabel("Validation Perplexity")
legend(solvers)
```



For the stochastic solver `"savb"`, the function, by default, passes through the training data once. To process more passes of the data, set `'DataPassLimit'` to a larger value (the default value is 1). For the batch solvers (`"cgs"`, `"avb"`, and `"cvb0"`), to reduce the number of iterations used to fit the models, set the `'IterationLimit'` option to a lower value (the default value is 100).

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 2-18
- “Choose Number of Topics for LDA Model” on page 2-25

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 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"
```

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 array
    "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 ! □□#
```

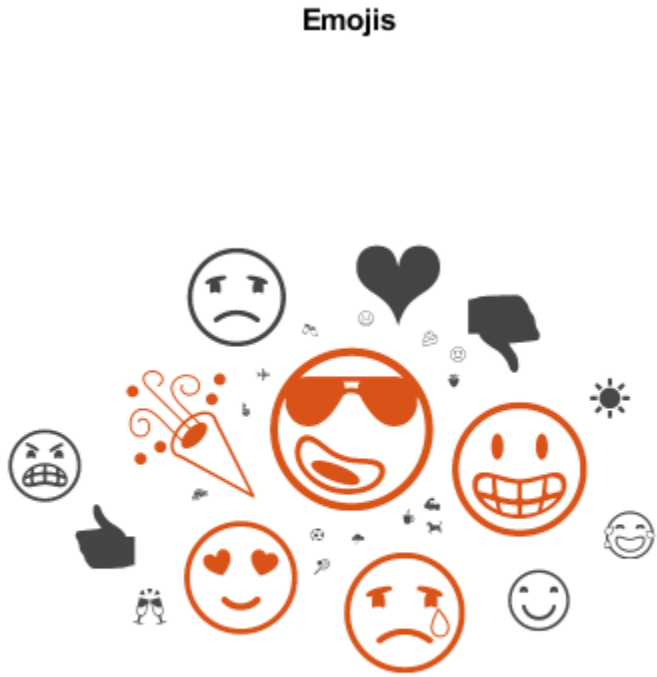
The `tokenizedDocument` function automatically detects emoji and assigns the token type `"emoji"`. View the first few token details of the documents using the `tokenDetails` function.

```
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 2-18
- “Analyze Text Data Using Multiword Phrases” on page 2-9
- “Train a Sentiment Classifier” on page 2-47
- “Classify Text Data Using Deep Learning” on page 2-57

- “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 =
  5×1 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 ! □□#
```

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

- “Train a Sentiment Classifier” on page 2-47
- “Prepare Text Data for Analysis” on page 1-12
- “Analyze Text Data Containing Emojis” on page 2-35
- “Create Simple Text Model for Classification” on page 2-2
- “Analyze Text Data Using Topic Models” on page 2-18
- “Analyze Text Data Using Multiword Phrases” on page 2-9

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.

To reproduce the results in this example, set `rng` to `'default'`.

```
rng('default')
```

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 *king is to queen as man is to woman* is described by the equation *king – man + woman = queen.*

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.

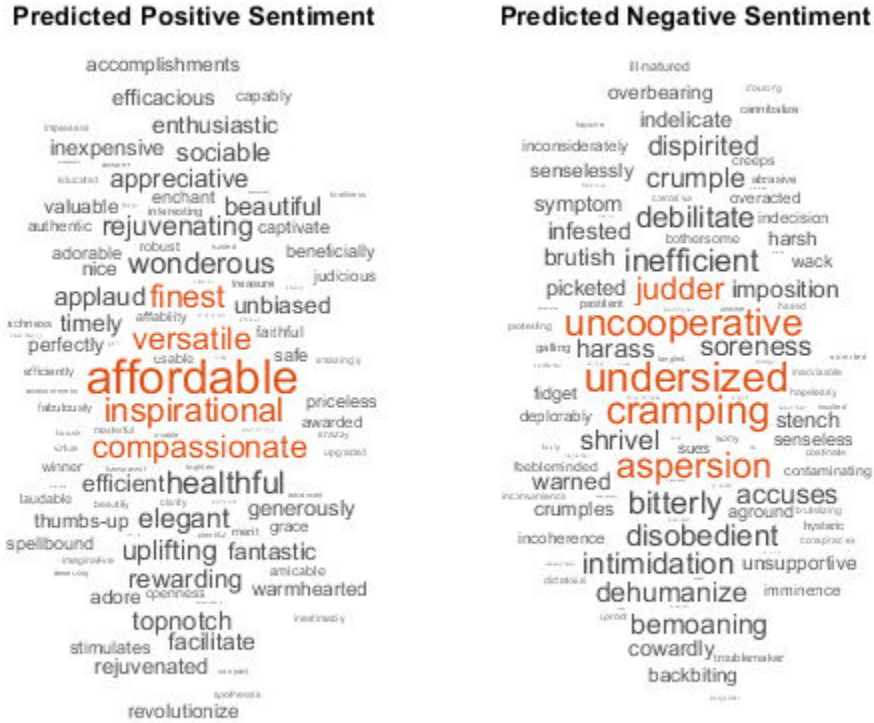
```
[YPred,scores] = predict(mdl,XTest);
```

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! □□
      -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! □□□□□B□st trip EVER! □□□□#□acation #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 s
      0.52992    "You know we all worked hard for this! We totes deserve this □□#vacat
        ⋮
```

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 2-43
- “Create Simple Text Model for Classification” on page 2-2
- “Analyze Text Data Containing Emojis” on page 2-35
- “Analyze Text Data Using Topic Models” on page 2-18
- “Analyze Text Data Using Multiword Phrases” on page 2-9
- “Classify Text Data Using Deep Learning” on page 2-57
- “Generate Text Using Deep Learning” (Deep Learning Toolbox)

Classify Text Data Using Deep Learning

This example shows how to classify text descriptions of weather reports 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 "*king is to queen as man is to woman*" is described by the equation *king – man + woman = queen*.

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 weather reports data. This data contains labeled textual descriptions of weather events. To import the text data as strings, specify the text type to be `'string'`.

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

```
ans=8×16 table
            Time             event_id          state              event_type          da
    _____________________    __________    _______________    ____________________    __

    22-Jul-2016 16:10:00    6.4433e+05    "MISSISSIPPI"       "Thunderstorm Wind"
```

```
    15-Jul-2016 17:15:00    6.5182e+05    "SOUTH CAROLINA"    "Heavy Rain"
    15-Jul-2016 17:25:00    6.5183e+05    "SOUTH CAROLINA"    "Thunderstorm Wind"
    16-Jul-2016 12:46:00    6.5183e+05    "NORTH CAROLINA"    "Thunderstorm Wind"
    15-Jul-2016 14:28:00    6.4332e+05    "MISSOURI"          "Hail"
    15-Jul-2016 16:31:00    6.4332e+05    "ARKANSAS"          "Thunderstorm Wind"
    15-Jul-2016 16:03:00    6.4343e+05    "TENNESSEE"         "Thunderstorm Wind"
    15-Jul-2016 17:27:00    6.4344e+05    "TENNESSEE"         "Hail"
```

Remove the rows of the table with empty reports.

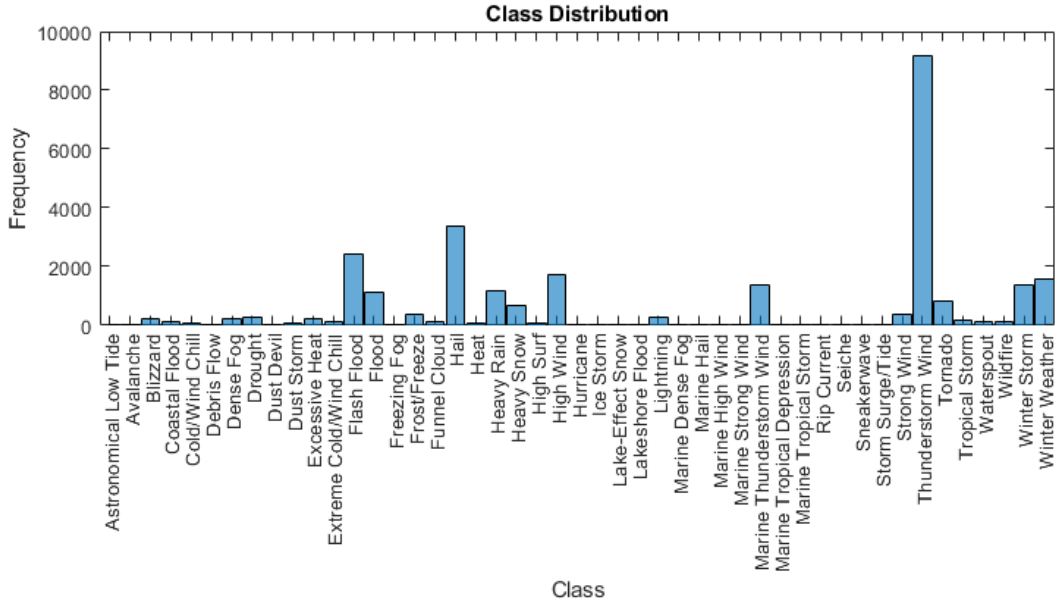
```
idxEmpty = strlength(data.event_narrative) == 0;
data(idxEmpty,:) = [];
```

The goal of this example is to classify events by the label in the `event_type` column. To divide the data into classes, convert these labels to categorical.

```
data.event_type = categorical(data.event_type);
```

View the distribution of the classes in the data using a histogram. To make the labels easier to read, increase the width of the figure.

```
f = figure;
f.Position(3) = 1.5*f.Position(3);

h = histogram(data.event_type);
xlabel("Class")
ylabel("Frequency")
title("Class Distribution")
```



The classes of the data are imbalanced, with many classes containing few observations. When the classes are imbalanced in this way, the network might converge to a less accurate model. To prevent this problem, remove any classes which appear fewer than ten times.

Get the frequency counts of the classes and the class names from the histogram.

```
classCounts = h.BinCounts;
classNames = h.Categories;
```

Find the classes containing fewer than ten observations.

```
idxLowCounts = classCounts < 10;
infrequentClasses = classNames(idxLowCounts)
```

```
infrequentClasses = 1×8 cell array
    {'Freezing Fog'}    {'Hurricane'}    {'Lakeshore Flood'}    {'Marine Dense Fog'}
```

Remove these infrequent classes from the data. Use `removecats` to remove the unused categories from the categorical data.

```
idxInfrequent = ismember(data.event_type,infrequentClasses);
data(idxInfrequent,:) = [];
data.event_type = removecats(data.event_type);
```

Now the data is sorted into classes of reasonable size. The next step is to partition it into sets for training, validation, and testing. Partition the data into a training partition and a held-out partition for validation and testing. Specify the holdout percentage to be 30%.

```
cvp = cvpartition(data.event_type,'Holdout',0.3);
dataTrain = data(training(cvp),:);
dataHeldOut = data(test(cvp),:);
```

Partition the held-out set again to get a validation set. Specify the holdout percentage to be 50%. This results in a partitioning of 70% training observations, 15% validation observations, and 15% test observations.

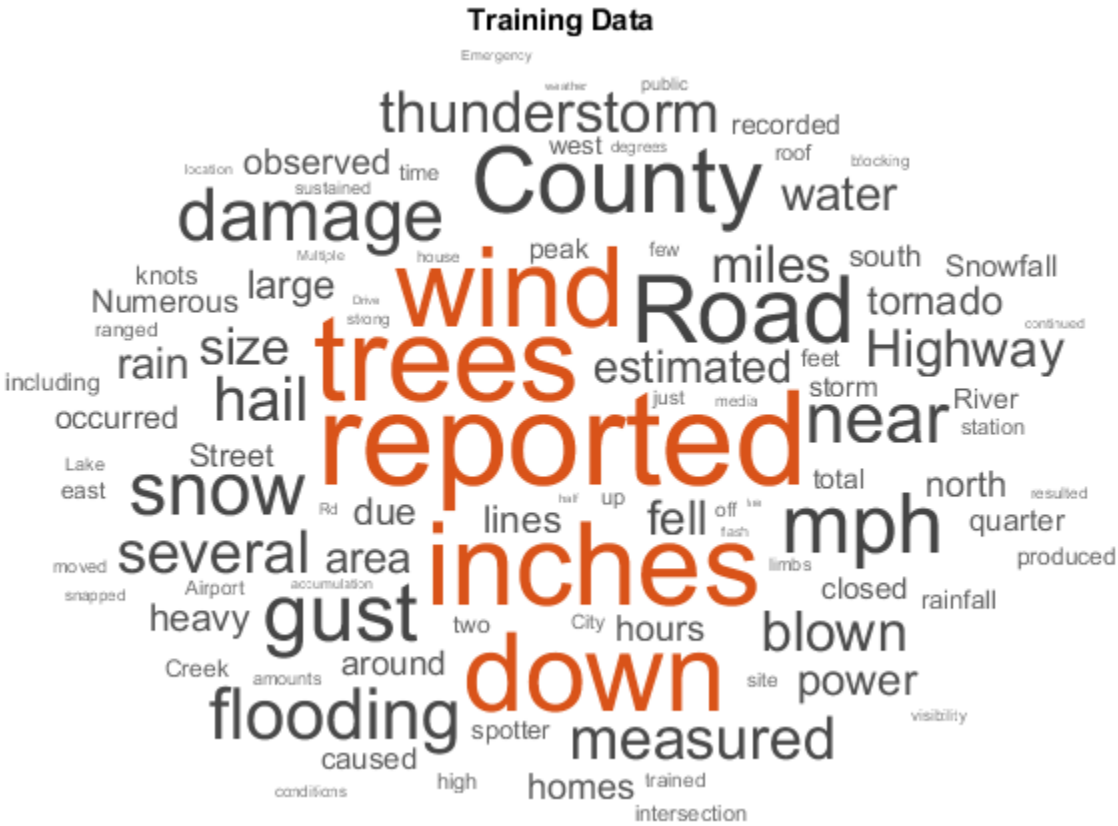
```
cvp = cvpartition(dataHeldOut.event_type,'HoldOut',0.5);
dataValidation = dataHeldOut(training(cvp),:);
dataTest = dataHeldOut(test(cvp),:);
```

Extract the text data and labels from the partitioned tables.

```
textDataTrain = dataTrain.event_narrative;
textDataValidation = dataValidation.event_narrative;
textDataTest = dataTest.event_narrative;
YTrain = dataTrain.event_type;
YValidation = dataValidation.event_type;
YTest = dataTest.event_type;
```

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:

     7 tokens: large tree down between plantersville and nettleton
    37 tokens: one to two feet of deep standing water developed on a street on the wint
    13 tokens: nws columbia relayed a report of trees blown down along tom hall st
    13 tokens: media reported two trees blown down along i40 in the old fort area
    14 tokens: a few tree limbs greater than 6 inches down on hwy 18 in roseland
```

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.

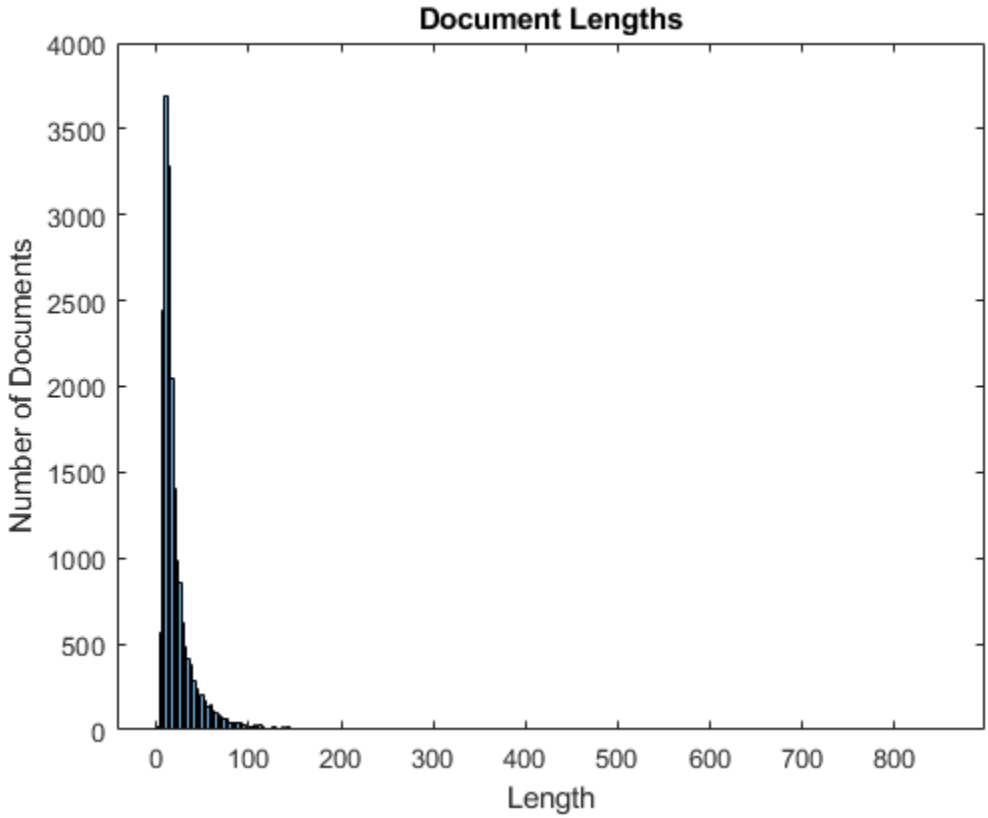
To create a word encoding, use the `wordEncoding` function.

```
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 75 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 75, set the `'Length'` option to 75.

```
XTrain = doc2sequence(enc,documentsTrain,'Length',75);
XTrain(1:5)
```

```
ans=5×1 cell
    {1×75 double}
    {1×75 double}
    {1×75 double}
    {1×75 double}
```

```
    {1×75 double}
```

Convert the validation documents to sequences using the same options.

```
XValidation = doc2sequence(enc,documentsValidation,'Length',75);
```

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 set the number of hidden units to 180. 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 = 100;
numWords = enc.NumWords;
numHiddenUnits = 180;
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 100 dimensions and 169
     3   ''   LSTM                   LSTM with 180 hidden units
     4   ''   Fully Connected         39 fully connected layer
     5   ''   Softmax                softmax
     6   ''   Classification Output   crossentropyex
```

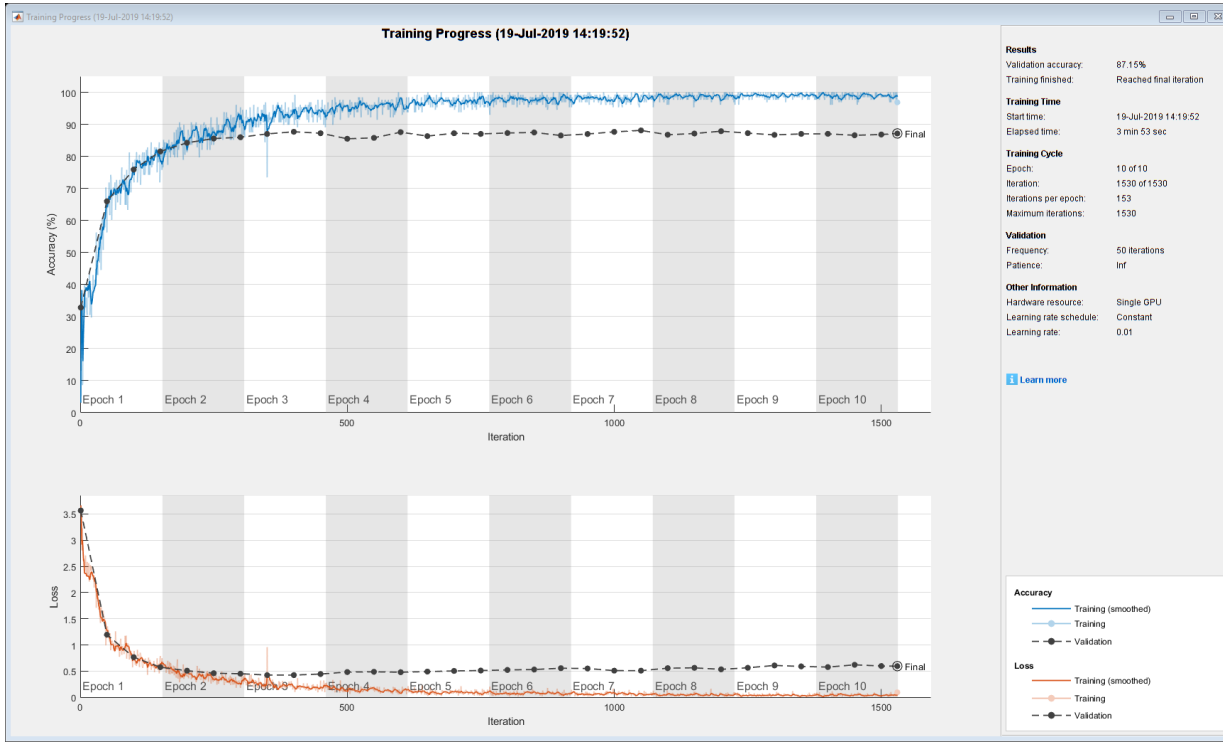
Specify the training options. Set the solver to `'adam'`, train for 10 epochs, and set the gradient threshold to 1. Set the initial learn rate to 0.01. To monitor the training progress, set the `'Plots'` option to `'training-progress'`. Specify the validation data using the `'ValidationData'` option. 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.

```
options = trainingOptions('adam', ...
    'MaxEpochs',10, ...
    'GradientThreshold',1, ...
    'InitialLearnRate',0.01, ...
    'ValidationData',{XValidation,YValidation}, ...
    'Plots','training-progress', ...
    'Verbose',false);
```

Train the LSTM network using the `trainNetwork` function.

```
net = trainNetwork(XTrain,YTrain,layers,options);
```



Test LSTM Network

To test the LSTM network, first prepare the test data in the same way as the training data. Then make predictions on the preprocessed test data using the trained LSTM network `net`.

Preprocess the test data using the same steps as the training documents.

```
textDataTest = lower(textDataTest);
documentsTest = tokenizedDocument(textDataTest);
documentsTest = erasePunctuation(documentsTest);
```

Convert the test documents to sequences using `doc2sequence` with the same options as when creating the training sequences.

```
XTest = doc2sequence(enc,documentsTest,'Length',75);
XTest(1:5)
```

```
ans=5×1 cell
    {1×75 double}
    {1×75 double}
    {1×75 double}
    {1×75 double}
    {1×75 double}
```

Classify the test documents using the trained LSTM network.

```
YPred = classify(net,XTest);
```

Calculate the classification accuracy. The accuracy is the proportion of labels that the network predicts correctly.

```
accuracy = sum(YPred == YTest)/numel(YPred)
```

```
accuracy = 0.8684
```

Predict Using New Data

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

```
reportsNew = [ ...
    "Lots of water damage to computer equipment inside the office."
    "A large tree is downed and blocking traffic outside Apple Hill."
    "Damage to many car windshields in parking lot."];
```

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',75);
```

Classify the new sequences using the trained LSTM network.

```
[labelsNew,score] = classify(net,XNew);
```

Show the weather reports with their predicted labels.

```
[reportsNew string(labelsNew)]
```

```
ans = 3×2 string array
    "Lots of water damage to computer equipment inside the office."      "Flash Flood"
    "A large tree is downed and blocking traffic outside Apple Hill."    "Thunderstorm
    "Damage to many car windshields in parking lot."                     "Hail"
```

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 2-69
- “Classify Out-of-Memory Text Data Using Deep Learning” on page 2-106
- “Generate Text Using Deep Learning” (Deep Learning Toolbox)
- “Word-By-Word Text Generation Using Deep Learning” on page 2-120
- “Create Simple Text Model for Classification” on page 2-2
- “Analyze Text Data Using Topic Models” on page 2-18
- “Analyze Text Data Using Multiword Phrases” on page 2-9
- “Train a Sentiment Classifier” on page 2-47
- “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 `weatherReportsTrain.csv`. Read the data from the `"event_narrative"` and `"event_type"` columns only.

```
filenameTrain = "weatherReportsTrain.csv";
textName = "event_narrative";
labelName = "event_type";
ttdsTrain = tabularTextDatastore(filenameTrain,'SelectedVariableNames',[textName labelN
```

Preview the datastore.

```
ttdsTrain.ReadSize = 8;
preview(ttdsTrain)
```

```
ans=8×2 table
```

{'Large tree down between Plantersville and Nettleton.'	
{'One to two feet of deep standing water developed on a street on the Winthrop Univ	
{'NWS Columbia relayed a report of trees blown down along Tom Hall St.'	
{'Media reported two trees blown down along I-40 in the Old Fort area.'	
{'A few tree limbs greater than 6 inches down on HWY 18 in Roseland.'	
{'Awning blown off a building on Lamar Avenue. Multiple trees down near the interse	
{'Tin roof ripped off house on Old Memphis Road near Billings Drive. Several large	
{'Powerlines down at Walnut Grove and Cherry Lane roads.'	

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 100.

```
sequenceLength = 100;
tdsTrain = transform(ttdsTrain, @(data) transformTextData(data,sequenceLength,emb,class
```

```
tdsTrain =
  TransformedDatastore with properties:

    UnderlyingDatastore: [1×1 matlab.io.datastore.TabularTextDatastore]
             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
    ____________________    _______________

    {1×100×300 single}    Thunderstorm Wind
    {1×100×300 single}    Heavy Rain
    {1×100×300 single}    Thunderstorm Wind
    {1×100×300 single}    Thunderstorm Wind
    {1×100×300 single}    Thunderstorm Wind
    {1×100×300 single}    Thunderstorm Wind
    {1×100×300 single}    Thunderstorm Wind
    {1×100×300 single}    Thunderstorm Wind
```

Create a transformed datastore containing the validation data in `weatherReportsValidation.csv` using the same steps.

```
filenameValidation = "weatherReportsValidation.csv";
ttdsValidation = tabularTextDatastore(filenameValidation,'SelectedVariableNames',[textN

tdsValidation = transform(ttdsValidation, @(data) transformTextData(data,sequenceLengt
```

```
tdsValidation =
  TransformedDatastore with properties:

    UnderlyingDatastore: [1×1 matlab.io.datastore.TabularTextDatastore]
             Transforms: {@(data)transformTextData(data,sequenceLength,emb,classNames)}
            IncludeInfo: 0
```

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);
```

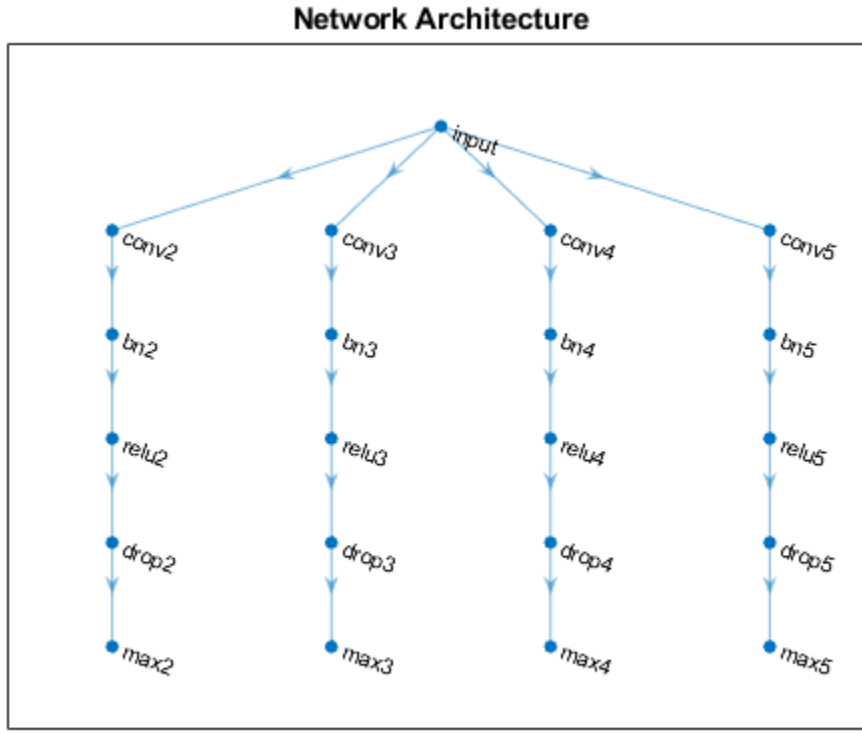
For each of the n-gram lengths, create a block of convolution, batch normalization, ReLU, dropout, and max pooling layers. Connect each block to the input layer.

```
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.

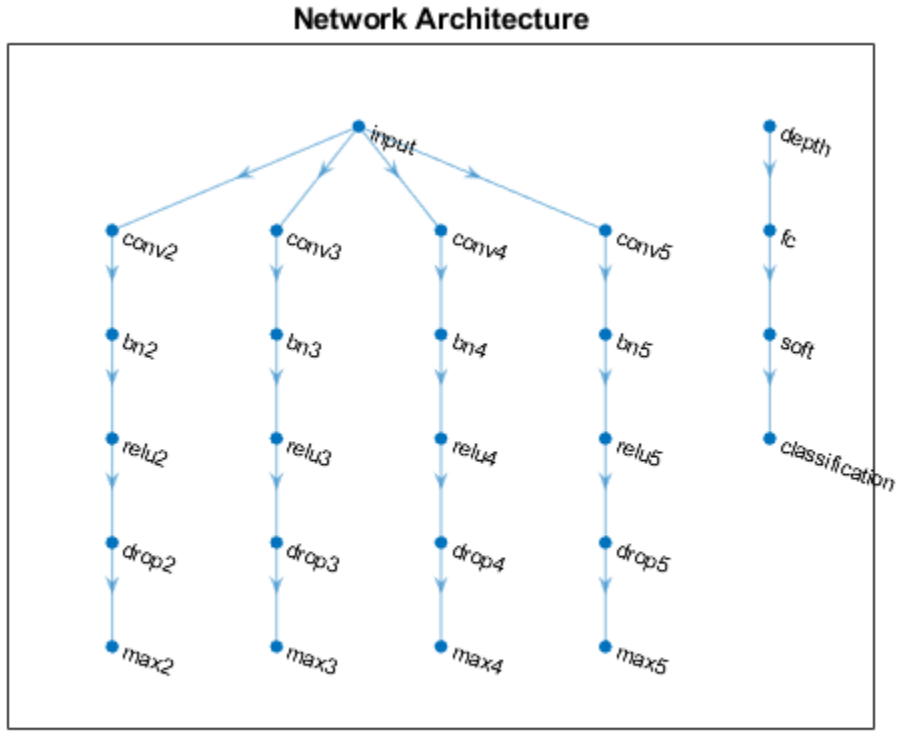
```
figure
plot(lgraph)
title("Network Architecture")
```



Add the depth concatenation layer, the fully connected layer, the softmax layer, and the classification layer.

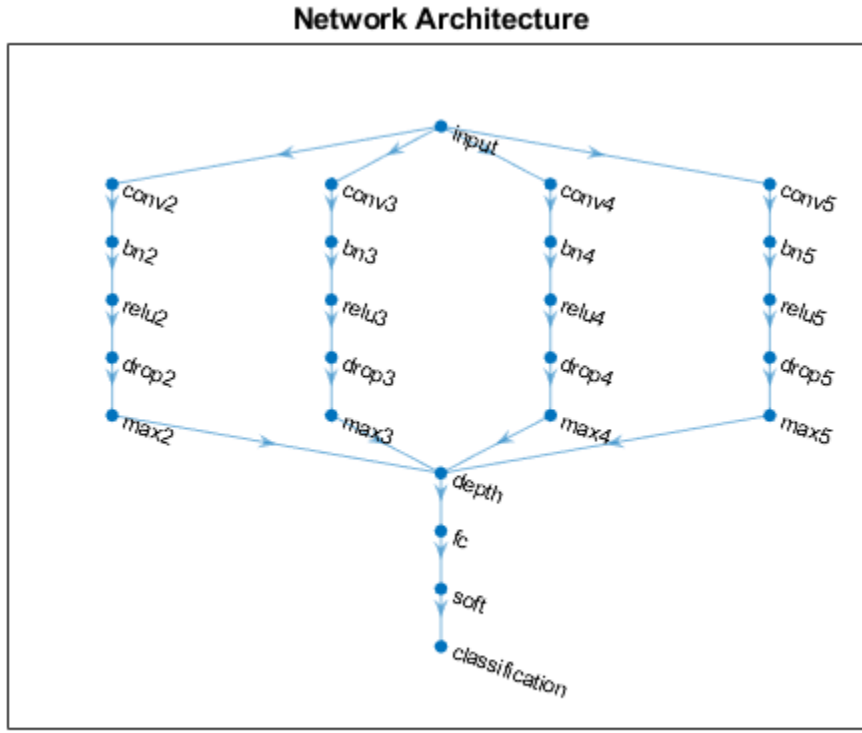
```
layers = [
    depthConcatenationLayer(numBlocks,'Name','depth')
```

```
    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:

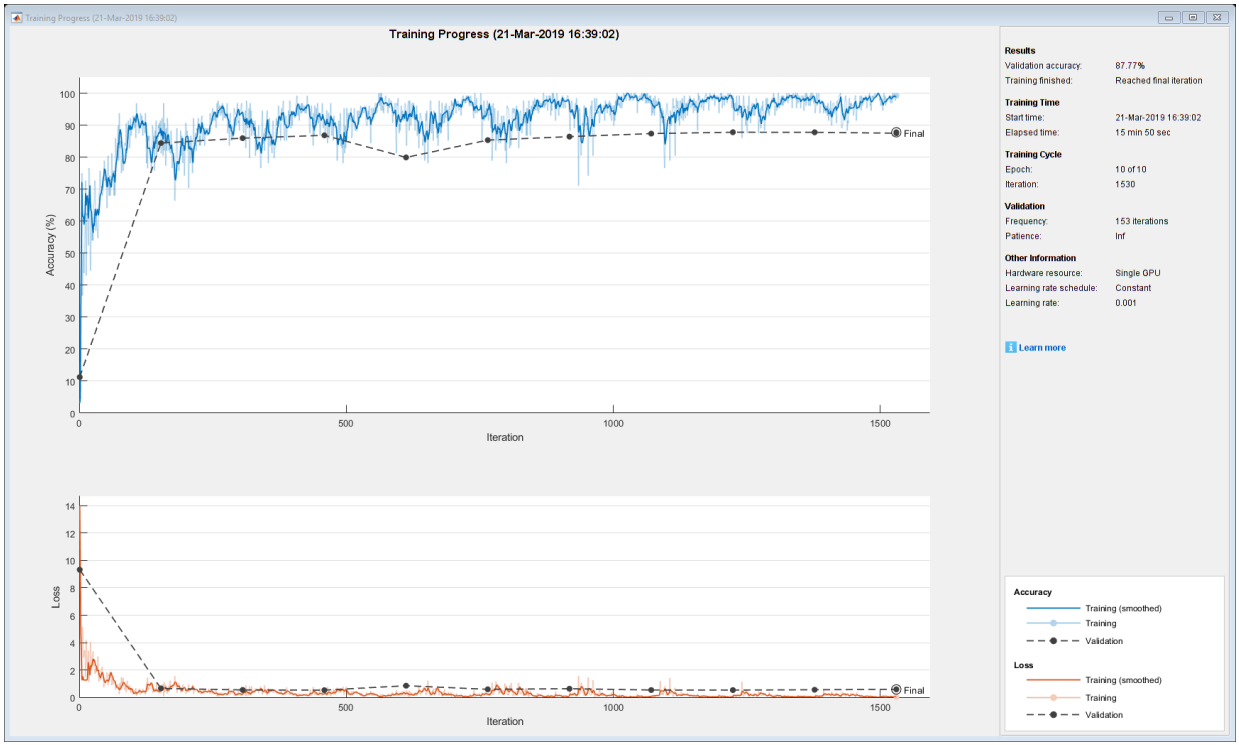
- Train for 10 epochs with a mini-batch size of 128.

- Do not shuffle the data because the datastore is not shuffleable.
- Validate the network at each epoch by setting the validation frequency to the number of iterations per epoch.
- Display the training progress plot and suppress the verbose output.

```
miniBatchSize = 128;
numIterationsPerEpoch = floor(numObservations/miniBatchSize);

options = trainingOptions('adam', ...
    'MaxEpochs',10, ...
    'MiniBatchSize',miniBatchSize, ...
    'Shuffle','never', ...
    'ValidationData',tdsValidation, ...
    'ValidationFrequency',numIterationsPerEpoch, ...
    'Plots','training-progress', ...
    'Verbose',false);
```

Train the network using the `trainNetwork` function.

```
net = trainNetwork(tdsTrain,lgraph,options);
```



Test Network

Create a transformed datastore containing the held-out test data in `weatherReportsTest.csv`.

```
filenameTest = "weatherReportsTest.csv";
ttdsTest = tabularTextDatastore(filenameTest,'SelectedVariableNames',[textName labelNar

tdsTest = transform(ttdsTest, @(data) transformTextData(data,sequenceLength,emb,classNa

tdsTest =
  TransformedDatastore with properties:

    UnderlyingDatastore: [1×1 matlab.io.datastore.TabularTextDatastore]
             Transforms: {@(data)transformTextData(data,sequenceLength,emb,classNames)}
            IncludeInfo: 0
```

Read the labels from the `tabularTextDatastore`.

```
labelsTest = readLabels(ttdsTest,labelName);
YTest = categorical(labelsTest,classNames);
```

Make predictions on the test data using the trained network.

```
YPred = classify(net,tdsTest);
```

Calculate the classification accuracy on the test data.

```
accuracy = mean(YPred == YTest)
```

```
accuracy = 0.8795
```

Functions

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
```

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};
textData = lower(textData);
documents = tokenizedDocument(textData);

% 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);

% Read labels.
labels = data{:,2};
responses = categorical(labels,classNames);

% Convert data to table.
dataTransformed = table(predictors,responses);

end
```

See Also

`batchNormalizationLayer` | `convolution2dLayer` | `doc2sequence` | `fastTextWordEmbedding` | `layerGraph` | `tokenizedDocument` | `trainNetwork` | `trainingOptions` | `wordEmbedding` | `wordcloud`

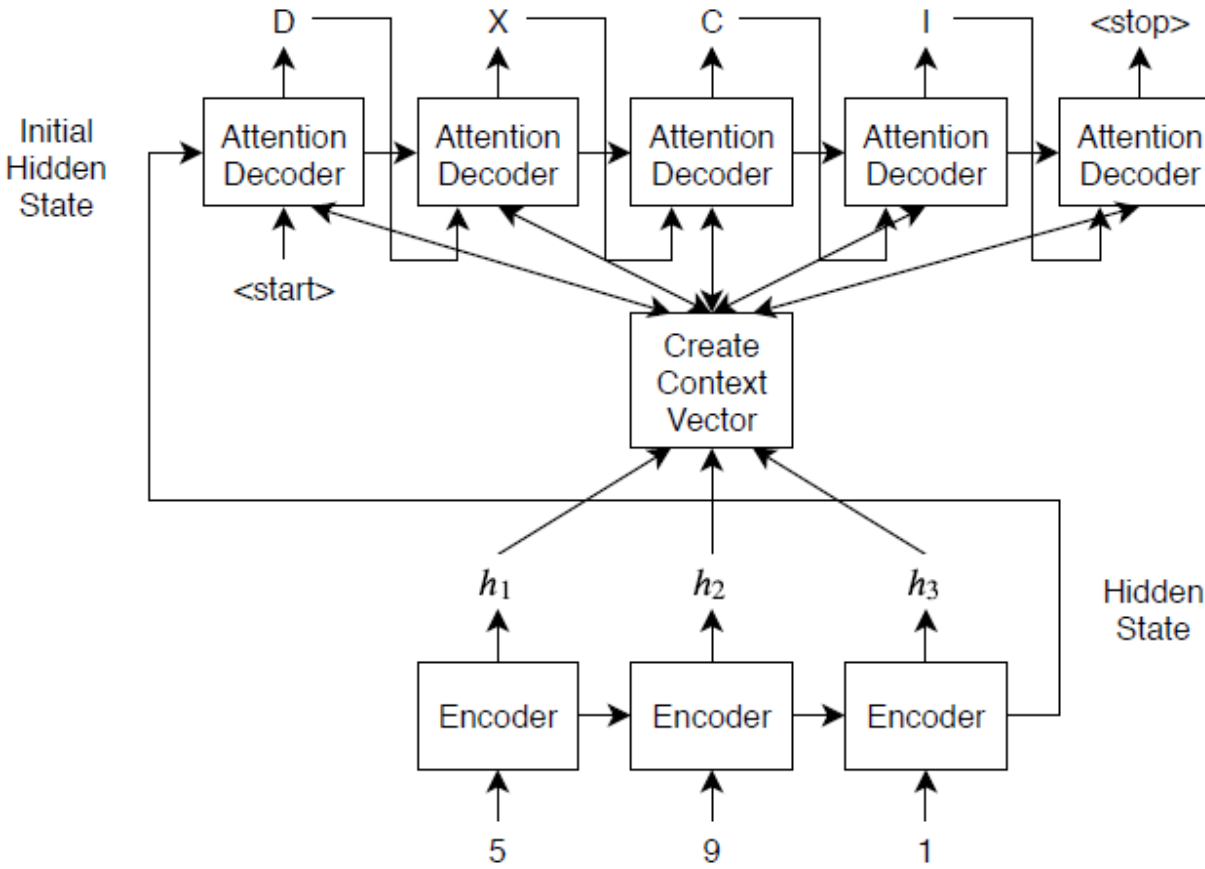
Related Examples

- “Classify Text Data Using Deep Learning” on page 2-57
- “Classify Out-of-Memory Text Data Using Deep Learning” on page 2-106
- “Create Simple Text Model for Classification” on page 2-2
- “Analyze Text Data Using Topic Models” on page 2-18
- “Analyze Text Data Using Multiword Phrases” on page 2-9
- “Train a Sentiment Classifier” on page 2-47
- “Sequence Classification Using Deep Learning” (Deep Learning Toolbox)
- “Deep Learning in MATLAB” (Deep Learning Toolbox)

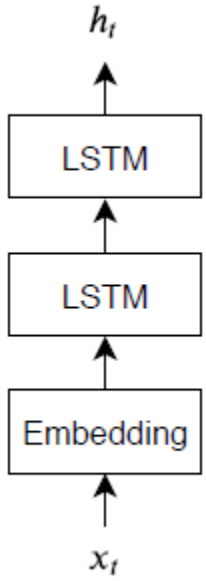
Sequence-to-Sequence Translation Using Attention

This example shows how to convert decimal strings to Roman numerals using a recurrent sequence-to-sequence encoder-decoder model with attention.

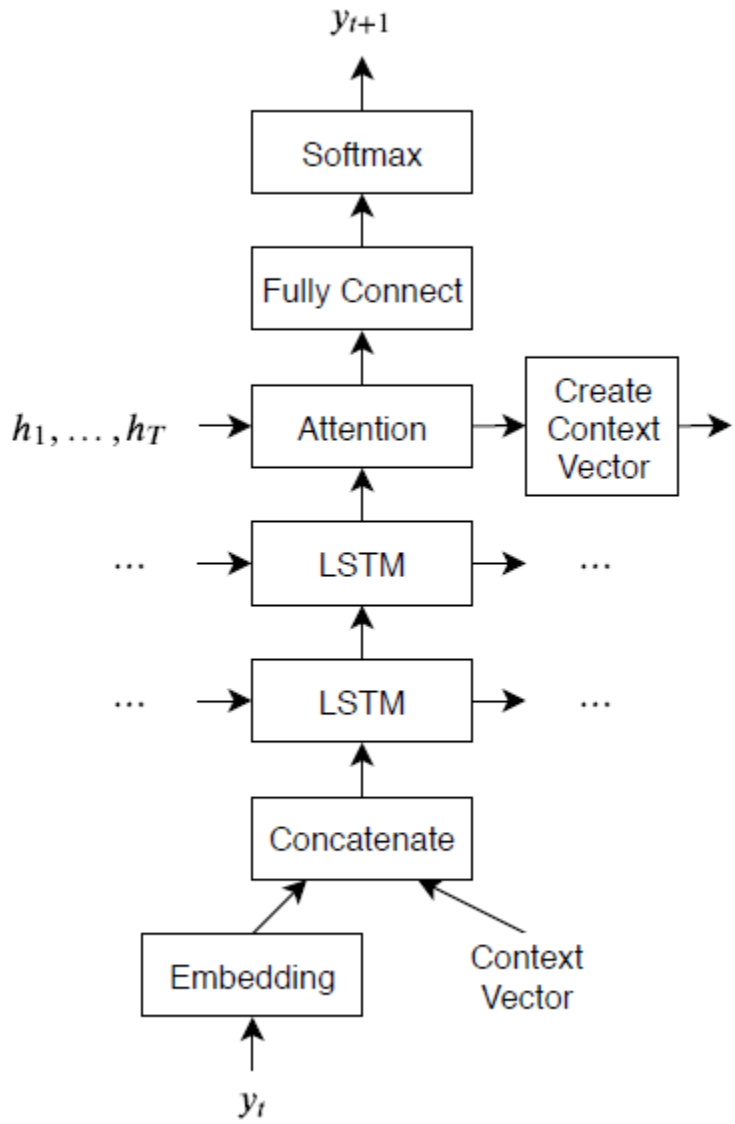
Recurrent encoder-decoder models have proven successful at tasks like abstractive text summarization and neural machine translation. The models consistent of an *encoder* which typically processes input data with a recurrent layer such as LSTM, and a *decoder* which maps the encoded input into the desired 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.



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
    ______    ___________

    "437"     "CDXXXVII"
    "431"     "CDXXXI"
    "102"     "CII"
    "862"     "DCCCLXII"
    "738"     "DCCXXXVIII"
    "527"     "DXXVII"
    "401"     "CDI"
    "184"     "CLXXXIV"
```

Preprocess Data

Preprocess the training data using the `preprocessSourceTargetPairs` function, listed at the end of the example. The `preprocessSourceTargetPairs` function converts the input text data to numeric sequences. The elements of the sequences are positive integers that index into a corresponding `wordEncoding` object. The `wordEncoding` maps tokens to a numeric index and vice-versa using a vocabulary. To highlight the beginning and the ends of sequences, the encoding also encapsulates the special tokens `"<start>"` and `"<stop>"`.

```
startToken = "<start>";
stopToken = "<stop>";
[sequencesSource, sequencesTarget, encSource, encTarget] = preprocessSourceTargetPairs
```

Representing Text as Numeric Sequences

For example, the decimal string `"441"` is encoded as follows:

```
strSource = "441";
```

Insert spaces between the characters.

```
strSource = strip(replace(strSource,"","  "));
```

Add the special start and stop tokens.

```
strSource = startToken + strSource + stopToken
```

```
strSource = 1×1 string
"<start>4 4 1<stop>"
```

Tokenize the text using the `tokenizedDocument` function and set the `'CustomTokens'` option to the special tokens.

```
documentSource = tokenizedDocument(strSource,'CustomTokens',[startToken stopToken])
```

```
documentSource =
  tokenizedDocument:

   5 tokens: <start> 4 4 1 <stop>
```

Convert the document to a sequence of token indices using the `word2ind` function with the corresponding `wordEncoding` object.

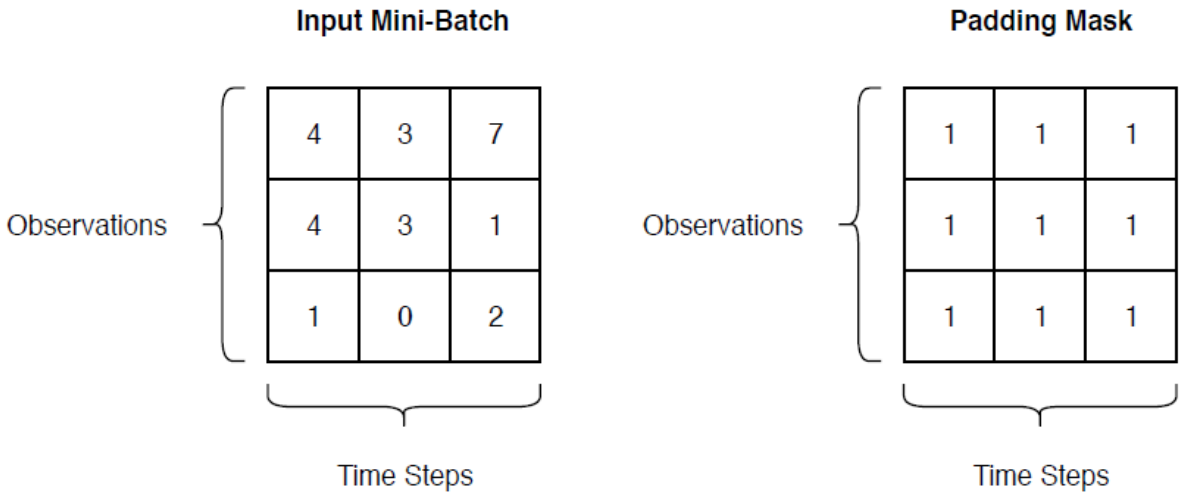
```
tokens = string(documentSource);
sequenceSource = word2ind(encSource,tokens)
```

```
sequenceSource = 1×5

     1     2     2     6     5
```

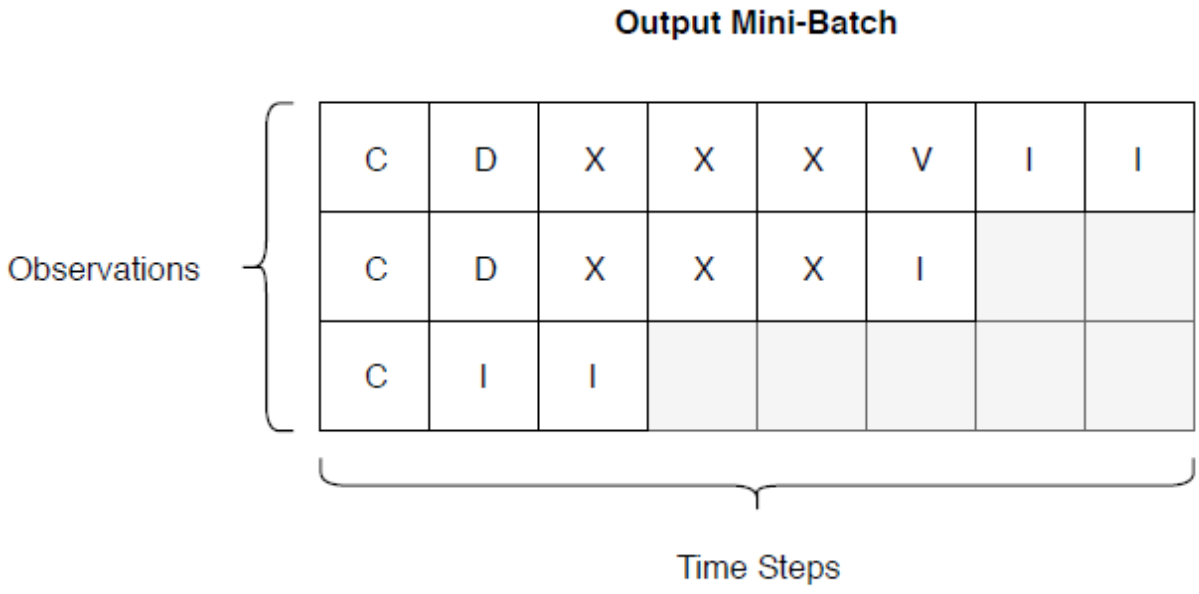
Padding and Masking

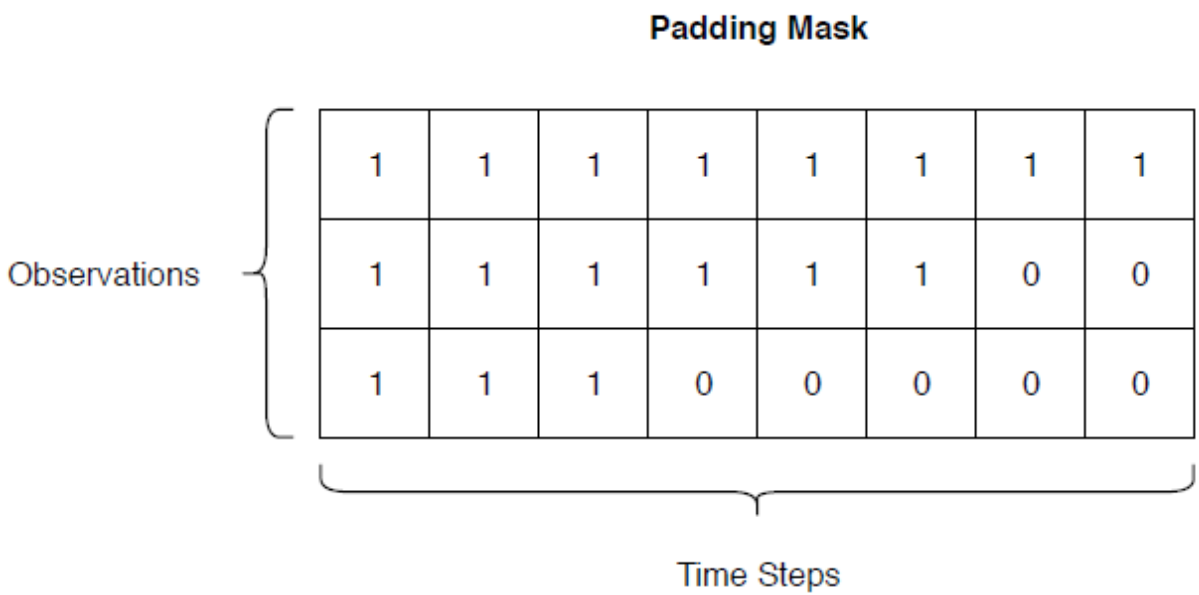
Sequence data such as text naturally have different sequence lengths. To train a model using variable length sequences, pad the mini-batches of input data to have the same length. To ensure that the padding values do not impact the loss calculations, create a mask which records which sequence elements are real, and which are just padding.

For example, consider a mini-batch containing the decimal strings "437", "431", and "102" with the corresponding Roman numeral strings "CDXXXVII", "CDXXXI", and "CII". For character-by-character sequences, the input sequences have the same length and do not need to be padded. The corresponding mask is an array of ones.



The output sequences have different lengths, so they require padding. The corresponding padding mask contains zeros where the corresponding time steps are padding values.


Output Mini-Batch
Observations
C D X X X V I I
C D X X X I
C I I
Time Steps


Padding Mask
Observations
1 1 1 1 1 1 1 1
1 1 1 1 1 1 0 0
1 1 1 0 0 0 0 0
Time Steps

Initialize Model Parameters

Initialize the model parameters. for both the encoder and decoder, specify an embedding dimension of 256, two LSTM layers with 200 hidden units, and dropout layers with random dropout with probability 0.05.

```
embeddingDimension = 256;
numHiddenUnits = 200;
dropout = 0.05;
```

Initialize the encoder model parameters:

- Specify an embedding dimension of 256 and the vocabulary size of the source vocabulary plus 1, where the extra value corresponds to the padding token.
- Specify two LSTM operations with 200 hidden units.
- Initialize the embedding weights by sampling from a random normal distribution.
- Initialize the LSTM weights and biases by sampling from a uniform distribution using the `uniformNoise` function, listed at the end of the example.

```
inputSize = encSource.NumWords + 1;
parametersEncoder.emb.Weights = dlarray(randn([embeddingDimension inputSize]));

parametersEncoder.lstm1.InputWeights = dlarray(uniformNoise([4*numHiddenUnits embedding
parametersEncoder.lstm1.RecurrentWeights = dlarray(uniformNoise([4*numHiddenUnits numH
parametersEncoder.lstm1.Bias = dlarray(uniformNoise([4*numHiddenUnits 1],1/numHiddenUn

parametersEncoder.lstm2.InputWeights = dlarray(uniformNoise([4*numHiddenUnits numHidde
parametersEncoder.lstm2.RecurrentWeights = dlarray(uniformNoise([4*numHiddenUnits numH
parametersEncoder.lstm2.Bias = dlarray(uniformNoise([4*numHiddenUnits 1],1/numHiddenUn
```

Initialize the decoder model parameters.

- Specify an embedding dimension of 256 and the vocabulary size of the target vocabulary plus 1, where the extra value corresponds to the padding token.
- Initialize the attention mechanism weights using the `uniformNoise` function.
- Initialize the embedding weights by sampling from a random normal distribution.
- Initialize the LSTM weights and biases by sampling from a uniform distribution using the `uniformNoise` function.

```
outputSize = encTarget.NumWords + 1;
parametersDecoder.emb.Weights = dlarray(randn([embeddingDimension outputSize]));
```

```
parametersDecoder.attn.Weights = dlarray(uniformNoise([numHiddenUnits numHiddenUnits],1

parametersDecoder.lstm1.InputWeights = dlarray(uniformNoise([4*numHiddenUnits embedding
parametersDecoder.lstm1.RecurrentWeights = dlarray(uniformNoise([4*numHiddenUnits numHi
parametersDecoder.lstm1.Bias = dlarray( uniformNoise([4*numHiddenUnits 1],1/numHiddenUn

parametersDecoder.lstm2.InputWeights = dlarray(uniformNoise([4*numHiddenUnits numHidden
parametersDecoder.lstm2.RecurrentWeights = dlarray(uniformNoise([4*numHiddenUnits numHi
parametersDecoder.lstm2.Bias = dlarray(uniformNoise([4*numHiddenUnits 1], 1/numHiddenUn

parametersDecoder.fc.Weights = dlarray(uniformNoise([outputSize 2*numHiddenUnits],1/(2*
parametersDecoder.fc.Bias = dlarray(uniformNoise([outputSize 1], 1/(2*numHiddenUnits))
```

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 40 epochs. Specify a learning rate of 0.002 and clip the gradients with a threshold of 5.

```
miniBatchSize = 32;
numEpochs = 40;
```

```
learnRate = 0.002;
gradientThreshold = 5;
```

Initialize the options from Adam.

```
gradientDecayFactor = 0.9;
squaredGradientDecayFactor = 0.999;
```

Specify to plot the training progress. To disable the training progress plot, set the `plots` value to `"none"`.

```
plots = "training-progress";
```

Train Model

Train the model using a custom training loop.

For the first epoch, 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. For subsequent epochs, shuffle the data.

For each mini-batch:

- Convert the data to `dlarray`.
- Compute loss and gradients.
- Clip the gradients.
- Update the encoder and decoder model parameters using the `adamupdate` function.
- Update the training progress plot.

Sort the sequences for the first epoch.

```
sequenceLengthsEncoder = cellfun(@(sequence) size(sequence,2), sequencesSource);
[~,idx] = sort(sequenceLengthsEncoder);
sequencesSource = sequencesSource(idx);
sequencesTarget = sequencesTarget(idx);
```

Initialize the training progress plot.

```
if plots == "training-progress"
    figure
    lineLossTrain = animatedline;
    xlabel("Iteration")
```

```
    ylabel("Loss")
end
```

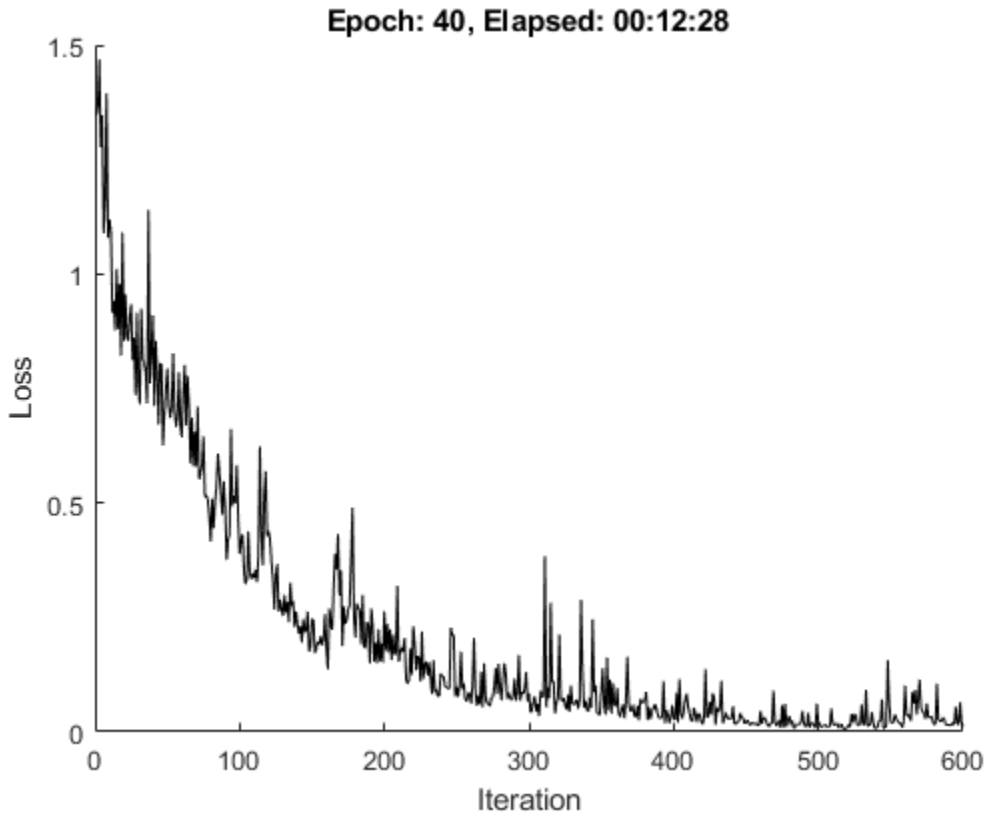
Initialize the values for the `adamupdate` function.

```
trailingAvgEncoder = [];
trailingAvgSqEncoder = [];

trailingAvgDecoder = [];
trailingAvgSqDecoder = [];
```

Train the model.

```
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
        idx = (i-1)*miniBatchSize+1:i*miniBatchSize;
        [XSource, XTarget, maskSource, maskTarget] = createBatch(sequencesSource(idx),
            sequencesTarget(idx), inputSize, outputSize);

        % Convert mini-batch of data to dlarray.
        dlXSource = dlarray(XSource);
        dlXTarget = dlarray(XTarget);

        % Compute loss and gradients.
        [gradientsEncoder, gradientsDecoder, loss] = dlfeval(@modelGradients, parameter
            parametersDecoder, dlXSource, dlXTarget, maskSource, maskTarget, dropout);

        % Gradient clipping.
        gradientsEncoder = dlupdate(@(w) clipGradient(w,gradientThreshold), gradientsEn
        gradientsDecoder = dlupdate(@(w) clipGradient(w,gradientThreshold), gradientsDe

        % Update encoder using adamupdate.
        [parametersEncoder, trailingAvgEncoder, trailingAvgSqEncoder] = adamupdate(para
```

```
            gradientsEncoder, trailingAvgEncoder, trailingAvgSqEncoder, iteration, lear
            gradientDecayFactor, squaredGradientDecayFactor);

        % Update decoder using adamupdate.
        [parametersDecoder, trailingAvgDecoder, trailingAvgSqDecoder] = adamupdate(para
            gradientsDecoder, trailingAvgDecoder, trailingAvgSqDecoder, iteration, lear
            gradientDecayFactor, squaredGradientDecayFactor);

        % Display the training progress.
        if plots == "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

    % Shuffle data.
    idx = randperm(numObservations);
    sequencesSource = sequencesSource(idx);
    sequencesTarget = sequencesTarget(idx);
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.

Prepare Data for Translation

Select a mini-batch of test observations.

```
numObservationsTest = 16;
idx = randperm(size(dataTest,1),numObservationsTest);
dataTest(idx,:)
```

```
ans=16×2 table
    Source       Target
    ______    ____________

    "412"     "CDXII"
    "274"     "CCLXXIV"
    "231"     "CCXXXI"
    "558"     "DLVIII"
    "187"     "CLXXXVII"
    "828"     "DCCCXXVIII"
    "1"       "I"
    "217"     "CCXVII"
    "309"     "CCCIX"
    "489"     "CDLXXXIX"
    "406"     "CDVI"
    "840"     "DCCCXL"
    "757"     "DCCLVII"
    "268"     "CCLXVIII"
    "371"     "CCCLXXI"
    "988"     "CMLXXXVIII"
```

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{idx,1};
strTarget = dataTest{idx,2};

documentsSource = transformText(strSource,startToken,stopToken);
```

Convert the tokenized text into a batch of padded sequences by using the `doc2sequence` function. To automatically pad the sequences, set the `'PaddingDirection'` option to `'right'` and set the padding value to the input size (the token index of the padding token).

```
sequencesSource = doc2sequence(encSource,documentsSource, ...
    'PaddingDirection','right', ...
    'PaddingValue',inputSize);
```

Concatenate and permute the sequence data into the required shape for the encoder model function (1-by-*N*-by-*S*, where *N* is the number of observations and *S* is the sequence length).

```
XSource = cat(3,sequencesSource{:});
XSource = permute(XSource,[1 3 2]);
```

Convert input data to `dlarray` and calculate the encoder model outputs.

```
dlXSource = dlarray(XSource);
[dlZ, hiddenState] = modelEncoder(dlXSource, parametersEncoder);
```

Translate Source Text

To generate translations for new data input the sequences into the encoder-decoder model and convert the resulting sequences back into text using the token indices.

To initialize the translations, create a vector containing only the indices corresponding to the start token.

```
decoderInput = repmat(word2ind(encTarget,startToken),[1 numObservationsTest]);
decoderInput = dlarray(decoderInput);
```

Initialize the context vector and the cell arrays containing the translated sequences and the attention scores for each observation.

```
context = dlarray(zeros([size(dlZ, 1) numObservationsTest]));
sequencesTranslated = cell(1,numObservationsTest);
attentionScores = cell(1,numObservationsTest);
```

Loop over time steps and translate the sequences. Keep looping over the time steps until all sequences translated. For each observation, when the translation is finished (when the decoder predicts the stop token), set a flag to stop translating that sequence.

```
stopIdx = word2ind(encTarget,stopToken);
stopTranslating = false(1, numObservationsTest);

while ~all(stopTranslating)

    % Forward through decoder.
    [dlY, context, hiddenState, attn] = modelDecoder(decoderInput, parametersDecoder, c
        hiddenState, dlZ);

    % Loop over observations.
    for i = 1:numObservationsTest
        % Skip already-translated sequences.
        if stopTranslating(i)
            continue
        end
```

```
        % Update attention scores.
        attentionScores{i} = [attentionScores{i} extractdata(attn(:,i))];

        % Predict next time step.
        prob = softmax(dlY(:,i), 'DataFormat', 'CB');
        [~, idx] = max(prob(1:end-1,:), [], 1);

        % Set stopTranslating flag when translation done.
        if idx == stopIdx
            stopTranslating(i) = true;
        else
            sequencesTranslated{i} = [sequencesTranslated{i} extractdata(idx)];
            decoderInput(i) = idx;
        end
    end
end
```

View the source text, target text, and translations in a table.

```
tbl = table;
tbl.Source = strSource;
tbl.Target = strTarget;
tbl.Translated = cellfun(@(sequence) join(ind2word(encTarget,sequence),""),sequencesTra
tbl
```

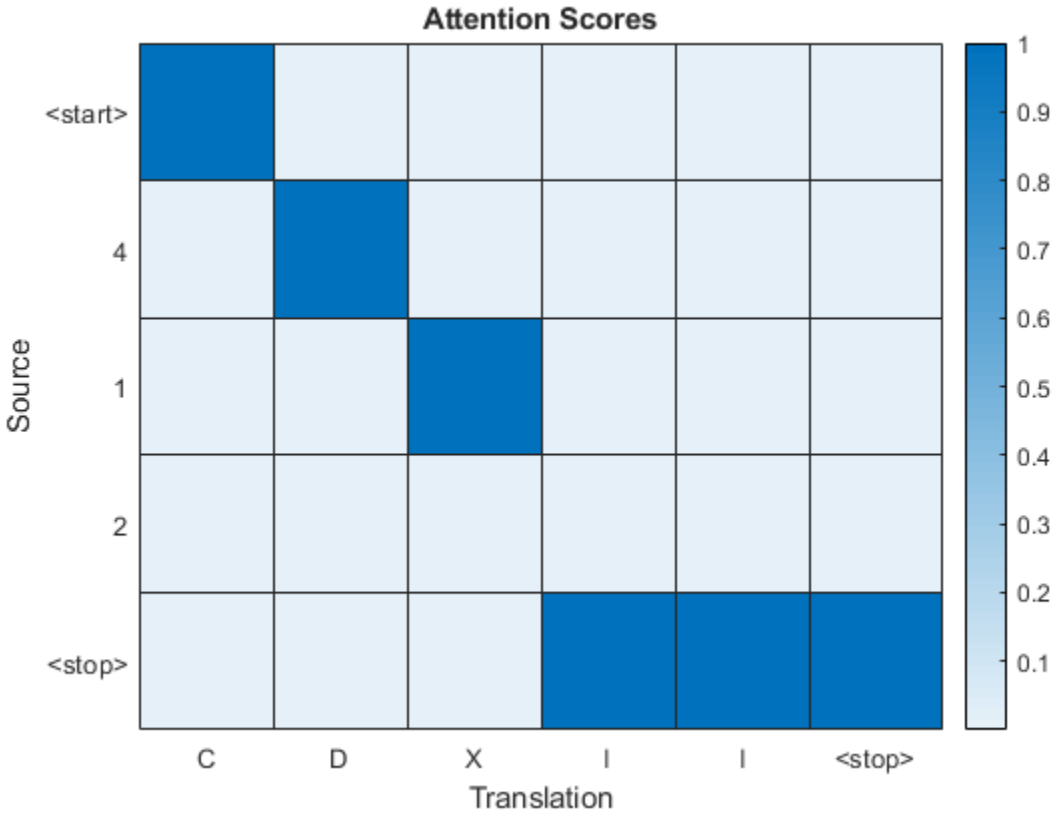
```
tbl=16×3 table
    Source       Target         Translated
    ______    _____________    _____________

    "412"     "CDXII"          "CDXII"
    "274"     "CCLXXIV"        "CCLXXIV"
    "231"     "CCXXXI"         "CCXXXI"
    "558"     "DLVIII"         "DLVIII"
    "187"     "CLXXXVII"       "CLXXXVII"
    "828"     "DCCCXXVIII"     "DCCCXXVIII"
    "1"       "I"              "CII"
    "217"     "CCXVII"         "CCXVII"
    "309"     "CCCIX"          "CCCIX"
    "489"     "CDLXXXIX"       "CDLXXXIX"
    "406"     "CDVI"           "CDVI"
    "840"     "DCCCXL"         "DCCCXL"
    "757"     "DCCLVII"        "DCCLVII"
    "268"     "CCLXVIII"       "CCLXVIII"
    "371"     "CCCLXXI"        "CCCLXXI"
```

```
    "988"     "CMLXXXVIII"    "CMLXXXVIII"
```

Plot Attention Scores

Plot the attention scores of the first sequence in a heat map. The attention scores highlight which areas of the source and translated sequences the model attends to when processing the translation.

```
idx = 1;
figure
xlabs = [ind2word(encTarget,sequencesTranslated{idx}) stopToken];
ylabs = string(documentsSource(idx));

heatmap(attentionScores{idx}, ...
    'CellLabelColor','none', ...
    'XDisplayLabels',xlabs, ...
    'YDisplayLabels',ylabs);

xlabel("Translation")
ylabel("Source")
title("Attention Scores")
```



Preprocessing Function

The `preprocessSourceTargetPairs` takes a table `data` containing the source-target pairs in two columns and for each column returns sequences of token indices and a corresponding `wordEncoding` object that maps the indices to words and vice versa.

```
function [sequencesSource, sequencesTarget, encSource, encTarget] = preprocessSourceTar

% Extract text data.
strSource = data{:,1};
strTarget = data{:,2};

% Create tokenized document arrays.
documentsSource = transformText(strSource,startToken,stopToken);
```

```
documentsTarget = transformText(strTarget,startToken,stopToken);

% Create word encodings.
encSource = wordEncoding(documentsSource);
encTarget = wordEncoding(documentsTarget);

% Convert documents to numeric sequences.
sequencesSource = doc2sequence(encSource, documentsSource,'PaddingDirection','none');
sequencesTarget = doc2sequence(encTarget, documentsTarget,'PaddingDirection','none');

end
```

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 spliting the text into words instead of characters, skip the first step.

```
function documents = transformText(str,startToken,stopToken)

% Split text into characters.
str = strip(replace(str,"",""));

% Add start and stop tokens.
str = startToken + str + stopToken;

% Create tokenized document array.
documents = tokenizedDocument(str,'CustomTokens',[startToken stopToken]);

end
```

Batch Creation Function

The `createBatch` function takes a mini-batch of source and target sequences and returns padded sequences with the corresponding padding masks.

```
function [XSource, XTarget, maskSource, maskTarget] = createBatch(sequencesSource, sequ
    paddingValueSource, paddingValueTarget)

numObservations = size(sequencesSource,1);
sequenceLengthSource = max(cellfun(@(x) size(x,2), sequencesSource));
sequenceLengthTarget = max(cellfun(@(x) size(x,2), sequencesTarget));

% Initialize masks.
maskSource = false(numObservations, sequenceLengthSource);
```

```
maskTarget = false(numObservations, sequenceLengthTarget);

% Initialize mini-batch.
XSource = zeros(1,numObservations,sequenceLengthSource);
XTarget = zeros(1,numObservations,sequenceLengthTarget);

% Pad sequences and create masks.
for i = 1:numObservations

    % Source
    L = size(sequencesSource{i},2);
    paddingSize = sequenceLengthSource - L;
    padding = repmat(paddingValueSource, [1 paddingSize]);

    XSource(1,i,:) = [sequencesSource{i} padding];
    maskSource(i,1:L) = true;

    % Target
    L = size(sequencesTarget{i},2);
    paddingSize = sequenceLengthTarget - L;
    padding = repmat(paddingValueTarget, [1 paddingSize]);

    XTarget(1,i,:) = [sequencesTarget{i} padding];
    maskTarget(i,1:L) = true;
end

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.

```
function [dlZ, hiddenState] = modelEncoder(dlX, parametersEncoder, maskSource)

% Embedding
weights = parametersEncoder.emb.Weights;
dlZ = embedding(dlX,weights);

% LSTM
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
inputWeights = parametersEncoder.lstm2.InputWeights;
recurrentWeights = parametersEncoder.lstm2.RecurrentWeights;
bias = parametersEncoder.lstm2.Bias;

[dlZ, hiddenState] = lstm(dlZ,initialHiddenState, initialCellState, ...
    inputWeights, recurrentWeights, bias, 'DataFormat', 'CBT');

% Mask output for training
if nargin > 2
    dlZ = dlZ.*permute(maskSource, [3 1 2]);
    sequenceLengths = sum(maskSource, 2);

    % Mask final hidden state
    for ii = 1:size(dlZ, 2)
        hiddenState(:, ii) = dlZ(:, ii, sequenceLengths(ii));
    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(dlX, parameters, c
    hiddenState, encoderOutputs, dropout)

% Embedding
weights = parameters.emb.Weights;
dlX = embedding(dlX, weights);

% RNN input
dlY = cat(1, dlX, context);
```

```
% LSTM 1
initialCellState = dlarray(zeros(size(hiddenState)));

inputWeights = parameters.lstm1.InputWeights;
recurrentWeights = parameters.lstm1.RecurrentWeights;
bias = parameters.lstm1.Bias;

dlY = lstm(dlY, hiddenState, initialCellState, inputWeights, ...
    recurrentWeights, bias, 'DataFormat', 'CBT');

if nargin > 5
    % Dropout
    mask = ( rand(size(dlY), 'like', dlY) > dropout );
    dlY = dlY.*mask;
end

% LSTM 2
inputWeights = parameters.lstm2.InputWeights;
recurrentWeights = parameters.lstm2.RecurrentWeights;
bias = parameters.lstm2.Bias;
[~, hiddenState] = lstm(dlY, hiddenState, initialCellState, ...
    inputWeights, recurrentWeights, bias, 'DataFormat', 'CBT');

% Attention
weights = parameters.attn.Weights;
[attentionScores, N] = attention(hiddenState, encoderOutputs, weights);

% Context
encoderOutputs = permute(encoderOutputs, [1 3 2]);
for ii = 1:N
    context(:, ii) = encoderOutputs(:, :, ii)*attentionScores(:, ii);
end

% Fully connect
weights = parameters.fc.Weights;
bias = parameters.fc.Bias;
dlY = weights*cat(1, hiddenState, context) + bias;

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 out batch and sequence dimensions
Z = reshape(Z, [], N, T);
end
```

Attention Function

The `attention` function computes the attention scores accoring to Luong "general" scoring.

```
function [attentionScores, N] = attention(hiddenState, encoderOutputs, weights)

[N, S] = size(encoderOutputs, 2:3);
attentionEnergies = dlarray(zeros( [S N] ));
for tt = 1:S
    % The energy at each time step is the dot product of the hidden state
    % and the learnable attention weights times the encoder output
    attentionEnergies(tt, :) = sum(hiddenState.*(weights*encoderOutputs(:, :, tt)), 1);
end

% Compute softmax scores
attentionScores = softmax(attentionEnergies, 'DataFormat', 'CB');
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 [gradientsEncoder, gradientsDecoder, maskedLoss] = modelGradients(parametersEn
    parametersDecoder, dlXSource, dlXTarget, maskSource, maskTarget, dropout)

% Forward through encoder.
[dlZ, hiddenState] = modelEncoder(dlXSource, parametersEncoder, maskSource);

% Get parameter sizes.
```

```
[miniBatchSize, sequenceLength] = size(dlXTarget,2:3);
sequenceLength = sequenceLength - 1;
numHiddenUnits = size(dlZ,1);

% Initialize context vector.
context = dlarray(zeros([numHiddenUnits miniBatchSize]));

% Initialize loss.
loss = dlarray(zeros([miniBatchSize sequenceLength]));

% Get first time step for decoder.
decoderInput = dlXTarget(:,:,1);

% Choose whether to use teacher forcing.
doTeacherForcing = rand < 0.5;

if doTeacherForcing
    for t = 1:sequenceLength
        % Forward through decoder.
        [dlY, context, hiddenState] = modelDecoder(decoderInput, parametersDecoder, cor
            hiddenState, dlZ, dropout);

        % Update loss.
        dlT = dlarray(oneHot(dlXTarget(:,:,t+1), size(dlY,1)));
        loss(:,t) = crossEntropyAndSoftmax(dlY, dlT);

        % Get next time step.
        decoderInput = dlXTarget(:,:,t+1);
    end
else
    for t = 1:sequenceLength
        % Forward through decoder.
        [dlY, context, hiddenState] = modelDecoder(decoderInput, parametersDecoder, cor
            hiddenState, dlZ, dropout);

        % Update loss.
        dlT = dlarray(oneHot(dlXTarget(:,:,t+1), size(dlY,1)));
        loss(:,t) = crossEntropyAndSoftmax(dlY, dlT);

        % Greedily update next input time step.
        prob = softmax(dlY,'DataFormat','CB');
        [~, decoderInput] = max(prob,[],1);
    end
end
```

```
% Determine masked loss.
maskedLoss = sum(sum(loss.*maskTarget(:,2:end))) / miniBatchSize;

% Update gradients.
[gradientsEncoder, gradientsDecoder] = dlgradient(maskedLoss, parametersEncoder, parame

% For plotting, return loss normalized by sequence length.
maskedLoss = extractdata(maskedLoss) ./ sequenceLength;

end
```

Cross-Entropy and Softmax Loss Function

The `crossEntropyAndSoftmax` loss computes the cross-entropy and softmax loss.

```
function loss = crossEntropyAndSoftmax(dlY, dlT)

offset = max(dlY);
logSoftmax = dlY - offset - log(sum(exp(dlY-offset)));
loss = -sum(dlT.*logSoftmax);

end
```

Uniform Noise Function

The `uniformNoise` function samples weights from a uniform distribution.

```
function weights = uniformNoise(sz, k)

weights = -sqrt(k) + 2*sqrt(k).*rand(sz);

end
```

Gradient Clipping Function

The `clipGradient` function clips the model gradients.

```
function g = clipGradient(g, gradientThreshold)

wnorm = norm(extractdata(g));
if wnorm > gradientThreshold
    g = (gradientThreshold/wnorm).*g;
end

end
```

One-Hot Encoding Function

The `oneHot` function encodes word indices as one-hot vectors.

```
function oh = oneHot(idx, numTokens)
tokens = (1:numTokens)';
oh = (tokens == idx);
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 1-12
- “Analyze Text Data Using Topic Models” on page 2-18
- “Classify Text Data Using Deep Learning” on page 2-57
- “Classify Text Data Using Convolutional Neural Network” on page 2-69
- “Train a Sentiment Classifier” on page 2-47
- “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 2-57 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 `weatherReportsTrain.csv`. Specify to read the data from the `"event_narrative"` and `"event_type"` columns only.

```
filenameTrain = "weatherReportsTrain.csv";
textName = "event_narrative";
labelName = "event_type";
ttdsTrain = tabularTextDatastore(filenameTrain,'SelectedVariableNames',[textName labelN
```

View a preview of the datastore.

```
preview(ttdsTrain)
```

```
ans=8×2 table
```

'Large tree down between Plantersville and Nettleton.'	
'One to two feet of deep standing water developed on a street on the Winthrop Unive	
'NWS Columbia relayed a report of trees blown down along Tom Hall St.'	
'Media reported two trees blown down along I-40 in the Old Fort area.'	
'A few tree limbs greater than 6 inches down on HWY 18 in Roseland.'	
'Awning blown off a building on Lamar Avenue. Multiple trees down near the intersec	
'Tin roof ripped off house on Old Memphis Road near Billings Drive. Several large t	
'Powerlines down at Walnut Grove and Cherry Lane roads.'	

Transform Datastore

Create a custom transform function that converts data read from the datastore to a table containing the predictors and the responses. The `transformTextData` 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 tablular 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 = 128;
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) transformTextData(data,emb,classNames))
```

```
tdsTrain =
  TransformedDatastore with properties:

    UnderlyingDatastore: [1×1 matlab.io.datastore.TabularTextDatastore]
             Transforms: {@(data)transformTextData(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
    ________________    _________________

    [300×164 single]    Thunderstorm Wind
    [300×164 single]    Heavy Rain
    [300×164 single]    Thunderstorm Wind
    [300×164 single]    Thunderstorm Wind
    [300×164 single]    Thunderstorm Wind
    [300×164 single]    Thunderstorm Wind
    [300×164 single]    Thunderstorm Wind
    [300×164 single]    Thunderstorm Wind
```

Create a transformed datastore containing the validation data in `weatherReportsValidation.csv` using the same steps.

```
filenameValidation = "weatherReportsValidation.csv";
ttdsValidation = tabularTextDatastore(filenameValidation,'SelectedVariableNames',[textN
ttdsValidation.ReadSize = miniBatchSize;
tdsValidation = transform(ttdsValidation, @(data) transformTextData(data,emb,classNames
```

```
tdsValidation =
  TransformedDatastore with properties:
```

```
    UnderlyingDatastore: [1×1 matlab.io.datastore.TabularTextDatastore]
             Transforms: {@(data)transformTextData(data,emb,classNames)}
            IncludeInfo: 0
```

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`.

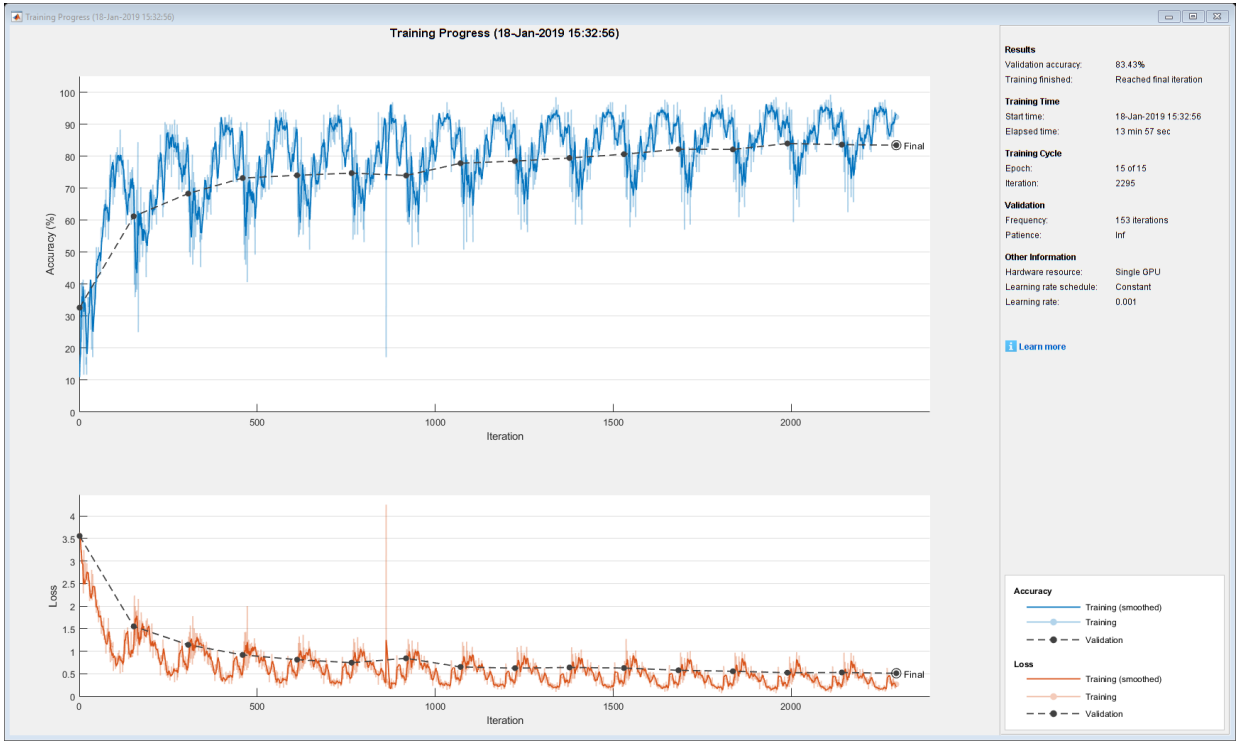
By default, `trainNetwork` uses a GPU if one is available (requires Parallel Computing Toolbox™ and a CUDA® enabled GPU with compute capability 3.0 or higher). Otherwise, it uses the CPU. To specify the execution environment manually, use the `'ExecutionEnvironment'` name-value pair argument of `trainingOptions`. 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', ...
    'ValidationData',tdsValidation, ...
    'ValidationFrequency',numIterationsPerEpoch, ...
```

```
    'Plots','training-progress', ...
    'Verbose',false);
```

Train the LSTM network using the `trainNetwork` function.

```
net = trainNetwork(tdsTrain,layers,options);
```



Test LSTM Network

Create a transformed datastore containing the held-out test data in `weatherReportsTest.csv`.

```
filenameTest = "weatherReportsTest.csv";
ttdsTest = tabularTextDatastore(filenameTest,'SelectedVariableNames',[textName labelNar
ttdsTest.ReadSize = miniBatchSize;
tdsTest = transform(ttdsTest, @(data) transformTextData(data,emb,classNames))
```

```
tdsTest =
  TransformedDatastore with properties:
```

```
    UnderlyingDatastore: [1×1 matlab.io.datastore.TabularTextDatastore]
             Transforms: {@(data)transformTextData(data,emb,classNames)}
           IncludeInfo: 0
```

Read the labels from the `tabularTextDatastore`.

```
labelsTest = readLabels(ttdsTest,labelName);
YTest = categorical(labelsTest,classNames);
```

Make predictions on the test data using the trained network.

```
YPred = classify(net,tdsTest,'MiniBatchSize',miniBatchSize);
```

Calculate the classification accuracy on the test data.

```
accuracy = mean(YPred == YTest)
```

```
accuracy = 0.8293
```

Functions

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
```

The `transformTextData` 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 = transformTextData(data,emb,classNames)

% Preprocess documents.
textData = data{:,1};
```

```
textData = lower(textData);
documents = tokenizedDocument(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
```

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 2-18
- “Analyze Text Data Using Multiword Phrases” on page 2-9
- “Train a Sentiment Classifier” on page 2-47
- “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.

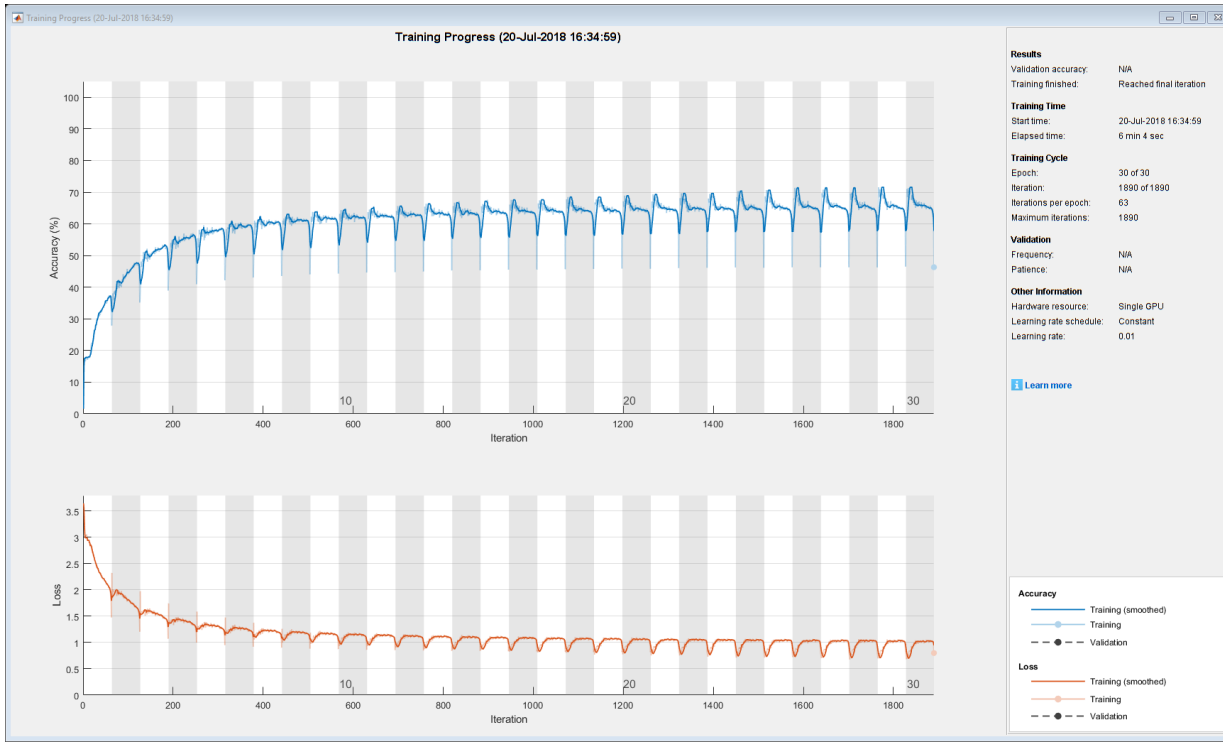
```
inputSize = size(XTrain{1},1);
numClasses = numel(categories([YTrain{:}]));
numCharacters = max([textData{:}]);

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
```

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 2-120
- “Create Simple Text Model for Classification” on page 2-2
- “Analyze Text Data Using Topic Models” on page 2-18
- “Analyze Text Data Using Multiword Phrases” on page 2-9
- “Train a Sentiment Classifier” on page 2-47
- “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 o
    "So she was considering in her own mind (as well as she could, for the hot day made
    "There was nothing so very remarkable in that; nor did Alice think it so very much
    "In another moment down went Alice after it, never once considering how in the wor
```

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 o
    "So she was considering in her own mind (as well as she could, for the hot day made
    "There was nothing so very remarkable in that; nor did Alice think it so very much
    "In another moment down went Alice after it, never once considering how in the wor
    "The rabbit-hole went straight on like a tunnel for some way, and then dipped sudde
    "Either the well was very deep, or she fell very slowly, for she had plenty of time
    "'Well!' thought Alice to herself, 'after such a fall as this, I shall think nothin
    "Down, down, down. Would the fall never come to an end! 'I wonder how many miles I
    "Presently she began again. 'I wonder if I shall fall right through the earth! How
    "Down, down, down. There was nothing else to do, so Alice soon began talking again
```

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.

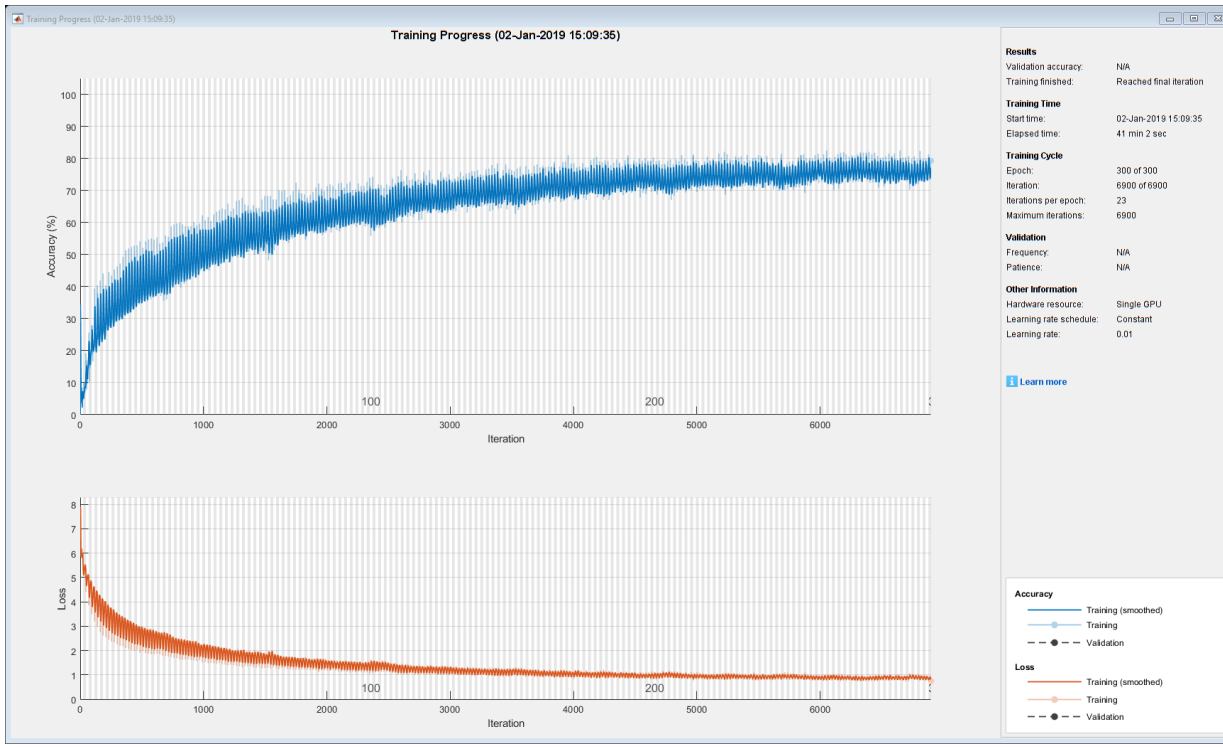
```
inputSize = 1;
embeddingDimension = 100;
numWords = numel(ds.Encoding.Vocabulary);
numClasses = numWords + 1;

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 replacing 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 2-18
- “Analyze Text Data Using Multiword Phrases” on page 2-9
- “Train a Sentiment Classifier” on page 2-47
- “Sequence Classification Using Deep Learning” (Deep Learning Toolbox)
- “Deep Learning in MATLAB” (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 2-57 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 `"weatherReportsTrain.csv"` and that the text and labels are in the columns `"event_narrative"` and `"event_type"` respectively.

```
filenameTrain = "weatherReportsTrain.csv";
textName = "event_narrative";
labelName = "event_type";
dsTrain = textDatastore(filenameTrain,textName,labelName,emb)
```

```
dsTrain =
  textDatastore with properties:

            ClassNames: [1×39 string]
             Datastore: [1×1 matlab.io.datastore.TransformedDatastore]
    EmbeddingDimension: 300
             LabelName: "event_type"
         MiniBatchSize: 128
            NumClasses: 39
       NumObservations: 19683
```

Create a datastore containing the validation data from the CSV file `"weatherReportsValidation.csv"` using the same steps.

```
filenameValidation = "weatherReportsValidation.csv";
dsValidation = textDatastore(filenameValidation,textName,labelName,emb)
```

```
dsValidation =
  textDatastore with properties:

            ClassNames: [1×39 string]
             Datastore: [1×1 matlab.io.datastore.TransformedDatastore]
    EmbeddingDimension: 300
             LabelName: "event_type"
         MiniBatchSize: 128
            NumClasses: 39
       NumObservations: 4218
```

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];
```

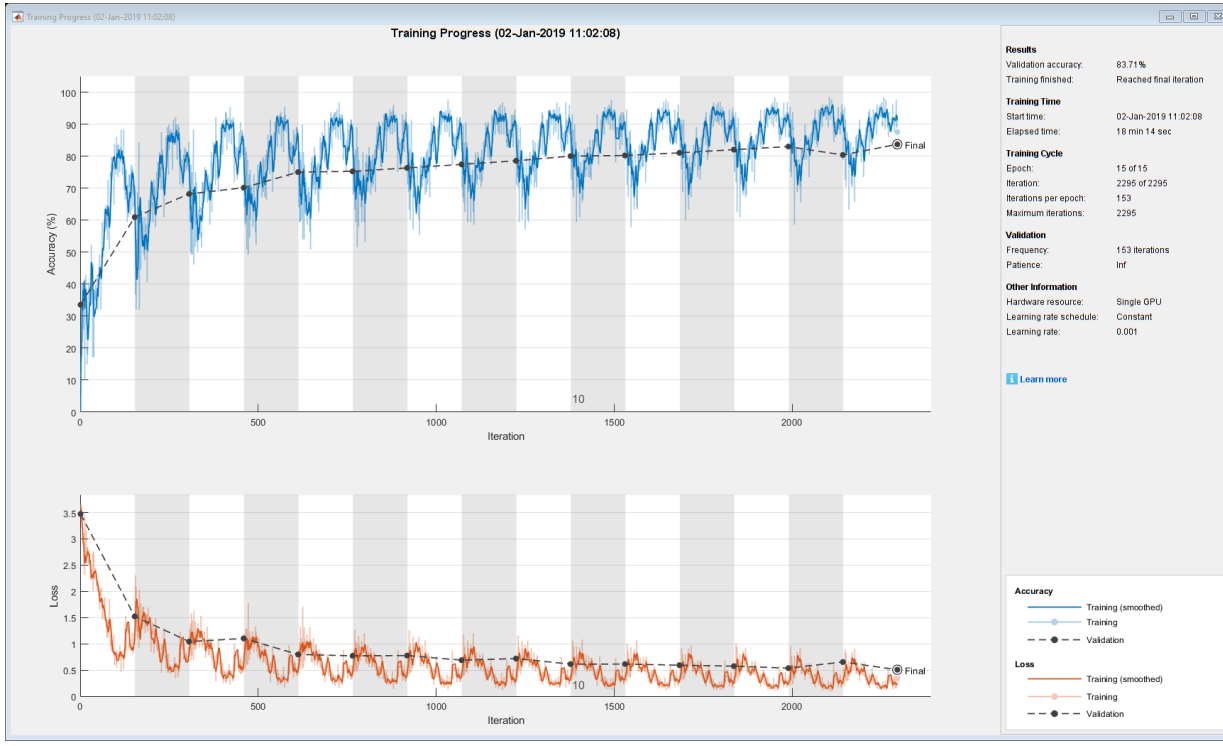
Specify the training options. Specify the solver to be `'adam'` and the gradient threshold to be 2. The datastore `textDatastore.m` does not support shuffling, so set `'Shuffle'`, to `'never'`. For an example showing how to implement a datastore with support for shuffling, see “Develop Custom Mini-Batch Datastore” (Deep Learning Toolbox). 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.

```
miniBatchSize = 128;
numObservations = dsTrain.NumObservations;
numIterationsPerEpoch = floor(numObservations / miniBatchSize);

options = trainingOptions('adam', ...
    'MaxEpochs',15, ...
    'MiniBatchSize',miniBatchSize, ...
    'GradientThreshold',2, ...
    'Shuffle','never', ...
    'ValidationData',dsValidation, ...
    'ValidationFrequency',numIterationsPerEpoch, ...
    'Plots','training-progress', ...
    'Verbose',false);
```

Train the LSTM network using the `trainNetwork` function.

```
net = trainNetwork(dsTrain,layers,options);
```



Test LSTM Network

Create a datastore containing the documents and the labels

```
filenameTest = "weatherReportsTest.csv";
dsTest = textDatastore(filenameTest,textName,labelName,emb)
```

```
dsTest =
  textDatastore with properties:

            ClassNames: [1×39 string]
             Datastore: [1×1 matlab.io.datastore.TransformedDatastore]
    EmbeddingDimension: 300
             LabelName: "event_type"
         MiniBatchSize: 128
            NumClasses: 39
       NumObservations: 4217
```

Read the labels from the datastore using the `readLabels` function of the custom datastore.

```
YTest = readLabels(dsTest);
```

Classify the test documents using the trained LSTM network.

```
YPred = classify(net,dsTest);
```

Calculate the classification accuracy. The accuracy is the proportion of labels that the network predicts correctly.

```
accuracy = mean(YPred == YTest)
```

```
accuracy = 0.8084
```

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 2-18
- “Analyze Text Data Using Multiword Phrases” on page 2-9
- “Train a Sentiment Classifier” on page 2-47
- “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 `weatherReports.csv` contains weather reports, including a text description and categorical labels for each event.

```
filename = "weatherReports.csv";
T = readtable(filename,'TextType','string');
```

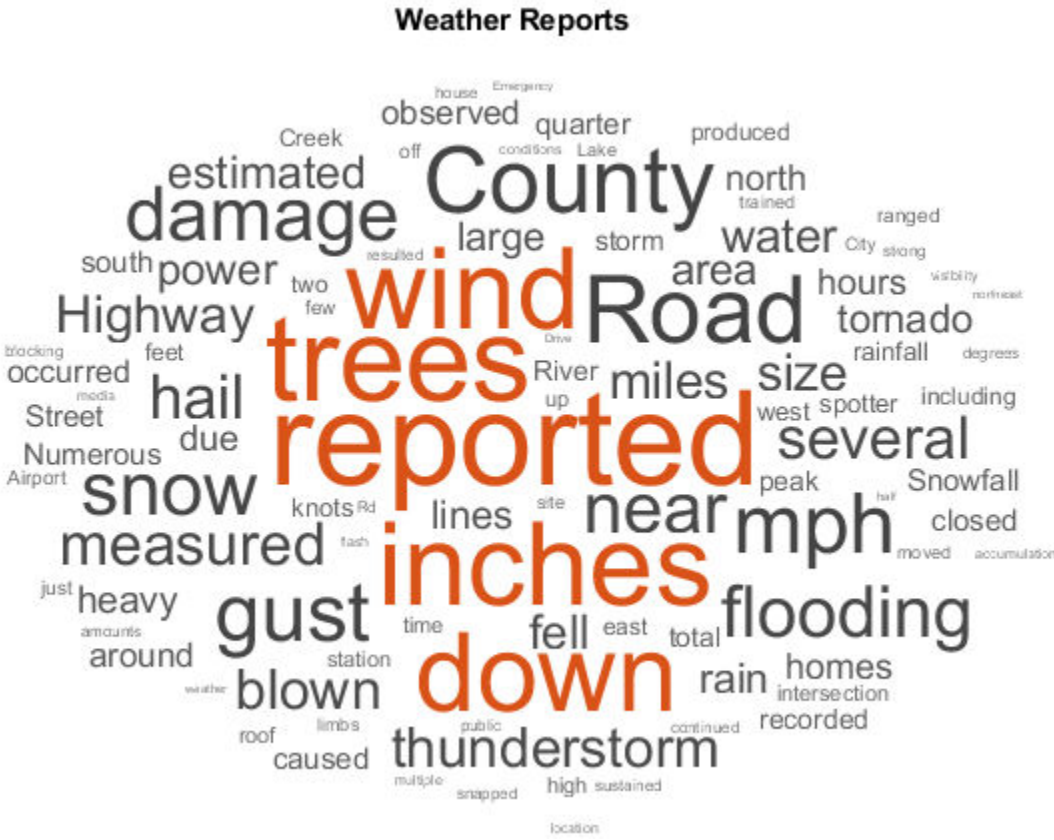
Extract the text data from the `event_narrative` column.

```
textData = T.event_narrative;
textData(1:10)
```

```
ans = 10x1 string array
    "Large tree down between Plantersville and Nettleton."
    "One to two feet of deep standing water developed on a street on the Winthrop Unive
    "NWS Columbia relayed a report of trees blown down along Tom Hall St."
    "Media reported two trees blown down along I-40 in the Old Fort area."
    ""
    "A few tree limbs greater than 6 inches down on HWY 18 in Roseland."
    "Awning blown off a building on Lamar Avenue. Multiple trees down near the intersec
    "Quarter size hail near Rosemark."
    "Tin roof ripped off house on Old Memphis Road near Billings Drive. Several large t
    "Powerlines down at Walnut Grove and Cherry Lane roads."
```

Create a word cloud from all the weather reports.

```
figure
wordcloud(textData);
title("Weather Reports")
```



Compare the words in the reports with labels `"Hail"` and `"Thunderstorm Wind"`. 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.

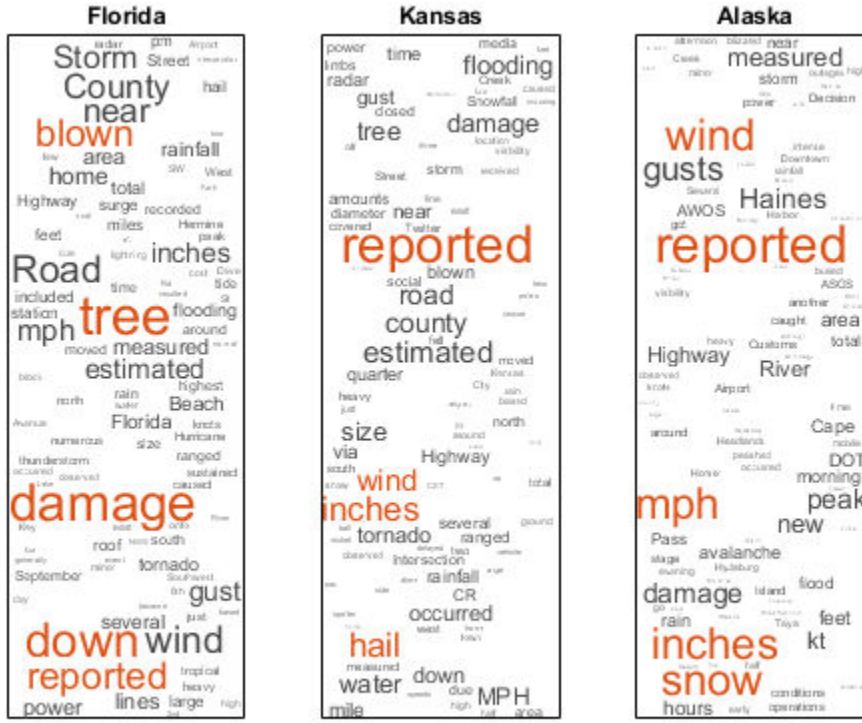
```
figure
labels = T.event_type;

subplot(1,2,1)
idx = labels == "Hail";
wordcloud(textData(idx),'Color','blue');
title("Hail")

subplot(1,2,2)
idx = labels == "Thunderstorm Wind";
```

```
wordcloud(textData(idx),'Color','magenta');
title("Thunderstorm Wind")
```

Compare the words in the reports from the states Florida, Kansas, and Alaska. Create word clouds of the reports for each of these states in rectangles and draw a border around each word cloud.

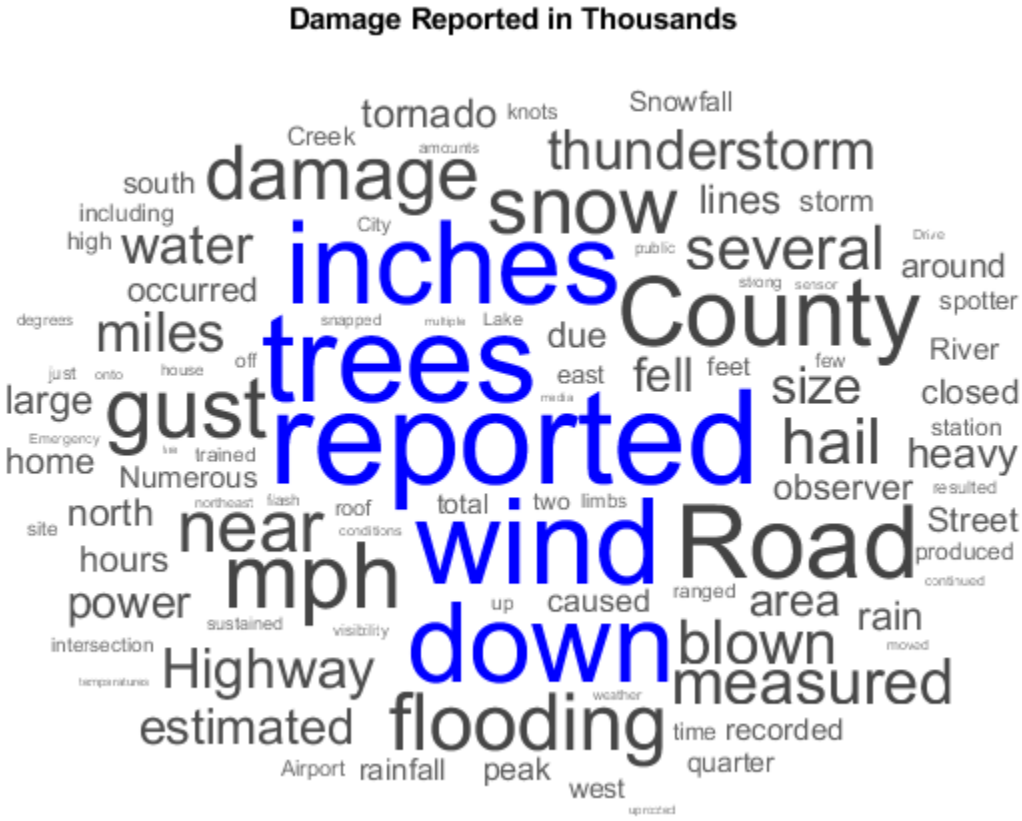
```
figure
state = T.state;

subplot(1,3,1)
idx = state == "FLORIDA";
wordcloud(textData(idx),'Shape','rectangle','Box','on');
title("Florida")
```

```
subplot(1,3,2)
idx = state == "KANSAS";
wordcloud(textData(idx),'Shape','rectangle','Box','on');
title("Kansas")

subplot(1,3,3)
idx = state == "ALASKA";
wordcloud(textData(idx),'Shape','rectangle','Box','on');
title("Alaska")
```

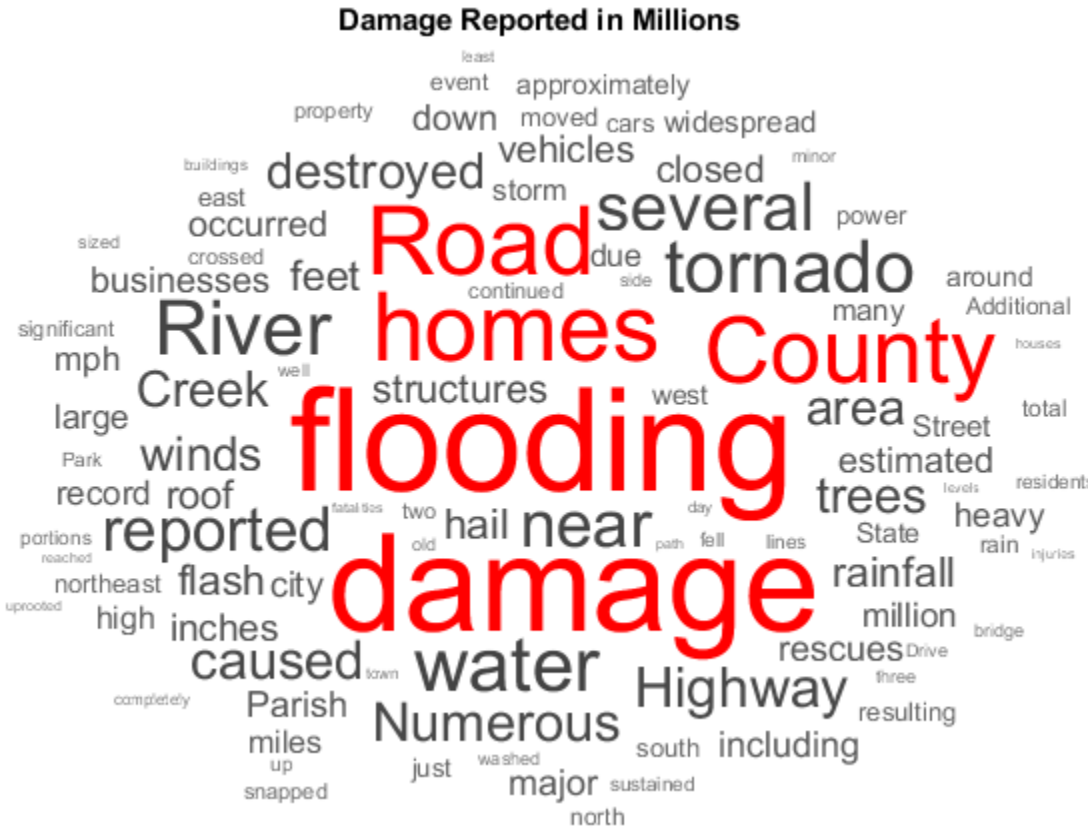


Compare the words in the reports with property damage reported in thousands of dollars to the reports with damage reported in millions of dollars. Create word clouds of the reports for each of these amounts with highlight color blue and red respectively.

```
cost = T.damage_property;
idx = endsWith(cost,"K");
figure
wordcloud(textData(idx),'HighlightColor','blue');
title("Damage Reported in Thousands")
```




```
idx = endsWith(cost,"M");
figure
wordcloud(textData(idx),'HighlightColor','red');
title("Damage Reported in Millions")
```



See Also

`bagOfWords` | `tokenizedDocument` | `wordcloud`

Related Examples

- “Prepare Text Data for Analysis” on page 1-12
- “Analyze Text Data Using Topic Models” on page 2-18
- “Classify Text Data Using Deep Learning” on page 2-57
- “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 $Italy - Rome + Paris = France$.

To reproduce the results in this example, set `rng` to `'default'`.

```
rng('default')
```

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    0.0
```

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 500 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)
```

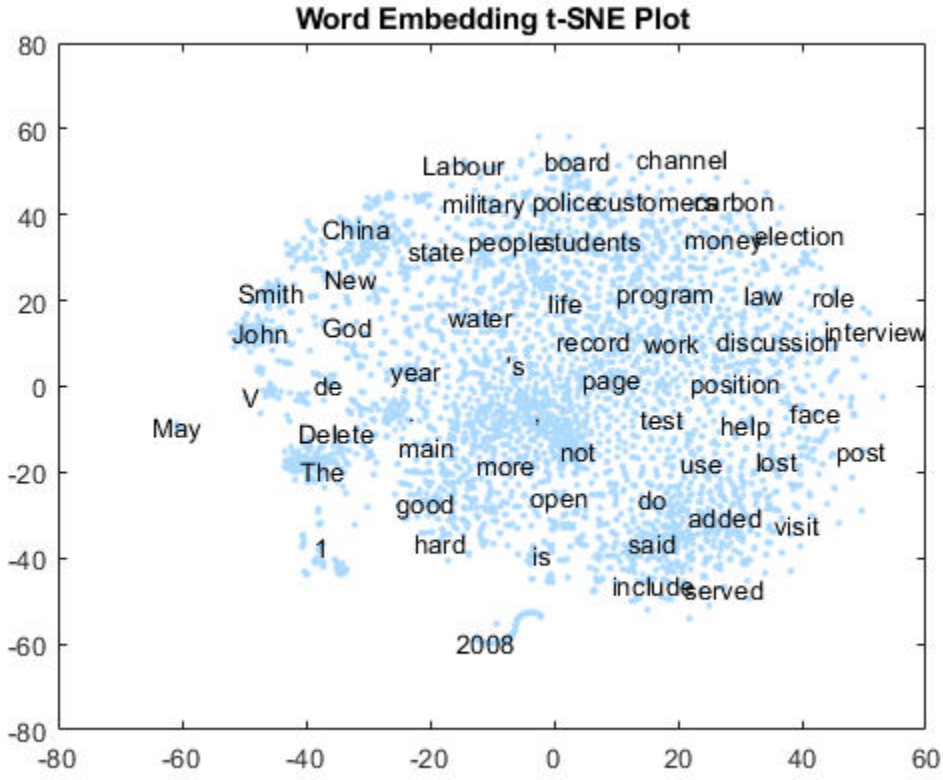
```
ans = 1×2

        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);
```

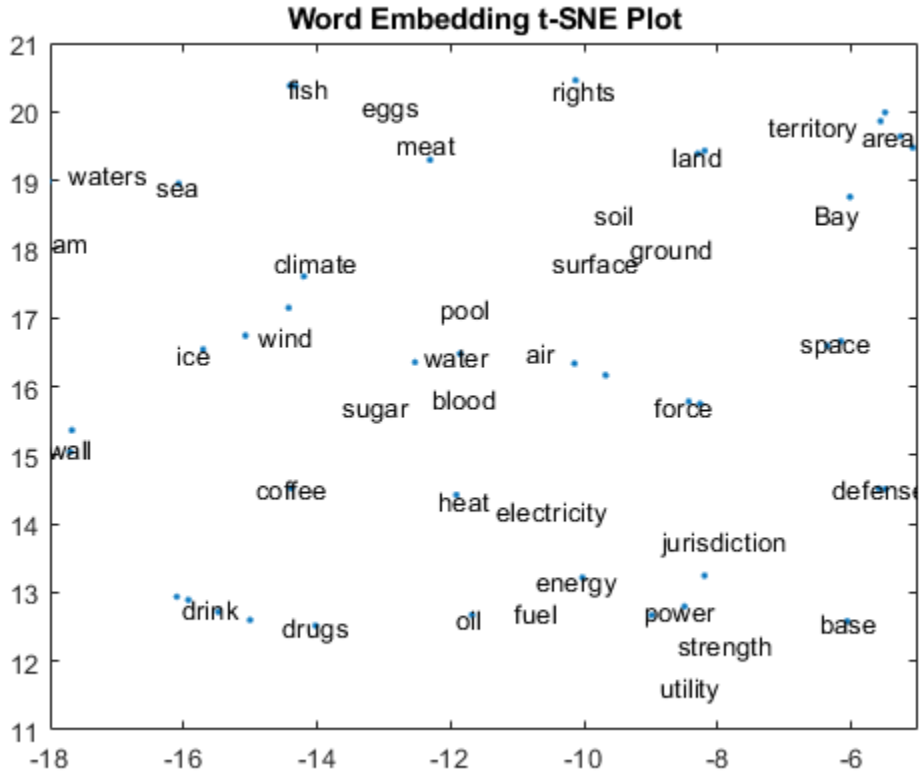
Plot the words at the coordinates specified by `XY` in a 2-D text scatter plot. For readability, `textscatter`, by default, does not display all of the input words and displays markers instead.

```
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
```

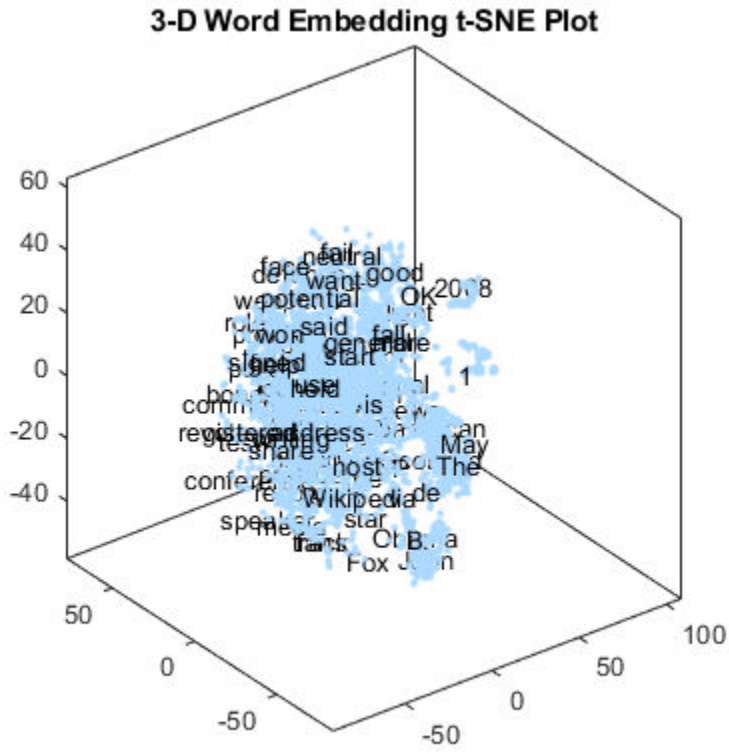
```
        5000         300
```

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);
```

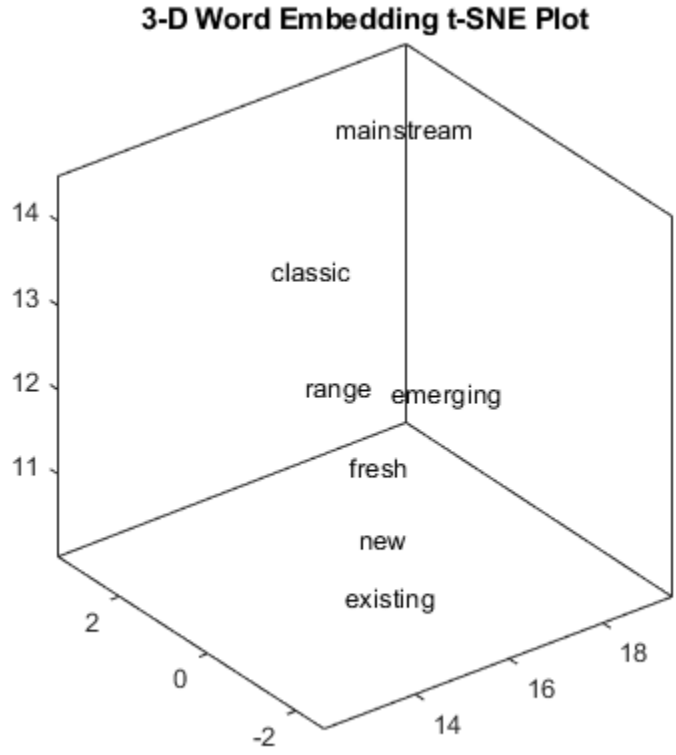
Plot the words at the coordinates specified by XYZ in a 3-D text scatter plot.

```
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)
```

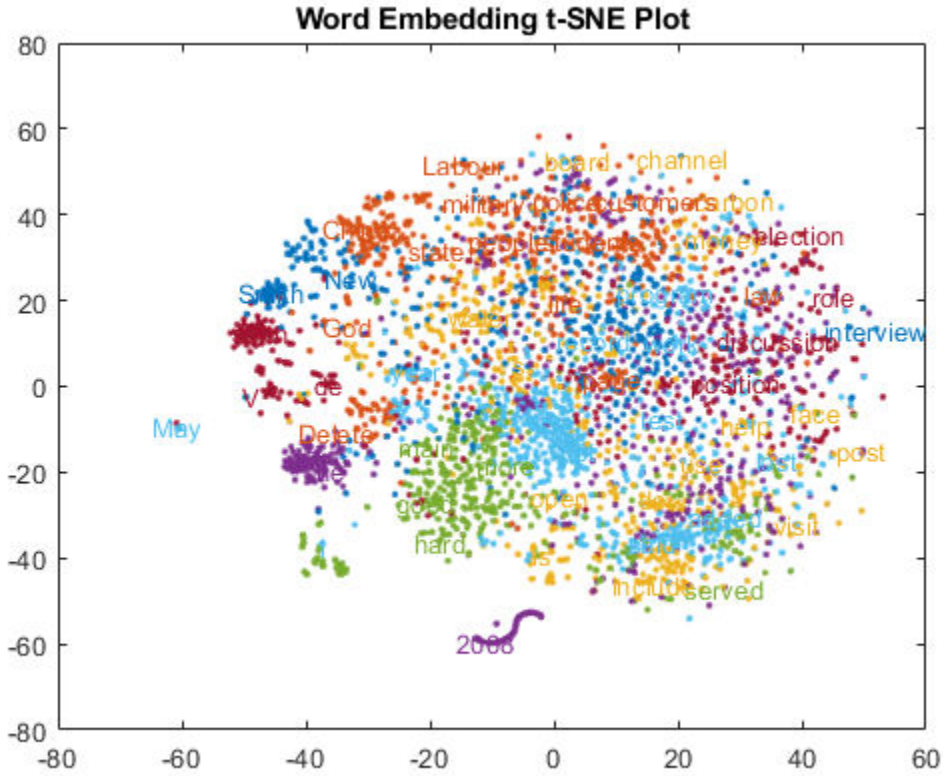
```
ans = 1×2

        5000         300
```

Discover 25 clusters using `kmeans`.

```
cidx = kmeans(V,25,'dist','sqeuclidean');
```

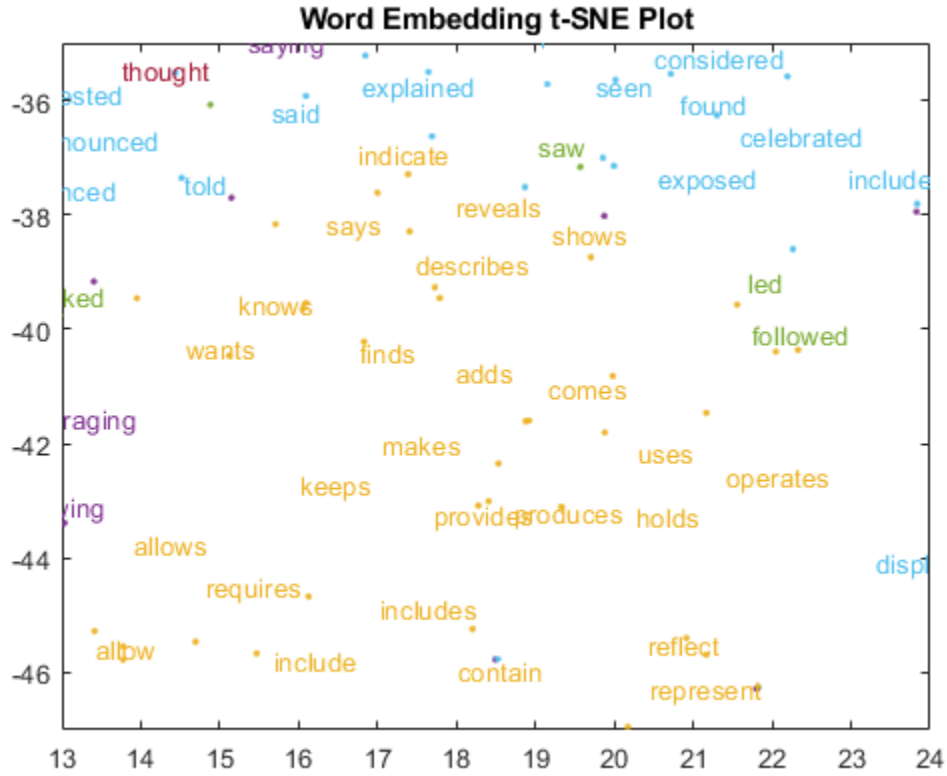
Visualize the clusters in a text scatter plot using the 2-D t-SNE data coordinates calculated earlier.

```
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 1-12
- “Visualize Text Data Using Word Clouds” on page 3-2
- “Classify Text Data Using Deep Learning” on page 2-57

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.

Feature	Language Consideration	Workaround
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`.
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`.

Feature	Language Consideration	Workaround
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`.

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 4-12
- “German Language Support” on page 4-26
- “Analyze German Text Data” on page 4-33

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 4-12.

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
    ________    ______________    __________    ___________    ________    ____________

    "恋"              1               1         letters          ja        noun
    "に"              1               1         letters          ja        adposition
    "悩み"            1               1         letters          ja        verb
    "、"              1               1         punctuation      ja        punctuation
    "苦しむ"          1               1         letters          ja        verb
    "。"              1               1         punctuation      ja        punctuation
    "恋"              2               1         letters          ja        noun
    "の"              2               1         letters          ja        adposition
```

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たました。"];
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
    ___________    ______________    __________    _______    ________    ____________

    "マリー"             1               1         letters       ja       proper-noun
    "さん"               1               1         letters       ja       noun
    "は"                 1               1         letters       ja       adposition
    "ボストン"           1               1         letters       ja       proper-noun
    "から"               1               1         letters       ja       adposition
    "ニューヨーク"       1               1         letters       ja       proper-noun
    "に"                 1               1         letters       ja       adposition
    "引っ越し"           1               1         letters       ja       verb
```

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 = 11×1 string array
    "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 4-12

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
```

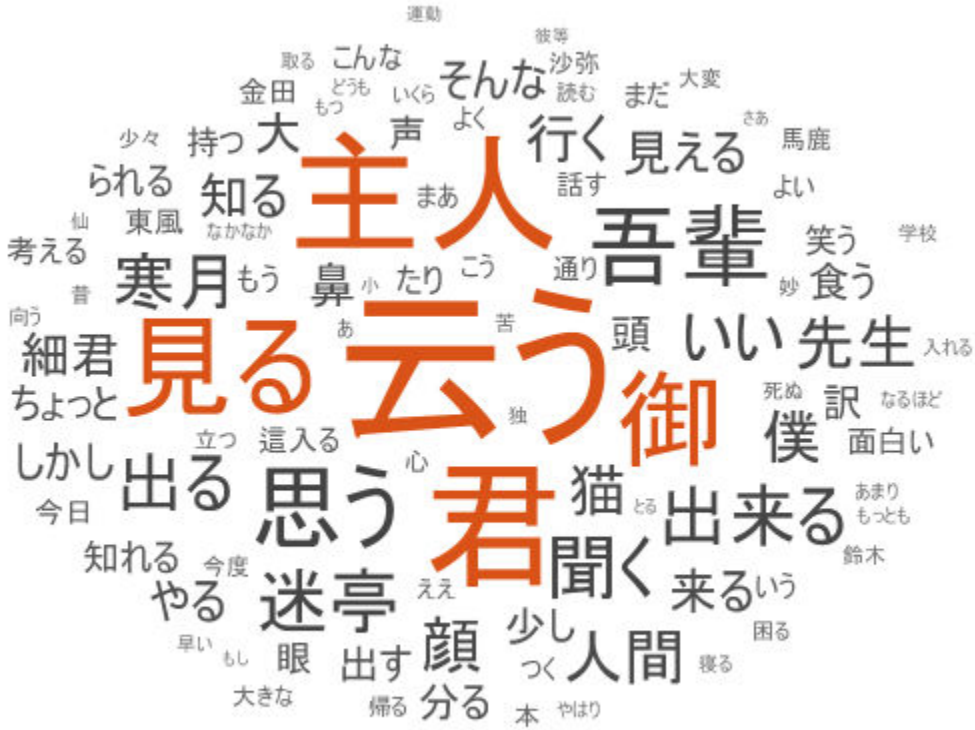
Remove the empty lines of text.

```
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
    " 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
    " 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
    " 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
    " 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き
```

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
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
 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
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
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 た
```

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
    ______     ______________    __________    _______     ________    ______________

    "吾輩"           1               1         letters        ja        pronoun
    "は"             1               1         letters        ja        adposition
    "猫"             1               1         letters        ja        noun
    "で"             1               1         letters        ja        auxiliary-verb
    "ある"           1               1         letters        ja        auxiliary-verb
    "夏目"           2               1         letters        ja        proper-noun
    "漱石"           2               1         letters        ja        proper-noun
    "ー"             4               1         letters        ja        numeral
```

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
    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
    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
   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
   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
```

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
    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
    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
   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
   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
```

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
    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
    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
   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
```

```
    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
```

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.

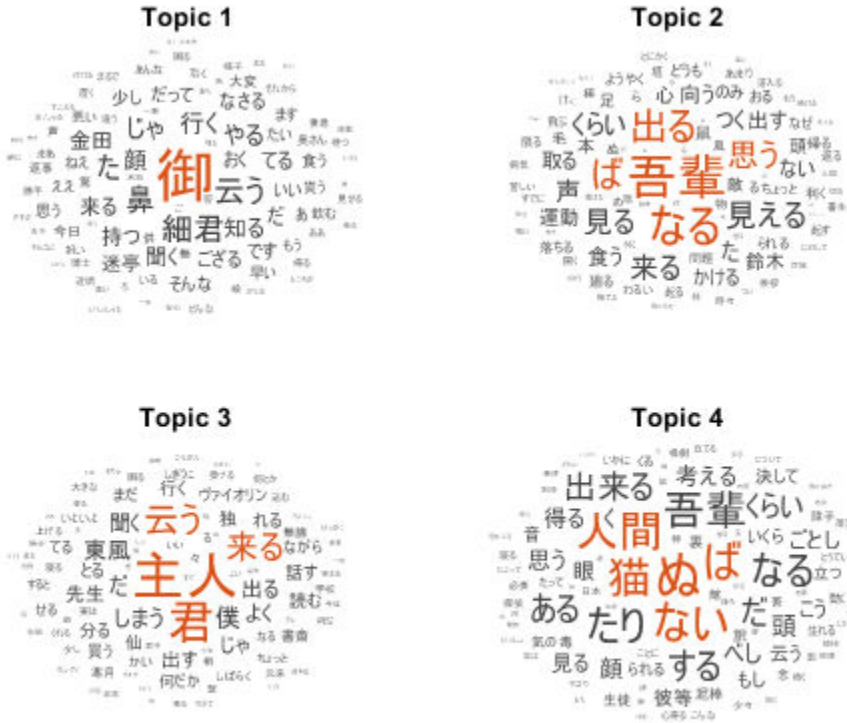
```
bag = removeEmptyDocuments(bag);
```

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
```



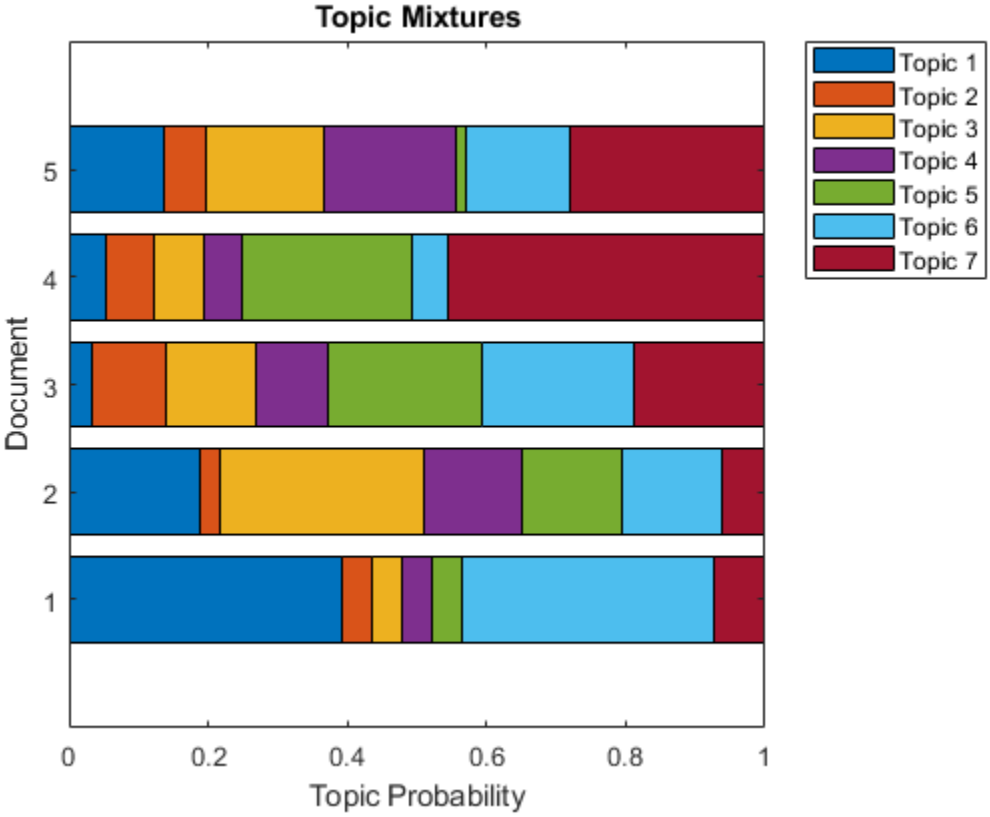
Visualize multiple topic mixtures using stacked bar charts. View five input documents at random and visualize the corresponding topic mixtures.

```
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
```

```
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 2-18
- “Analyze Text Data Using Multiword Phrases” on page 2-9
- “Analyze Text Data Containing Emojis” on page 2-35
- “Train a Sentiment Classifier” on page 2-47
- “Classify Text Data Using Deep Learning” on page 2-57
- “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 4-33.

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        Langu
    _________    ______________    ______________    __________    ___________    _____

    "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        Langua
    ________    ______________    ______________    __________    ___________    ______

    "Guten"           1                 1               1         letters          de
    "Morgen"          1                 1               1         letters          de
```

```
    "."             1               1               1          punctuation       de
    "Wie"           1               2               1          letters           de
    "geht"          1               2               1          letters           de
    "es"            1               2               1          letters           de
    "dir"           1               2               1          letters           de
    "?"             1               2               1          punctuation       de
```

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         Lan
    ___________    ______________    ______________    __________    ___________    ___

    "Ernst"             1                 1                1         letters
    "zog"               1                 1                1         letters
    "von"               1                 1                1         letters
    "Frankfurt"         1                 1                1         letters
    "nach"              1                 1                1         letters
    "Berlin"            1                 1                1         letters
    "."                 1                 1                1         punctuation
    "Besuchen"          2                 1                1         letters
```

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      Langua
    ____________    ______________    ______________    __________    _______    ______

    "Ernst"              1                 1                1         letters       de
    "Frankfurt"          1                 1                1         letters       de
    "Berlin"             1                 1                1         letters       de
    "Volkswagen"         2                 1                1         letters       de
    "Wolfsburg"          2                 1                1         letters       de
```

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 4-33

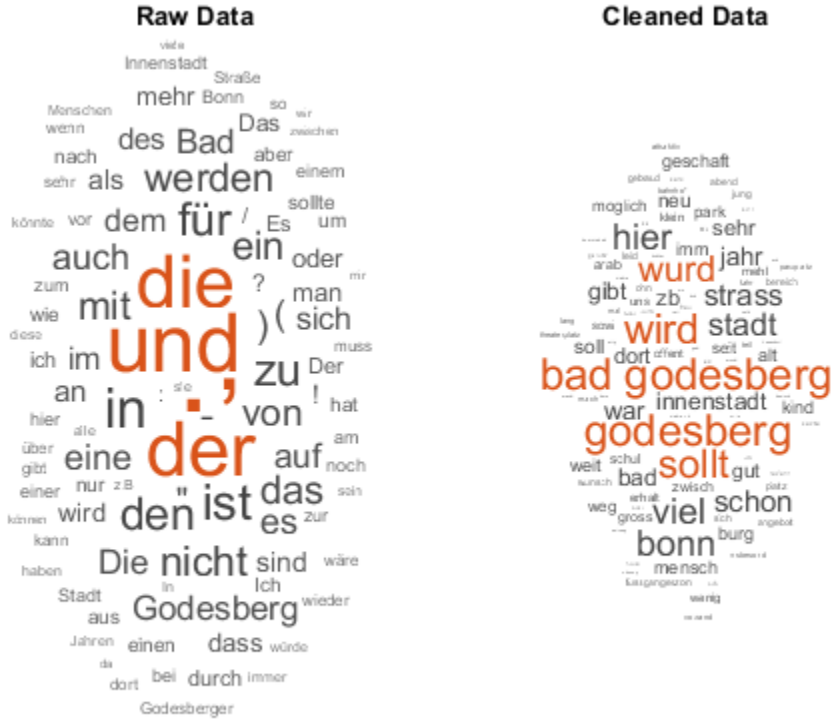
Analyze German Text Data

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. 662(
    "Bauleitplanverfahren zur Aufstellung des vorhabenbezogenen↵Bebauungsplans Nr. 652
    "Bauleitplanverfahren zur Aufstellung des Bebauungsplans↵Nr. 7621-56 ?Sebastianstra
    "EPICURO - European Partnership for Innovative Cities within and Urban Resilience (
    "Bauleitplanverfahren zur Aufstellung des Bebauungsplanes Nr. 6719-3 "Schwimmbad Wa
    "Bürgerbeteiligung an der Konzepterstellung für den Neubau eines Schwimmbades in Bo
    "Integriertes Handlungskonzept Grüne Infrastruktur (InHK GI) zur↵zukünftigen Freira
    "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-Brand
     46 tokens: Für den vorhabenbezogenen Bebauungsplan Nr . 6522-1 ? Didinkirica ? der
     41 tokens: Für das Gebiet zwischen Alfred-Bucherer-Straße , Sebastianstraße und de
    134 tokens: In den vergangenen Jahren führte der Klimawandel zu einer Vielzahl von
```

```
    24 tokens: Schaffung von Planungsrecht für den Bau eines neuen Familien - , Schul
    80 tokens: Für die begleitende Bürgerbeteiligung bei der Konzepterstellung für das
    60 tokens: In der Gebietskulisse des Grünen C sollen die Freiräume auch zukünftig
    51 tokens: Zur Entlastung von Pützchen / Bechlinghoven vor Durchgangsverkehr und
    29 tokens: Für das Areal der ehemaligen Landwirtschaftskammer sowie einer angrenze
    37 tokens: Für das Areal Herbert-Rabius-Straße im Stadtbezirk Beuel , Ortsteil Beu
```

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
    ___________________    ______________    ______________    __________    _________

    "Für"                         1                 1              1         letters
    "das"                         1                 1              1         letters
    "Gebiet"                      1                 1              1         letters
    "zwischen"                    1                 1              1         letters
    "Reuterstraße"                1                 1              1         letters
    ","                           1                 1              1         punctuati
    "Bundeskanzlerplatz"          1                 1              1         letters
    ","                           1                 1              1         punctuati
```

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-Brand
    46 tokens: Für den vorhabenbezogenen Bebauungsplan Nr . 6522-1 ? Didinkirica ? der
    41 tokens: Für das Gebiet zwischen Alfred-Bucherer-Straße , Sebastianstraße und de
   134 tokens: In den vergangenen Jahren führte der Klimawandel zu einer Vielzahl von
    24 tokens: Schaffung von Planungsrecht für den Bau eines neuen Familien - , Schul
    80 tokens: Für die begleitende Bürgerbeteiligung bei der Konzepterstellung für das
    60 tokens: In der Gebietskulisse des Grünen C sollen die Freiräume auch zukünftig
    51 tokens: Zur Entlastung von Pützchen / Bechlinghoven vor Durchgangsverkehr und
    29 tokens: Für das Areal der ehemaligen Landwirtschaftskammer sowie einer angrenze
    37 tokens: Für das Areal Herbert-Rabius-Straße im Stadtbezirk Beuel , Ortsteil Beu
```

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
    33 tokens: vorhabenbezogenen Bebauungsplan Nr . 6522-1 ? Didinkirica ? Bundesstadt
    27 tokens: Gebiet zwischen Alfred-Bucherer-Straße , Sebastianstraße Fußweg zwischen
    81 tokens: vergangenen Jahren führte Klimawandel Vielzahl Folgen Umwelt , Wirtschaf
    15 tokens: Schaffung Planungsrecht Bau neuen Familien - , Schul - Sportschwimmbades
    57 tokens: begleitende Bürgerbeteiligung Konzepterstellung neue Schwimmbad soll fol
    40 tokens: Gebietskulisse Grünen C sollen Freiräume zukünftig Sinne Naherholung , E
    32 tokens: Entlastung Pützchen / Bechlinghoven Durchgangsverkehr Verbindung geplant
    19 tokens: Areal ehemaligen Landwirtschaftskammer sowie angrenzenden städtischen Fl
    25 tokens: Areal Herbert-Rabius-Straße Stadtbezirk Beuel , Ortsteil Beuel-Mitte sol
```

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 , ec
    33 tokens: vorhabenbezog bebauungsplan nr . 6522-1 ? didinkirica ? bundesstadt bonn
```

```
    27 tokens: gebiet zwisch alfred-bucherer-strass , sebastianstrass fussweg zwisch ro
    81 tokens: vergang jahr fuhrt klimawandel vielzahl folg umwelt , wirtschaft mensch
    15 tokens: schaffung planungsrecht bau neu famili - , schul - sportschwimmbad flach
    57 tokens: begleit burgerbeteil konzepterstell neu schwimmbad soll folgend them bet
    40 tokens: gebietskuliss grun c soll freiraum zukunft sinn naherhol , landwirtschaf
    32 tokens: entlast putzch / bechlinghov durchgangsverkehr verbind geplant anschluss
    19 tokens: areal ehemal landwirtschaftskamm sowi angrenz stadtisch flach stadtbezir
    25 tokens: areal herbert-rabius-strass stadtbezirk beuel , ortsteil beuel-mitt soll
```

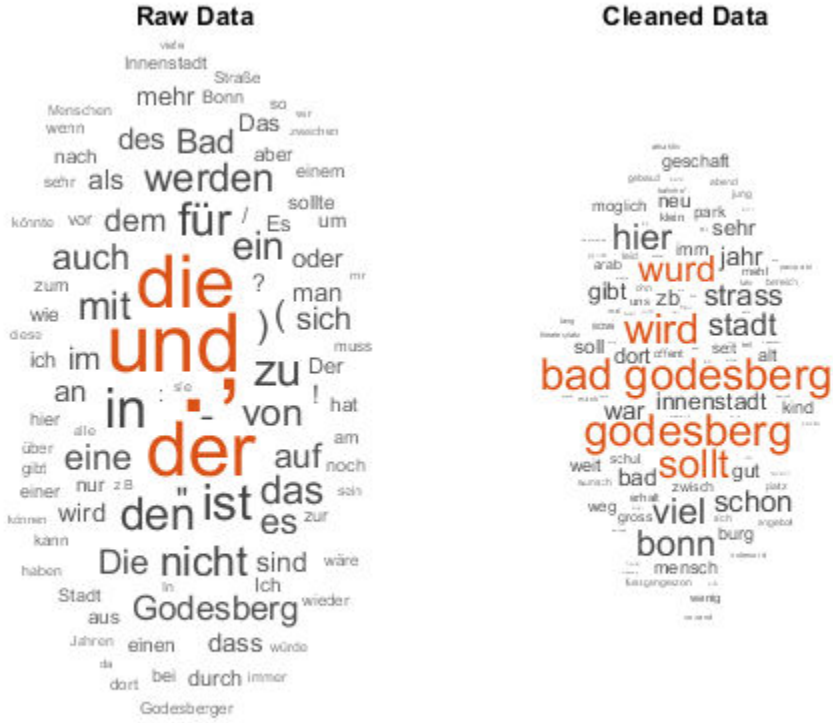
Erase the punctuation using the `erasePunctuation` function.

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

```
ans =
  10×1 tokenizedDocument:

    27 tokens: gebiet zwisch reuterstrass bundeskanzlerplatz willybrandtalle eduardpflu
    25 tokens: vorhabenbezog bebauungsplan nr 65221 didinkirica bundesstadt bonn stadtb
    22 tokens: gebiet zwisch alfredbuchererstrass sebastianstrass fussweg zwisch rockur
    64 tokens: vergang jahr fuhrt klimawandel vielzahl folg umwelt wirtschaft mensch st
    11 tokens: schaffung planungsrecht bau neu famili schul sportschwimmbad flach nordl
    41 tokens: begleit burgerbeteil konzepterstell neu schwimmbad soll folgend them bet
    31 tokens: gebietskuliss grun c soll freiraum zukunft sinn naherhol landwirtschaft
    27 tokens: entlast putzch bechlinghov durchgangsverkehr verbind geplant anschlussst
    18 tokens: areal ehemal landwirtschaftskamm sowi angrenz stadtisch flach stadtbezir
    19 tokens: areal herbertrabiusstrass stadtbezirk beuel ortsteil beuelmitt soll vorh
```

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 eduardpflu
    25 tokens: vorhabenbezog bebauungsplan nr 65221 didinkirica bundesstadt bonn stadtb
    22 tokens: gebiet zwisch alfredbuchererstrass sebastianstrass fussweg zwisch rockun
    64 tokens: vergang jahr fuhrt klimawandel vielzahl folg umwelt wirtschaft mensch st
    11 tokens: schaffung planungsrecht bau neu famili schul sportschwimmbad flach nordl
```

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.

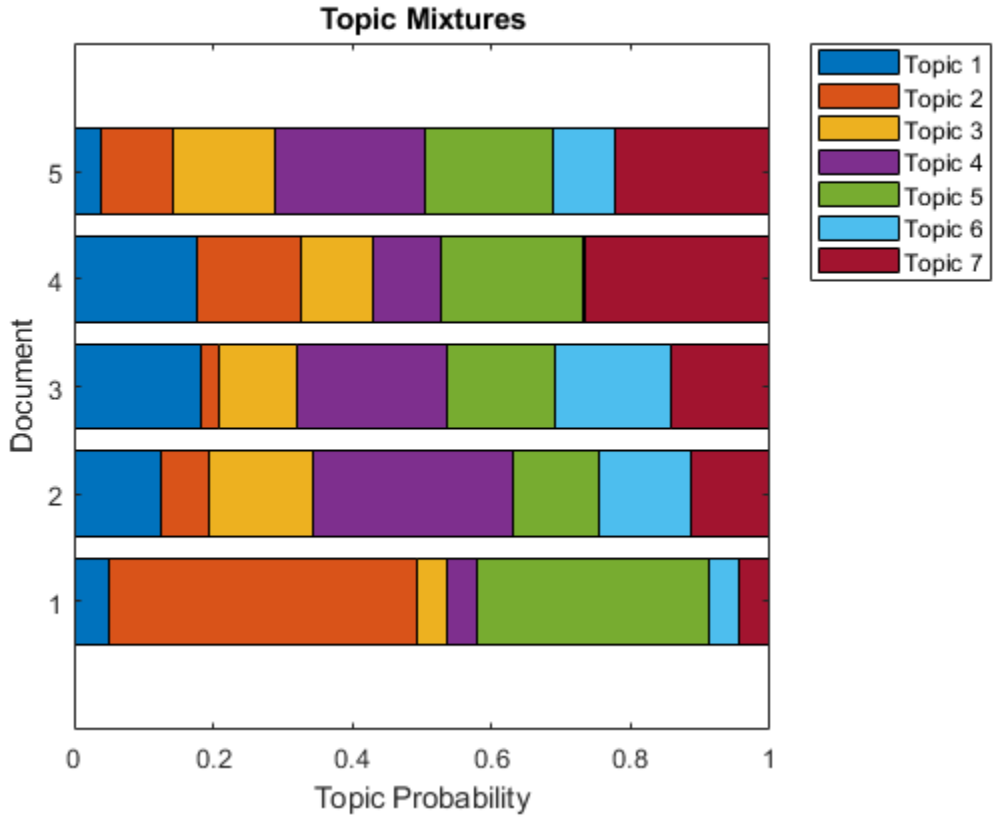
```
figure
for i = 1:4
    subplot(2,2,i)
    wordcloud(mdl,i);
    title("Topic " + i)
end
```

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 b
   116 tokens: sportplatz plittersdorf kommt leid regelmass unschon vorfall einsehbar
    64 tokens: mainz strass bereich geschaft kirch uberwieg beidseit zugeparkt unschon
    50 tokens: "1" "bezirksverodnet" "sollt" "kulturburgermeist" "gewahlt" "hatt" "auf
```

```
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 2-18
- “Analyze Text Data Using Multiword Phrases” on page 2-9
- “Analyze Text Data Containing Emojis” on page 2-35
- “Train a Sentiment Classifier” on page 2-47
- “Classify Text Data Using Deep Learning” on page 2-57
- “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 4-12
- “German Language Support” on page 4-26
- “Analyze German Text Data” on page 4-33

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`

Term	Definition	More Information
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`

Term	Definition	More Information
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	Definition	More Information
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`

Term	Definition	More Information
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`
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