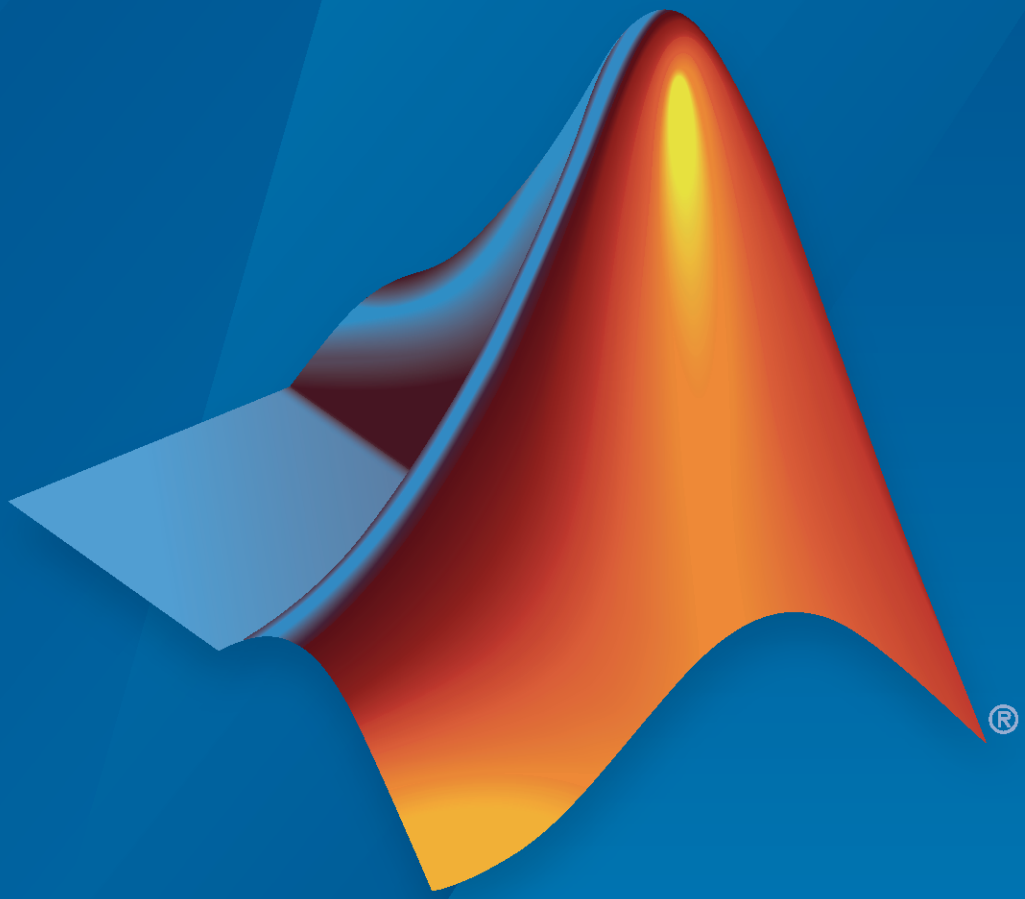
Text Analytics Toolbox™ Release Notes



MATLAB®

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™ Release Notes

Contents

R2023a

R2022b

R2022a

R2021b

R2021a

R2020b

R2020a

R2019b

R2019a

R2018b

R2018a

R2017b

R2023a

Version: 1.10

New Features

Bug Fixes

Text Preprocessing Live Editor Task: Preprocess text data in Live Editor

Interactively explore text preprocessing steps using the **Preprocess Text Data** Live Editor task. The task helps you explore and visualize different preprocessing steps for your workflows. You can clean up HTML, tokenize, add token details, change and remove words, and display results. For an example that explores different preprocessing options, see “Preprocess Text Data in Live Editor”.

The **Preprocess Text Data** Live Editor task automatically generates MATLAB® code for your live script. For an example that shows how to use the **Preprocess Text Data** task to create a preprocessing function that you can reuse in your workflows, see “Create Simple Preprocessing Function”.

PDF File Information: Extract file information from PDF files

Extract information from PDF files using the `pdfinfo` function. The function returns a structure containing information about:

- The number and sizes of the pages
- The PDF version
- The title, language, and keywords
- The author, creator, and producer
- The dates and times when the file was created and modified
- If the file is encrypted and allows text extraction
- The PDF filename

Text Preprocessing: Split text into paragraphs

Split strings and documents into paragraphs using the `splitParagraphs` function.

Named Entity Recognition: Train custom NER model

Train a custom model for named entity recognition (NER) using the `trainHMMEntityModel`. The function returns an `hmmEntityModel` object.

Add entity details to tokenized documents using the `addEntityDetails` function. To specify a custom NER model, use the `Model` name-value argument. To view the details of document tokens, use the `tokenDetails` function.

For an example that shows how to train a custom NER model, see “Train Custom Named Entity Recognition Model”.

R2022b

Version: 1.9

New Features

Bug Fixes

Grammatical Dependency Parsing: Visualize grammatical dependency parse trees

Visualize the grammatical dependency parse tree of a sentence using the `sentenceChart` function.

You can customize the resulting `DependencyChart` object by its modifying properties. For a list of properties, see `DependencyChart Properties`.

For string input, this feature requires Deep Learning Toolbox™ and the Text Analytics Toolbox Model *for UDify Data* support package. If this support package is not installed, then the function provides a download link.

This feature supports English, German, Japanese, and Korean text.

For an example showing how to visualize and extract information from a sentence, see Analyze Sentence Structure Using Grammatical Dependency Parsing.

Grammatical Dependency Parsing: Add grammatical dependency details to documents

To add grammatical dependency details to documents, use the `addDependencyDetails` function. To view the token details, use the `tokenDetails` function. The grammatical dependency details are in the `Head` and `Dependency` columns, where `Head` contains the indices of the head for each token, and `Dependency` contains the grammatical dependency categories.

This feature requires the Text Analytics Toolbox Model *for UDify Data* support package. If this support package is not installed, then the function provides a download link.

This feature supports English, German, Japanese, and Korean text.

For an example showing how to visualize and extract information from a sentence, see Analyze Sentence Structure Using Grammatical Dependency Parsing.

Keyword Extraction: Extract case-insensitive keywords and use case-sensitive delimiters using RAKE algorithm

Extract case-insensitive keywords using the rapid automatic keyword extraction (RAKE) algorithm by setting the `IgnoreKeywordCase` option of the `rakeKeywords` function to `1` (`true`). Use this option when you expect the same keywords to appear with variations in letter case and want to treat them as the same keyword, for example, the words "analytics", "Analytics", and "ANALYTICS".

Use case-sensitive delimiters for RAKE by setting the `IgnoreDelimiterCase` option of the `rakeKeywords` function to `0` (`false`). Use this option when you expect keywords and delimiters to differ only by case, for example, the delimiter "and" and the acronym "AND".

Document Similarity: Ignore case in BLEU score evaluation

Ignore case when you evaluate the bilingual evaluation understudy (BLEU) scores of a candidate document against a set of reference documents by setting the `IgnoreCase` option of the `bleuEvaluationScore` function to `true`.

Document Preprocessing: Check for substrings, words, and n-grams in documents

Check for substrings, words, and n-grams in `tokenizedDocument` objects using the `contains`, `containsWords`, and `containsNgrams` functions, respectively.

Document Preprocessing: Split document into sentences

Split a single `tokenizedDocument` object into sentences using the `splitSentences` function.

Unicode Representation: Convert text to more normalized forms

Strings that look identical can have different underlying representations. The Unicode composed normalized form (NFC), Unicode compatibility composed normalized form (NFKC), and Unicode compatibility decomposed normalized form (NFKD) ensure that equivalent strings have unique binary representations. Use these functions to help compare strings that contain accented characters, which can have several representations.

- To normalize text using NFC, use the `textanalytics.unicode.nfc` function.
- To normalize text using NFKC, use the `textanalytics.unicode.nfkc` function.
- To normalize text using NFKD, use the `textanalytics.unicode.nfkd` function.

Text Analytics Examples: Explore text analytics workflows

The Classify Documents Using Document Embeddings example shows how to train a document classifier by converting documents to feature vectors using word embeddings.

R2022a

Version: 1.8.1

Bug Fixes

Compatibility Considerations

Functionality Being Removed or Changed

FontSmoothing property of TextScatter objects will have no effect in a future release
Behavior change in future release

The `FontSmoothing` property of `TextScatter` objects will have no effect in a future release. Font smoothing will be enabled regardless of the value of the property.

tokenizedDocument does not split tokens containing digits and some special characters
Behavior change

Starting in R2022a, `tokenizedDocument` does not split some tokens where digits appear next to some special characters such as periods, hyphens, colons, slashes, and scientific notation. This behavior can produce better results when tokenizing text containing numbers, dates, and times.

In previous versions, `tokenizedDocument` might split at these characters. To reproduce the behavior, tokenize the text manually or insert whitespace characters around special characters before using `tokenizedDocument`.

R2021b

Version: 1.8

Bug Fixes

Topic Modeling: Explore Visualization Examples

New examples help you visualize data using topic models:

- Visualize LDA Topics Using Word Clouds
- Visualize LDA Topic Probabilities of Documents
- Visualize Document Clusters Using LDA Model
- Visualize LDA Topic Correlations
- Visualize Correlations Between LDA Topics and Document Labels

For more information about topic modeling, see Analyze Text Data Using Topic Models.

Deep Learning: Language Translation Example

For an example showing how to construct and train a neural network for language translation, see Language Translation Using Deep Learning.

R2021a

Unicode Representation: Convert text to Unicode canonical decomposition form (NFD)

Strings that look identical can have different underlying representations. The Unicode canonical decomposition form (NFD) ensures that equivalent strings have a unique binary representation. This is useful when comparing strings that contain accented characters which can have several ways to represent them.

To normalize text using NFD, use the `textanalytics.unicode.nfd` function.

Unicode Representation: Convert text to UTF-32 string representation

The 32-bit Unicode transformation format (UTF-32) is a fixed length Unicode code point encoding that uses exactly 32 bits per code point.

To convert string arrays to Unicode UTF-32 string representation, use the `textanalytics.unicode.UTF32` function. The `Data` property contains the UTF-32 code points, specified as a vector of integers with type `uint32`.

To get the Unicode character categories of the characters in a `UTF32` object, use the `characterCategories` function.

To convert the UTF-32 code points to hexadecimal values, use the `hex` function.

To convert the UTF-32 code points to string, use the `string` function.

Pretrained Models on GitHub: BERT and FinBERT transformer models

To learn how to load pretrained BERT and FinBERT transformer models into MATLAB, see the Transformer Models for MATLAB repository. You can use BERT and FinBERT for text classification, sentiment analysis, and other text analytics workflows.

To find the latest pretrained models and examples for deep learning, see MATLAB Deep Learning (GitHub).

Functionality Being Removed or Changed

extractFileText no longer supports extracting text from Microsoft Word 97–2003 binary DOC files

Errors

Support for extracting text from Microsoft® Word 97–2003 binary DOC files using the `extractFileText` function is no longer supported. Microsoft Word DOCX files will continue to be supported.

To extract text data from Microsoft Word 97–2003 binary DOC files, first save the file as a Microsoft Word DOCX, PDF, HTML, or plain text file, then use the `extractFileText` function.

htmlTree uses different algorithm for restructuring malformed HTML

Behavior change

When creating an `htmlTree` object, the software automatically restructures malformed input HTML code to have valid structure. This restructuring process includes adding, removing, and editing

elements as well as rearranging the tree structure. Starting in R2021a, the software uses an updated algorithm to restructure malformed HTML. This change can result in `htmlTree` objects created in R2021a or later having different size, structure, and content when compared to previous releases.

Starting in R2021a, when loading `htmlTree` objects from MAT files created in an R2020b or before, the software automatically restructures the `htmlTree` object using the same algorithm used for creating `htmlTree` objects. When loading `htmlTree` objects from MAT files created in R2021a or later, the software does not restructure the `htmlTree` object.

This table highlights some notable steps of the restructuring process:

Step	Change in Behavior
Automatically add head and title elements.	Starting in R2021a, when creating an `htmlTree` object from HTML code, the software automatically inserts missing `<HEAD>`, `<TITLE>`, and other elements. In previous versions, the `htmlTree` object only included these elements when they are present in the input code. When loading `htmlTree` objects from MAT files created in an earlier release, the software automatically inserts `<HEAD>` and `<TITLE>` elements. When loading `htmlTree` objects from MAT files created in R2021a or later, the software does not automatically insert these elements.
Automatically add missing elements.	Starting in R2021a, when creating an `htmlTree` object from HTML code, the software automatically inserts missing elements when parent and child elements are inconsistent. For example, when a `<li>` (list item) element does not have a parent `<ul>` (unordered list) or `<ol>` (unordered list) element, the software automatically adds a `<ul>` element to make the HTML valid. This can result in different outputs when compared with earlier releases. When loading `htmlTree` objects from MAT files created in an earlier release, the software automatically inserts missing elements. When loading `htmlTree` objects from MAT files created in R2021a or later, the software does not automatically insert missing elements.
Discard parts of malformed code.	When creating an `htmlTree` object with malformed HTML code, the software may discard parts of the text. For example, if the input code is the string `"<div>a</"`, then the software discards the text `"a</"`.

string function for htmlTree objects uses two spaces for indentation
Behavior change

The output of the `string` function for `htmlTree` objects is automatically indented for readability. Starting in R2021a, the function indents HTML code using two whitespace characters. In previous releases, the function indents HTML code with four spaces.

This change affects code that parses the HTML string directly. To parse and navigate HTML code, use `htmlTree` objects.

string function for htmlTree objects returns attributes in different order
Behavior change

When creating an `htmlTree` object, the software automatically parses the HTML element attributes of the input HTML code. Starting in R2021a, the software uses an updated algorithm to parse the HTML element attributes. This change can result in the `string` function returning HTML code with the attributes in a different order.

R2020b

Keyword Extraction: Extract keywords using RAKE and TextRank algorithms

Extract keywords using the `rakeKeywords` and `textrankKeywords` functions.

The Rapid Automatic Keyword Extraction (RAKE) algorithm extracts keywords using a delimiter-based approach to identify candidate keywords and scores them using word co-occurrences that appear in the candidate keywords. Keywords can contain multiple tokens. Furthermore, the RAKE algorithm also merges keywords when they appear multiple times, separated by the same merging delimiter. For an example, see Extract Keywords from Text Data Using RAKE.

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

Text Preprocessing: Preprocess text ignoring case

These functions now support case insensitive processing:

- `context`
- `removeWords`
- `replaceWords`
- `removeNgrams`
- `replaceNgrams`
- `word2vec`
- `word2ind`
- `isVocabularyWord`
- `topkwords`
- `topkngrams`
- `removeInfrequentWords`
- `removeInfrequentNgrams`

To use these functions ignoring case, set the `'IgnoreCase'` option to `true`.

The `removeStopWords` function, by default, removes stop words ignoring case. To remove stop words with case matching the stop word list given by the `stopWords` function, set the `'IgnoreCase'` option to `false`.

Sentiment Analysis: Generate sentiment lexicon for domain specific data

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

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

For an example showing how to generate a sentiment lexicon for finance data, see Generate Domain Specific Sentiment Lexicon.

Deep Learning: Use word embedding layers in custom training loops

`dlnetwork` (Deep Learning Toolbox) objects now support layer graphs containing `wordEmbeddingLayer` objects.

Use word embedding layers to convert text data to sequences of numeric vectors. For an example showing how to use a word embedding layer in a custom training loop, see Classify Text Data Using Custom Training Loop.

For a list of layers that `dlnetwork` (Deep Learning Toolbox) objects support, see the Supported Layers (Deep Learning Toolbox) section of the `dlnetwork` (Deep Learning Toolbox) reference page.

Text Preprocessing: Replace words and substrings matching patterns

Replace words and substrings that the match patterns in tokenized documents using the `replace` function.

Patterns are an intuitive alternative to regular expressions for matching patterns in text. To build complex patterns, you can combine pattern functions together in expressions and use them as inputs for text-searching functions. For example, to define a pattern for MATLAB release names, which start with "R", followed by the four-digit year, and then either "a" or "b":

```
pat = "R" + digitsPattern(4) + ("a"|"b");
```

For more information, see `pattern`.

Part-of-Speech Tagging: Specify abbreviations for sentence detection

When detecting part-of-speech details using the `addPartOfSpeechDetails` function, if the input documents do not contain sentence details, then the function first runs the `addSentenceDetails` function. The `addSentenceDetails` function uses abbreviation lists to help determine which periods terminate sentences.

To specify the abbreviations used for sentence detection in the `addPartOfSpeechDetails` function, use the `'Abbreviations'` option. To specify more options for sentence detection (for example, sentence starters), add sentence details using the `addSentenceDetails` function before using the `addPartOfSpeechDetails` function.

Functionality Being Removed or Changed

`extractFileText` will no longer support extracting text from Microsoft Word 97–2003 binary DOC files
Still runs

Support for extracting text from Microsoft Word 97–2003 binary DOC files using the `extractFileText` function will be removed in a future release. Microsoft Word DOCX files will continue to be supported.

To extract text data from Microsoft Word 97–2003 binary DOC files, first save the file as a PDF, Microsoft Word DOCX, HTML, or plain text file, then use the `extractFileText` function.

R2020a

Version: 1.5

New Features

Bug Fixes

Compatibility Considerations

Spelling Correction: Correct spelling of English, German, and Korean text

To correct spelling of tokenized documents, use the `correctSpelling` function. This function supports English, German, and Korean text. For an example showing how to correct the spelling of words in documents, see Correct Spelling in Documents.

You can customize the spelling correction by specifying a list of known words using the `'KnownWords'` option. Alternatively, you can specify custom Hunspell dictionary files using the `'ExtensionDictionary'`, `'Dictionary'`, and `'Affixes'` options. For an example showing how to create a custom Hunspell extension dictionary (also known as a personal dictionary), see Create Extension Dictionary for Spelling Correction.

Document Summarization: Extract summaries from text

Extract summaries from text data using the `extractSummary` function.

Document Similarity: Evaluate document similarity using BM25 algorithm and cosine similarity

To evaluate document similarity using the BM25 similarity scores or cosine similarity, use the `bm25Similarity` and `cosineSimilarity` functions, respectively.

Document Importance: Evaluate document importance using TextRank, LexRank, and Maximal Marginal Relevance (MMR)

Evaluate document importance with the TextRank and LexRank algorithms using the `textrankScores` and `lexrankScores` functions, respectively.

To compute similarities and importance scores, the `textrankScores` function uses the BM25 and PageRank algorithms, respectively. To compute similarities and importance scores, the `lexrankScores` function uses the cosine similarity function and the PageRank algorithm, respectively.

To evaluate document importance according to their relevance to a set of query documents avoiding redundancy, use the MMR algorithm. Evaluate the MMR scores using the `mmrScores` function. A document has a high MMR score if it is both relevant to the query and has minimal similarity relative to the other documents.

Document Similarity: Evaluate document similarity using BLEU and ROUGE scoring algorithms

The BiLingual Evaluation Understudy (BLEU) scoring algorithm evaluates the similarity between a candidate document and a collection of reference documents. Use the BLEU score to evaluate the quality of document translation and summarization models.

The Recall-Oriented Understudy for Gisting Evaluation (ROUGE) scoring algorithm evaluates the similarity between a candidate document and a collection of reference documents. Use the ROUGE score to evaluate the quality of document translation and summarization models.

To evaluate the BLEU and ROUGE scores of a candidate document against a set of reference documents, use the `bleuEvaluationScore` and `rougeEvaluationScore` functions, respectively.

Data Sets: Explore data sets for text analytics workflows

For a list of data sets that you can use to explore text analytics workflows, see Data Sets for Text Analytics.

Latent Dirichlet Allocation: Specify validation frequency

Specify how often to validate the LDA model during fitting using the `'ValidationFrequency'` option of the `fitlda` and `resume` functions.

Functionality Being Removed or Changed

`UIContextMenu` property of `TextScatter` is not recommended
Still runs

Starting in R2020a, using the `UIContextMenu` property of `TextScatter` to assign a context menu to a graphics object or UI component is not recommended. Use the `ContextMenu` property instead. The property values are the same.

There are no plans to remove support for the `UIContextMenu` property at this time. However, the `UIContextMenu` property no longer appears in the list returned by calling the `get` function on a graphics object or UI component.

R2019b

Korean Language Support: Perform text analytics on Korean language text including tokenization, lemmatization, part-of-speech tagging, and named entity recognition

Analyze Korean text using Text Analytics Toolbox. The following functions support Korean input:

- `tokenizedDocument`
- `normalizeWords`
- `removeStopWords`
- `addSentenceDetails`
- `addPartOfSpeechDetails`
- `addEntityDetails`
- `wordcloud`
- `wordCloudCounts`
- `splitSentences`
- `corpusLanguage`

For more information, see Korean Language Support.

Sentiment Analysis: Evaluate sentiment in text data using sentiment scoring algorithms including VADER

Analyze sentiment in text by evaluating the sentiment scores using the following scoring functions:

- `vaderSentimentScores` – Evaluate scores with the Valence Aware Dictionary and sEntiment Reasoner (VADER) algorithm.
- `ratioSentimentScores` – Evaluate scores with a ratio rule.

Specify custom lexicons which contain words and scores using the option.

When using `vaderSentimentScores`, you can further customize the VADER algorithm using the following options:

- Specify boosters (words like "very" and "really") using the `'Boosters'` option.
- Specify dampeners (words like "almost" and "somewhat") using the `'Dampeners'` option.
- Specify negations (words like "not" and "despite") using the `'Negations'` option.

Japanese and Korean Tokenization: Specify MeCab dictionary options

Customize Japanese and Korean tokenization by specifying options using the `mecabOptions` function. Specify the MeCab system and user models using the `Model` and `UserModel` options, respectively. Specify extractors for lemma, part-of-speech, and named entity details using the `LemmaExtractor`, `POSExtractor`, and `NERExtractor` options respectively.

To tokenize using the specified MeCab tokenization options, use the `'TokenizeMethod'` option of `tokenizedDocument`.

Deep Learning: Initialize word embedding layer with pretrained word embeddings

Initialize word embedding layer with pretrained word embeddings using the `Weights` option of `wordEmbeddingLayer` and create the corresponding word encoding directly from the word embedding vocabulary. For an example, see Create Word Encoding from Word Embedding.

Tokenization: Recompute sentence, part-of-speech, language, type, and named entity token details

The `addSentenceDetails`, `addPartOfSpeechDetails`, `addLanguageDetails`, `addTypeDetails`, and `addEntityDetails` functions, by default, do not recompute the details when contained in the documents. To recompute the corresponding token details, set the `'DiscardKnownValues'` option to `true`.

Deep Learning: Train Network For Language Translation using Attention

Train a deep learning network for language translation using a custom training loop. For an example showing how to train a network that translates decimals into roman numerals, see Sequence-to-Sequence Translation Using Attention.

Functionality Being Removed or Changed

`tokenizedDocument` detects Korean language
Behavior change

Starting in R2019b, `tokenizedDocument` detects the Korean language and sets the `'Language'` option to `'ko'`. This changes the default behavior of the `addSentenceDetails`, `addPartOfSpeechDetails`, `removeStopWords`, and `normalizeWords` functions for Korean document input. This change allows the software to use Korean-specific rules and word lists for analysis. If `tokenizedDocument` incorrectly detects text as Korean, then you can specify the language manually by setting the `'Language'` name-value pair of `tokenizedDocument`.

In previous versions, `tokenizedDocument` usually detects Korean text as English and sets the `'Language'` option to `'en'`. To reproduce this behavior, manually set the `'Language'` name-value pair of `tokenizedDocument` to `'en'`.

R2019a

German Language Support: Perform text analytics on German language text including tokenization, stop word removal, stemming, and part-of-speech tagging

Analyze German text using Text Analytics Toolbox. The functions `tokenizedDocument`, `addSentenceDetails`, `addPartOfSpeechDetails`, `addEntityDetails`, `removeStopWords`, and `normalizeWords` now support German input. For more information, see German Language Support.

For an example, see Analyze German Text Data.

Edit Distance: Find similarity between strings and documents using Levenshtein distance and other distance measures

Compute the edit distance between strings and documents using the `editDistance` function. This function computes the number of grapheme (human perceived character) or word insertions, deletions, swaps, and substitutions to transform one string or document to another.

Create edit distance searchers to perform nearest neighborhood search in a list of known strings, using edit distance. To create an edit distance searcher, use the `editDistanceSearcher` function. To use the edit distance searcher to find the nearest neighbors, or neighbors within a specified range, use the `knnsearch` and `rangesearch` functions respectively.

For an example showing how to use edit distance searchers for spelling correction, see Correct Spelling Using Edit Distance Searchers.

Named Entity Recognition: Detect locations, organizations, people's names, and other named entities in text

Detect named entities in English, Japanese, and German text using the `addEntityDetails` function. To get the entity tags from the documents, use the `tokenDetails` function.

For an example, see Add Named Entity Tags to Documents.

Tokenization and Preprocessing: Specify and detect patterns of custom tokens and replace words or phrases in tokenized documents

Detect custom tokens and custom token types by specifying a list of tokens or regular expressions using the `'CustomTokens'` and `'RegularExpressions'` options of `tokenizedDocument` respectively.

For an example, see Specify Custom Tokens.

Deep Learning Examples: Explore deep learning workflows (requires Deep Learning Toolbox)

Lean how to classify text data using convolutional neural network (CNN) or out-of-memory text data using transformed datastores.

- Classify Text Data Using Convolutional Neural Network

- Classify Out-of-Memory Text Data Using Deep Learning

Tokenization: Replace words and n-grams in documents

Find and replace words and n-grams in documents using the `replaceWords` and `replaceNgrams` functions respectively.

For an example, see Replace Words in Documents.

Multiword Phrases: Search documents for n-gram occurrences

Search documents for n-gram occurrences and view them in context using the `context` function.

For an example, see Search Documents for N-Gram Occurrences.

Functionality Being Removed or Changed

`tokenizedDocument` detects German language
Behavior change

Starting in R2019a, `tokenizedDocument` detects the German language and sets the `'Language'` option to `'de'`. This changes the default behavior of the `addSentenceDetails`, `addPartOfSpeechDetails`, `removeStopWords`, and `normalizeWords` functions for German document input. This change allows the software to use German-specific rules and word lists for analysis. If `tokenizedDocument` incorrectly detects English language text as German, then you can specify the English language manually by setting the `'Language'` name-value pair of `tokenizedDocument` to `'en'`.

In previous versions, `tokenizedDocument` usually detects German text as English and sets the `'Language'` option to `'en'`. To reproduce this behavior, manually set the `'Language'` name-value pair of `tokenizedDocument` to `'en'`.

R2018b

Japanese Language Support: Perform text analytics on Japanese language text, including tokenization, stop word removal, lemmatization, and part-of-speech tagging

Analyze Japanese text using Text Analytics Toolbox. The functions `tokenizedDocument`, `removeStopWords`, `normalizeWords`, and `addPartOfSpeechDetails` now support Japanese input. For more information, see Japanese Language Support.

For an example, see Analyze Japanese Text Data.

Word Normalization: Convert words to their dictionary form using lemmatization with parts of speech and other information

Lemmatize English and Japanese language text using the `'Style'` option of the `normalizeWords` function. For an example showing how to preprocess your text data using lemmatization and other techniques, see Prepare Text Data for Analysis.

Part-of-Speech Tagging: Identify parts of speech, such as adjectives, adverbs, nouns, and verbs

Add English and Japanese language part-of-speech tags to documents using the `addPartOfSpeechDetails` function. To get the part-of-speech tags from the documents, use the `tokenDetails` function.

Deep Learning: Train deep learning networks using word embedding layers (requires Deep Learning Toolbox)

For data to train deep learning networks, convert documents to sequences using the `doc2sequence` function and `wordEncoding` objects. Train word embeddings inside a deep learning network using word embedding layers. To create a word embedding layer, use the `wordEmbeddingLayer` function. For an example showing how to train a deep learning network for text classification using a word embedding layer, see Classify Text Data Using Deep Learning.

HTML Parsing: Extract HTML from specific parts of a web page using HTML structure and CSS classes

Parse HTML code using `htmlTree` objects. To find particular HTML elements using CSS selectors, use the `findElement` function. To get the attributes from HTML elements, use the `getAttribute` function.

Tokenization: Detect emoticons and emoji characters

Analyze text containing emoticons and emoji characters using `tokenizedDocument`. This function, by default, automatically detects emoticons and emoji characters and assigns the token types `'emoticon'` and `'emoji'`. To view the token types in of the tokens in documents, use the `tokenDetails` function. To learn more, see Analyze Text Data Containing Emojis.

Sentiment Analysis Example: Learn how to analyze sentiment in text

For an example showing how to train a classifier for sentiment analysis, using an annotated list of positive and negative sentiment words and a pretrained word embedding, see Train a Sentiment Classifier.

Deep Learning Examples: Learn about generating text and working with out-of-memory text data (requires Deep Learning Toolbox)

Use examples to learn about different applications of text analytics with deep learning. New examples include:

- Pride and Prejudice and MATLAB
- Word-By-Word Text Generation Using Deep Learning
- Classify Out-of-Memory Text Data Using Custom Mini-Batch Datastore

Functionality Being Removed or Changed

erasePunctuation skips complex tokens
Behavior change

Starting in R2018b, for `tokenizedDocument` input, `erasePunctuation`, by default, erases punctuation and symbol characters from tokens with type `'punctuation'` or `'other'` only. This prevents the function from affecting complex tokens such as URLs and email addresses.

In previous versions, `erasePunctuation` erases punctuation characters from all tokens. To reproduce this behavior, use the `'TokenTypes'` name-value pair in `erasePunctuation`.

normalizeWords skips complex tokens
Behavior change

Starting in R2018b, for `tokenizedDocument` input, `normalizeWords` normalizes tokens with type `'letters'` or `'other'` only. This prevents the function from affecting complex tokens such as URLs and email addresses.

In previous versions, `normalizeWords` normalizes all tokens. To reproduce this behavior, use the command `newDocuments = docfun(@(str) normalizeWords(str),documents)`.

tokenizedDocument does not split at slash and colon characters between digits
Behavior change

Starting in R2018b, if slash, backslash, or colon characters appear between two digits, then `tokenizedDocument` does not split at these characters. This behavior produces better results when tokenizing text containing dates and times.

In previous versions, `tokenizedDocument` splits at these characters. To reproduce this behavior, tokenize the text manually, or insert whitespace characters around slash, backslash, and colon characters before using `tokenizedDocument`.

tokenizedDocument does not split emoticons
Behavior change

Starting in R2018b, `tokenizedDocument`, by default, detects emoticon tokens. This behavior makes it easier to analyze text containing emoticons.

In R2017b and R2018a, `tokenizedDocument` splits emoticon tokens into multiple tokens. To reproduce this behavior, in `tokenizedDocument`, specify the `'DetectPatterns'` option to be `{'email-address','web-address','hashtag','at-mention'}`.

tokenDetails returns token type emoji for emoji characters
Behavior change

Starting in R2018b, `tokenizedDocument` detects emoji characters and the `tokenDetails` function reports these tokens with type `"emoji"`. This makes it easier to analyze text containing emoji characters.

In R2018a, `tokenDetails` reports emoji characters with type `"other"`. To find the indices of the tokens with type `"emoji"` or `"other"`, use the indices `idx = tdetails.Type == "emoji" | tdetails.Type == "other"`, where `tdetails` is a table of token details.

fitlda sorts topics
Behavior change

Starting in R2018b, `fitlda`, by default, sorts the topics in descending order of the topic probabilities of the input document set. This behavior makes it easier to find the topics with the highest probabilities.

In previous versions, `fitlda` does not change the topic order. To reproduce the behavior, set the `'TopicOrder'` option to `'unordered'`.

ismember will be removed
Warns

To update your code, for `wordEmbedding` object input, change the function name from `ismember` to `isVocabularyWord`. You do not need to change the arguments. The syntaxes are equivalent.

R2018a

Multiword Phrases: Extract and count multiword phrases (n-grams) from tokenized text

You can extract and count multiword phrases (n-grams) from tokenized text using `bagOfNgrams` objects. For an example showing how to analyze text using n-grams, see Analyze Text Data Using Multiword Phrases.

HTML Text: Extract text content from HTML pages.

Extract text directly from HTML code in a string using `extractHTMLText`. To extract text content from HTML files, use `extractFileText`.

Deep Learning: Learn how to use deep learning LSTM networks for text classification (requires Neural Network Toolbox)

An LSTM network is a type of deep learning network that can learn long-term dependencies between time steps of sequence data. By treating text data as sequences of words, you can use deep learning techniques with your text data.

To learn how to use deep learning long short-term memory (LSTM) networks for text classification, see Classify Text Data Using Deep Learning.

Pattern Detection: Detect sentences, email addresses, and URLs in text

You can detect complex tokens such as email addresses, web addresses, hashtags, and at-mentions using the `'DetectPatterns'` option in `tokenizedDocument`. Use `splitSentences` to split text into sentences, and `addSentenceDetails` to add sentence information to tokenized documents. To get information about the tokens in a `tokenizedDocument` array, use `tokenDetails`.

Stochastic LDA Model Training: Fit LDA models to large datasets

Fit latent Dirichlet allocation (LDA) models to large datasets using stochastic approximate variational Bayes (SAVB) by specifying the `'Solver'` name-value pair to be `'savb'` in `fitlda`. This solver is best suited for large datasets and can fit a good model in fewer passes through the data. For an example showing how to compare LDA solvers, see Compare LDA Solvers.

Pretrained Word Embedding: Download pretrained fastText word embedding

You can download a pretrained fastText 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, the function provides a download link.

Word Frequency Counting: Count words and n-grams in parallel (requires Parallel Computing Toolbox)

You can create multiple bag-of-words or bag-of-n-grams models in parallel and combine them using `join`. For an example showing how to create a bag-of-words model in parallel, see Create Bag-of-Words Model in Parallel.

Functionality Being Removed or Changed

Functionality	Result	Use Instead	Compatibility Considerations
`tokenizedDocument`	Still runs	Not applicable	In R2018a, `tokenizedDocument`, by default, detects complex tokens (email addresses, web addresses, hashtags, and at-mentions). In R2017b, `tokenizedDocument` splits complex tokens into multiple tokens. To reproduce this behavior, in `tokenizedDocument`, specify the `'DetectPatterns'` option to be `'none'`.

R2017b

Version: 1.0

New Features

Text Preprocessing: Prepare text for analysis by automatically extracting and preprocessing words from raw text

You can perform the following common character level preprocessing steps to prepare text data before splitting it into words:

- Erase HTML and XML tags using `eraseTags`.
- Erase URLs using `eraseURLs`.
- Erase punctuation using `erasePunctuation`.
- Convert HTML and XML entities into characters using `decodeHTMLEntities`.

After character level preprocessing, you can split text into words using `tokenizedDocument` which creates an array of `tokenizedDocument` objects. With a `tokenizedDocument` array, you can perform the following word level preprocessing steps:

- Remove specified words from an array of documents using `removeWords`.
- Remove a common list of stop words which are not useful for analysis (such as "a" and "the") using `removeWords` and `stopWords`.
- Remove long and short words using `removeLongWords` and `removeShortWords` respectively.
- Stem words using `normalizeWords`.

For an example showing how to preprocess text data and prepare for it for analysis, see Prepare Text Data for Analysis.

Machine Learning Algorithms: Discover topics and clusters of documents using Latent Dirichlet Allocation (LDA) and Latent Semantic Analysis (LSA)

You can analyze text data using the Latent Dirichlet Allocation topic model. Latent Dirichlet Allocation models a collection of documents as mixtures of topics.

Fit an `ldaModel` using `fitlda`. You can resume training using `resume`. Using `ldaModel` objects, you can perform the following tasks:

- Visualize topics and word importance of an LDA model using `wordcloud` and `topkwords`.
- Extract features, or reduce dimensionality using `transform`. This function transforms documents into the lower dimensional topic probability space.
- Predict top topics of documents using `predict`.
- Calculate document log probabilities and detect outliers using `logp`.

You can also use Latent Semantic Analysis to model your text data.

Fit an `lsaModel` using `fitlsa`. To use an LSA model as a feature extractor, or a dimension reducing tool, use `transform`. This function transforms documents into a lower dimensional semantic space.

For an example showing how to use LDA to analyze text data, see Analyze Text Data Using Topic Models. For more information on LSA models, see `lsaModel`.

Word Embeddings: Convert words to numeric vectors using word2vec, FastText, and GloVe word embedding models

Use word embeddings to discover relationships between words. Word embeddings model words as vectors in a fixed dimensional space. For example, a word embedding may learn the relationship "king" – "man" + "woman" = "queen".

Create a `WordEmbedding` object by using one of the following methods:

- Import word embedding files from word2vec, FastText, and GloVe using `readWordEmbedding`.
- Train your own word embeddings from text data using `trainWordEmbedding`.

With a `WordEmbedding` object, you can do the following:

- Map words to vectors and back using `word2vec` and `vec2word`.
- Write the word embedding to a file using `writeWordEmbedding`.

For an example showing how to explore word embeddings, see Visualize Word Embedding Using Text Scatter Plots.

Text Plots: Visualize text data using word clouds and text scatter plots

Text Analytics Toolbox extends the functionality of the `wordcloud` (MATLAB) function. It adds support for the following tasks:

- Create word clouds directly from string. `wordcloud` automatically tokenizes, preprocesses, and counts word frequencies of string input.
- Create word clouds from bag-of-words models.
- Create word clouds from LDA topics.

You can also visualize text data using 2-D and 3-D text scatter plots. Use `textscatter` and `textscatter3` to plot words at specified coordinates of 2-D and 3-D scatter plots respectively.

For an example showing how to visualize collections of text data using word clouds, see Visualize Text Data Using Word Clouds.

Document Import: Read text from PDF and Microsoft Word files

You can extract text data directly from plain text, PDF, and Microsoft Word files using `extractFileText`.

For an example showing how to extract text data from files and import it into MATLAB, see Extract Text Data From Files.

Text Statistics: Calculate word frequency and TF-IDF matrices from document collections

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 `bagOfWords` object using `bagOfWords`.

With a `bagOfWords` object, you can perform the following tasks:

- Encode documents as a matrix of word counts using `encode`.
- View the most frequent words using `topkwords`.
- Add and remove documents using `addDocument` and `removeDocument` respectively.
- Remove empty documents using `removeEmptyDocuments`.
- Remove infrequent words using `removeInfrequentWords`.

You can input `bagOfWords` objects directly into `fitlda`, `fitlsa`, and `wordcloud`.

You can create tf-idf matrices from a bag-of-words model using `tfidf`. A tf-idf matrix is a statistic that captures word importance in a collection of documents. It captures the number of times each word appear in a collection, and how many documents each word appears in.

For more information, see `bagOfWords`.

Word Normalization: Convert words to their word roots using the Porter stemming algorithm

To group different forms of English words by reducing them to a common stem, use `normalizeWords`. For example, use this function to reduce the words "walk", "walks", "walking" and "walk" all to their word root "walk". `normalizeWords` uses the Porter stemmer.

For more information, see `normalizeWords`.