Chapter 6: Topic Modeling

In this chapter, we will cover topic modeling, or the unsupervised discovery of topics present in a corpus of text. There are many different algorithms available to do this, and we will cover four of them: **Latent Dirichlet Allocation** (**LDA**) using two different packages, **non-negative matrix factorization**, K-means with **Bidirectional Encoder Representations from Transformers** (**BERT**) embeddings, and **Gibbs Sampling Dirichlet Multinomial Mixture** (**GSDMM**) for topic modeling of short texts, such as sentences or tweets.

The recipe list is as follows:

- LDA topic modeling with sklearn
- LDA topic modeling with gensim
- NMF topic modeling
- K-means topic modeling with BERT
- Topic modeling of short texts

Technical requirements

In this chapter, we will work with the same BBC dataset that we worked with in ***Chapter 4***, *Classifying Texts*. The dataset is available at **https://github.com/PacktPublishing/Python-Natural-Language-Processing-Cookbook/blob/master/Chapter04/bbc-text.csv** in the book's GitHub repository.

LDA topic modeling with sklearn

In this recipe, we will use the **LDA** algorithm to discover topics that appear in the BBC dataset. This algorithm can be thought of as dimensionality reduction, or going from a representation where words are counted (such as how we represent documents using **`CountVectorizer`** or **`TfidfVectorizer`**, see ***Chapter 3***, *Representing Text: Capturing Semantics*, we instead represent documents as sets of topics, each topic

with a weight. The number of topics is of course much smaller than the number of words in the vocabulary. To learn more about how the LDA algorithm works, see **https://highdemandskills.com/topic-modeling-intuitive/**.

Getting ready

We will use the **sklearn** and **pandas** packages. If you haven't installed them, do so using the following command:

```
pip install sklearn
pip install pandas
```

How to do it...

We will use a dataframe to parse in the data, then represent the documents using the **CountVectorizer** object, apply the LDA algorithm, and finally print out the topics' most common words. The steps for this recipe are as follows:

1. Perform the necessary imports:
```
import re
import pandas as pd
from sklearn.feature_extraction.text import
CountVectorizer
from sklearn.decomposition import
LatentDirichletAllocation as LDA
from Chapter04.preprocess_bbc_dataset import
get_stopwords
from Chapter04.unsupervised_text_classification
import tokenize_and_stem
```
2. Initialize the global variables:
```
stopwords_file_path = "Chapter01/stopwords.csv"
stopwords = get_stopwords(stopwords_file_path)
bbc_dataset = "Chapter04/bbc-text.csv"
```
3. We then use a function to create the vectorizer:
```
def create_count_vectorizer(documents):
    count_vectorizer = \
    CountVectorizer(stop_words=stopwords,
                    tokenizer=tokenize_and_stem)
    data =
count_vectorizer.fit_transform(documents)
```

```
    return (count_vectorizer, data)
```

4. The **clean_data** function will remove punctuation and digits:

```
def clean_data(df):
    df['text'] = \
    df['text'].apply(lambda x: re.sub(r'[^\w\s]',
                                      ' ', x))
    df['text'] = \
    df['text'].apply(lambda x: re.sub(r'\d', '',
x))
    return df
```

5. The following function will create an LDA model and fit it to the data:

```
def create_and_fit_lda(data, num_topics):
    lda = LDA(n_components=num_topics, n_jobs=-1)
    lda.fit(data)
    return lda
```

6. The **get_most_common_words_for_topics** function will get the most common words for each topic in a dictionary:

```
def get_most_common_words_for_topics(model,
vectorizer,
                                     n_top_words):
    words = vectorizer.get_feature_names()
    word_dict = {}
    for topic_index, topic in \
    enumerate(model.components_):
        this_topic_words = [words[i] for i in \
                topic.argsort()[:-n_top_words -
1:-1]]
        word_dict[topic_index] = this_topic_words
    return word_dict
```

7. The **print_topic_words** function will print the most common words for each topic:

```
def print_topic_words(word_dict):
    for key in word_dict.keys():
        print(f"Topic {key}")
        print("\t", word_dict[key])
```

8. Now we can read the data and clean it. The documents for processing will be in the **text** column of the dataframe:

```
df = pd.read_csv(bbc_dataset)
df = clean_data(df)
documents = df['text']
```

9. We then set the number of topics to **5**:

9. We then set the number of topics to 5:

```
number_topics = 5
```

10. We can now create the vectorizer, transform the data, and fit the LDA model:

```
(vectorizer, data) =
create_count_vectorizer(documents)
lda = create_and_fit_lda(data, number_topics)
```

11. Now, we create a dictionary with the most common words and print it:

```
topic_words = \
get_most_common_words_for_topics(lda, vectorizer,
10)
print_topic_words(topic_words)
```

The results will vary each time you run it, but one possible output might be as follows:

```
Topic 0
         ['film', 'best', 'award', 'year', 'm',
'star',
          'director', 'actor', 'nomin', 'includ']
Topic 1
         ['govern', 'say', 'elect', 'peopl',
'labour',
          'parti', 'minist', 'plan', 'blair',
'tax']
Topic 2
         ['year', 'bn', 'compani', 'market', 'm',
'firm',
          'bank', 'price', 'sale', 'share']
Topic 3
         ['use', 'peopl', 'game', 'music', 'year',
'new',
          'mobil', 'technolog', 'phone', 'show']
Topic 4
         ['game', 'year', 'play', 'm', 'win',
'time',
          'england', 'first', 'player', 'back']
```

How it works...

In *step 1*, we import the necessary packages. We use the **`CountVectorizer`** object for the vectorizer and the **`LatentDirichletAllocation`** object for the topic model. In *step 2*, we

initialize the path to the stopwords file, and read it into a list, and then initialize the path to the text dataset. In *step 3*, we define the function that creates the count vectorizer and fits it to the data. We then return both the transformed data matrix and the vectorizer itself, which we will use in later steps.

The **clean_data** function in *step 4* removes characters that are not word characters and not spaces (mostly punctuation) as well as digits from the dataset. For this we use the **apply** function on the pandas **Dataframe** object that applies a lambda function that uses the **re** package.

In *step 5*, we define the **create_and_fit_lda** function that creates an LDA model. It takes the vector-transformed data and the number of topics as arguments. The **n_jobs** argument passed in to the **LDA** object tells it to use all processors for parallel processing. In this case, we know the number of topics in advance, and we can pass that number to the function. In cases where we have unlabeled data, we don't know that number.

In *step 6*, we define the **get_most_common_words_for_topics** function that gets the most frequent words for each topic. It returns a dictionary indexed by the number of the topic.

The **print_topic_words** function in *step 7* will print out the frequent word dictionary.

In *step 8*, we read in the data from the CSV file to a dataframe, remove any unnecessary characters, and then get the documents from the **text** column of the dataframe. In *step 9*, we set the number of topics to *5*. In *step 10*, we create the vectorizer and the LDA model.

In *step 11*, we create the most frequent words dictionary and print it. The results correspond well to the predefined topics. **Topic 0** relates to entertainment, **topic 1** is about politics, **topic 2** concerns the economy, **topic 3** relates to technology, and **topic 4** concerns sports.

There's more...

Let's now save the model and test it on a new example:

1. Import the **pickle** package:
```
import pickle
```
2. Initialize the model and vectorizer paths:

```
model_path = "Chapter06/lda_sklearn.pkl"
vectorizer_path = "Chapter06/vectorizer.pkl"
```

3. Initialize the new example:

```
new_example = """Manchester United players slumped
to the turf at full-time in Germany on Tuesday in
acknowledgement of what their latest pedestrian
first-half display had cost them. The 3-2 loss at
RB Leipzig means United will not be one of the 16
teams in the draw for the knockout stages of the
Champions League. And this is not the only price
for failure. The damage will be felt in the
accounts, in the dealings they have with current
and potentially future players and in the faith the
fans have placed in manager Ole Gunnar Solskjaer.
With Paul Pogba's agent angling for a move for his
client and ex-United defender Phil Neville speaking
of a "witchhunt" against his former team-mate
Solskjaer, BBC Sport looks at the ramifications and
reaction to a big loss for United."""
```

4. Define the **`save_model`** function:

```
def save_model(lda, lda_path, vect, vect_path):
pickle.dump(lda, open(lda_path, 'wb'))
pickle.dump(vect, open(vect_path, 'wb'))
```

5. The **`test_new_example`** function applies the LDA model to the new input:

```
def test_new_example(lda, vect, example):
    vectorized = vect.transform([example])
    topic = lda.transform(vectorized)
    print(topic)
    return topic
```

6. Let's now run the function:

```
test_new_example(lda, vectorizer, new_example)
```

   The result will be as follows:

```
[[0.00509135 0.00508041 0.00508084 0.27087506
0.71387233]]
```

The result is an array of probabilities, one for each topic. The largest probability is for the last topic, which is sport, and the correct identification.

LDA topic modeling with gensim

LDA topic modeling with gensim

In the previous section, we saw how to create an LDA model with the **sklearn** package. In this recipe, we will create an LDA model using the **gensim** package.

Getting ready

We will be using the **gensim** package, which can be installed using the following command:

```
pip install gensim
```

How to do it...

We will load the data, clean it, preprocess it in a similar fashion to the previous recipe, and then create the LDA model. The steps for this recipe are as follows:

1. Perform the necessary imports:

```
import re
import pandas as pd
from gensim.models.ldamodel import LdaModel
import gensim.corpora as corpora
from gensim.utils import simple_preprocess
import matplotlib.pyplot as plt
from pprint import pprint
from Chapter06.lda_topic import stopwords,
bbc_dataset, clean_data
```

2. Define the function that will preprocess the data. It uses the **clean_data** function from the previous recipe:

```
def preprocess(df):
    df = clean_data(df)
    df['text'] = \
    df['text'].apply(lambda x: \
                     simple_preprocess(x,
deacc=True))
    df['text'] = \
    df['text'].apply(lambda x: [word for word in x
if \
                                word not in
stopwords])
```

```
    return df
```

3. The **`create_lda_model`** function creates and returns the model:

```
def create_lda_model(id_dict, corpus, num_topics):
    lda_model = LdaModel(corpus=corpus,
                         id2word=id_dict,
                         num_topics=num_topics,
                         random_state=100,
                         chunksize=100,
                         passes=10)
    return lda_model
```

4. Read and preprocess the BBC dataset:

```
df = pd.read_csv(bbc_dataset)
df = preprocess(df)
```

5. Create the **`Dictionary`** object and the corpus:

```
texts = df['text'].values
id_dict = corpora.Dictionary(texts)
corpus = [id_dict.doc2bow(text) for text in texts]
```

6. Set the number of topics to be **5** and create the LDA model:

```
number_topics = 5
lda_model = create_lda_model(id_dict, corpus,
number_topics)
```

7. Print the topics:

```
pprint(lda_model.print_topics())
```

The results will vary and may appear as follows:

```
[(0,
  '0.010*"net" + 0.008*"software" + 0.007*"users" +
0.007*"information" + '
  '0.007*"people" + 0.006*"attacks" +
0.006*"computer" + 0.006*"data" + '
  '0.006*"use" + 0.005*"firms"'),
(1,
  '0.012*"people" + 0.006*"blair" + 0.005*"labour"
+ 0.005*"new" + '
  '0.005*"mobile" + 0.005*"party" + 0.004*"get" +
0.004*"government" + '
  '0.004*"uk" + 0.004*"election"'),
(2,
  '0.012*"film" + 0.009*"best" + 0.006*"music" +
0.006*"year" + 0.005*"show" + '
  '0.005*"new" + 0.004*"uk" + 0.004*"awards" +
0.004*"films" + 0.004*"last"'),
```

```
(3,
  '0.008*"game" + 0.006*"england" + 0.006*"first" +
0.006*"time" + '
  '0.006*"year" + 0.005*"players" + 0.005*"win" +
0.005*"world" + 0.005*"back" '
  '+ 0.005*"last"'),
(4,
  '0.010*"bn" + 0.010*"year" + 0.007*"sales" +
0.005*"last" + '
  '0.004*"government" + 0.004*"new" +
0.004*"market" + 0.004*"growth" + '
  '0.004*"spending" + 0.004*"economic"')]
```

How it works...

In *step 1*, we import the necessary functions and variables. In *step 2*, we define the function that preprocesses the data. In this function, we use the **`clean_data`** function from the previous recipe that removes punctuation and digits from the texts. We then use the **`gensim simple_preprocess`** function, which puts the input into lowercase and tokenizes it. We then remove stopwords from the input.

In *step 3*, we create the LDA model. The inputs to the model are as follows: the corpus, or the transformed texts, the **`Dictionary`** object, which is analogous to a vectorizer, the number of topics, the random state, which, if set, ensures model reproducibility, chunk size, or the number of documents that are used in each training chunk, and the passes – the number of times the corpus is passed during training. The more passes that are done through the corpus, the better the model will be.

In *step 4*, we define the function that plots log perplexity against the number of topics in the model. This function creates several models, one each for **2** through **9** topics. It then calculates the log perplexity for each and graphs it.

In *step 5*, we read and preprocess the BBC dataset using the imported and predefined functions. In *step 6*, we create **`id_dict`**, a **`Dictionary`** object that is analogous to a vectorizer, and then use it to map the input texts to *bags of words* according to their mappings in the **`id_dict`** object.

In *step 7*, we create the model using five topics, the **`id`** dictionary and the

corpus. When we print the topics in *step 8*, we see that the topics make sense and correspond roughly as follows: **0** to **tech**, **1** to **politics**, **2** to **entertainment**, **3** to **sports**, and **4** to **business**.

There's more...

Now let's save the model and apply it to novel input:

1. Define the new example:

```
new_example = """Manchester United players slumped
to the turf
at full-time in Germany on Tuesday in
acknowledgement of what their
latest pedestrian first-half display had cost them.
The 3-2 loss at
RB Leipzig means United will not be one of the 16
teams in the draw
for the knockout stages of the Champions League.
And this is not the
only price for failure. The damage will be felt in
the accounts, in
the dealings they have with current and potentially
future players
and in the faith the fans have placed in manager
Ole Gunnar Solskjaer.
With Paul Pogba's agent angling for a move for his
client and ex-United
defender Phil Neville speaking of a "witchhunt"
against his former team-mate
Solskjaer, BBC Sport looks at the ramifications and
reaction to a big loss for United."""
```

2. Define the function that will save the model and the **Dictionary** object:

```
def save_model(lda, lda_path, id_dict, dict_path):
    lda.save(lda_path)
    id_dict.save(dict_path)
```

3. The **load_model** function loads the model and the **Dictionary** object:

```
def load_model(lda_path, dict_path):
    lda = LdaModel.load(lda_path)
    id_dict = corpora.Dictionary.load(dict_path)
    return (lda, id_dict)
```

4. The **test_new_example** function preprocesses the input and predicts the topic using the LDA model:

```
def test_new_example(lda, id_dict, input_string):
    input_list = clean_text(input_string)
    bow = id_dict.doc2bow(input_list)
    topics = lda[bow]
    print(topics)
    return topics
```

5. Save our model and **Dictionary** object:

```
save_model(lda_model, model_path, id_dict,
dict_path)
```

6. Let's now use the trained model to make a prediction on the new example:

```
test_new_example(lda_model, id_dict, new_example)
```

   The result will appear as follows:

```
[(0, 0.023436226), (1, 0.036407135), (3, 0.758486),
(4, 0.17845567)]
```

The prediction is a list of tuples, where the first element in each tuple is the number of the topic and the second element is the probability that this text belongs to this particular topic. In this example, we see that the third topic is the most probable, which is sport, and is the correct identification.

NMF topic modeling

In this recipe, we will use another unsupervised topic modeling technique, **NMF**. We will also explore another evaluation technique, topic model coherence. NMF topic modeling is very fast and memory efficient and works best with sparse corpora.

Getting ready

We will continue using the **gensim** package in this recipe.

How to do it...

We will create an NMF topic model and evaluate it using the coherence measure, which measures human topic interpretability. Many of the functions used for NMF models are the same as for LDA models in the **gensim** package. The steps for this recipe are as follows:

package. The steps for this recipe are as follows:

1. Perform the necessary imports:

```
import re
import pandas as pd
from gensim.models.nmf import Nmf
from gensim.models import CoherenceModel
import gensim.corpora as corpora
from gensim.utils import simple_preprocess
import matplotlib.pyplot as plt
from pprint import pprint
from Chapter06.lda_topic_sklearn import stopwords,
bbc_dataset, new_example
from Chapter06.lda_topic_gensim import preprocess,
test_new_example
```

2. The **`create_nmf_model`** function creates and returns the model:

```
def create_nmf_model(id_dict, corpus, num_topics):
    nmf_model = Nmf(corpus=corpus,
                    id2word=id_dict,
                    num_topics=num_topics,
                    random_state=100,
                    chunksize=100,
                    passes=50)
    return nmf_model
```

3. The **`plot_coherence`** function plots the coherence of the model as a function of the number of topics:

```
def plot_coherence(id_dict, corpus, texts):
    num_topics_range = range(2, 10)
    coherences = []
    for num_topics in num_topics_range:
        nmf_model = create_nmf_model(id_dict,
corpus,
                                     num_topics)
        coherence_model_nmf = \
        CoherenceModel(model=nmf_model,
texts=texts,
                       dictionary=id_dict,
                       coherence='c_v')
        coherences.append(
            coherence_model_nmf.get_coherence())
    plt.plot(num_topics_range, coherences,
             color='blue', marker='o'
```

```
                color='blue', marker='o',
markersize=5)
    plt.title('Coherence as a function of number of
\
               topics')
    plt.xlabel('Number of topics')
    plt.ylabel('Coherence')
    plt.grid()
    plt.show()
```

4. Read and preprocess the BBC dataset:

```
df = pd.read_csv(bbc_dataset)
df = preprocess(df)
```

5. Create the **`Dictionary`** object and the corpus:

```
texts = df['text'].values
id_dict = corpora.Dictionary(texts)
corpus = [id_dict.doc2bow(text) for text in texts]
```

6. Set the number of topics to be **5** and create the NMF model:

```
number_topics = 5
nmf_model = create_nmf_model(id_dict, corpus,
                             number_topics)
```

7. Print the topics:

```
pprint(nmf_model.print_topics())
```

The results will vary and may appear as follows:

```
[(0,
  '0.017*"people" + 0.013*"music" + 0.008*"mobile"
+ 0.006*"technology" + '
  '0.005*"digital" + 0.005*"phone" + 0.005*"tv" +
0.005*"use" + 0.004*"users" '
  '+ 0.004*"net"'),
(1,
  '0.017*"labour" + 0.014*"party" +
0.013*"election" + 0.012*"blair" + '
  '0.009*"brown" + 0.008*"government" +
0.008*"people" + 0.007*"minister" + '
  '0.006*"howard" + 0.006*"tax"'),
(2,
  '0.009*"government" + 0.008*"bn" + 0.007*"new" +
0.006*"year" + '
  '0.004*"company" + 0.003*"uk" + 0.003*"yukos" +
0.003*"last" + 0.003*"state" '
  '+ 0.003*"market"'),
```

```
(3,
  '0.029*"best" + 0.016*"song" + 0.012*"film" +
0.011*"years" + 0.009*"music" '
  '+ 0.009*"last" + 0.009*"awards" + 0.008*"year" +
0.008*"won" + '
  '0.008*"angels"'),
(4,
  '0.012*"game" + 0.008*"first" + 0.007*"time" +
0.007*"games" + '
  '0.006*"england" + 0.006*"new" + 0.006*"world" +
0.005*"wales" + '
  '0.005*"play" + 0.004*"back"')]
```

8. Now, let's plot model coherence as a function of the number of topics:

```
plot_coherence(id_dict, corpus, texts)
```

Results, again, might vary and may appear as follows:

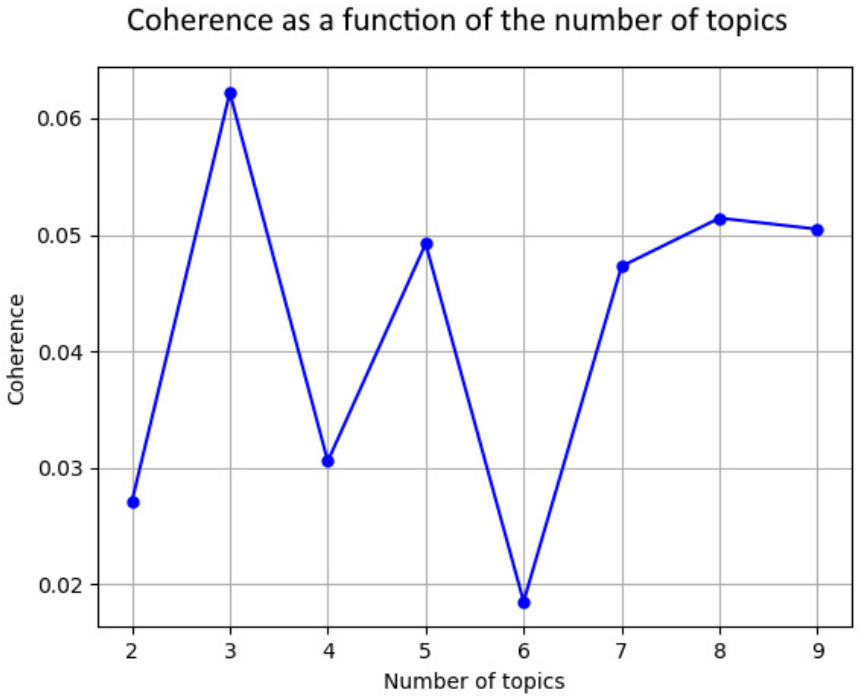

Figure 6.1 – Topic coherence as a function of the number of topics

9. Let's now test a new example, the same example about soccer we used in previous recipes:

```
test_new_example(nmf_model, id_dict, new_example)
```

The result will be as follows:

```
[(2, 0.10007018750350244), (4, 0.8999298124964975)]
```

How it works...

In *step 1*, we import the necessary functions. Many of the functions that work with the **`LdaModel`** object work the same way with the **`Nmf`** object.

In *step 2*, we have the **`create_nmf_model`** function that takes in the corpus of texts, encoded using the **`Dictionary`** object, the **`Dictionary`** object itself, the number of topics, the random state that ensures model reproducibility, chunk size, and the number of passes through the corpus.

In *step 3*, we create the **`plot_coherence`** function that plots the coherence measure against the number of topics in the model. We create a separate NMF model for 2 to 9 topics and measure their coherence.

In *step 4*, we read and preprocess the BBC dataset. In *step 5*, we create the **`Dictionary`** object and use it to encode the texts creating a corpus. In *step 6*, we set the number of topics to **`5`** and create the NMF model. In *step 7*, we then print the topics, which look very similar to the ones modeled by the LDA.

In *step 8*, we plot topic coherence. The plot shows the best coherence for 3, 5, and 8 topics. The more topics, the more granular they will be, so it might be the case that 8 topics make sense, and they divide the 5 main topics into coherent subtopics.

In *step 9*, we test the model with a new example about sports, defined in the *LDA topic modeling with genism* recipe, and it shows topic 4 as the most probable, which is sport, and is correct.

K-means topic modeling with BERT

In this recipe, we will use the K-means algorithm to execute unsupervised topic classification, using the BERT embeddings to encode the data. This recipe shares lots of commonalities with the *Clustering sentences using K-means: unsupervised text classification* recipe from ***Chapter 4***, *Classifying Texts*.

Getting ready

We will be using the **`sklearn.cluster.KMeans`** object to do the unsupervised clustering, along with Hugging Face sentence transformers. To install sentence transformers, use the following commands:

```
conda create -n newenv python=3.6.10 anaconda
```

```
conda install pytorch torchvision cudatoolkit=10.2 -c
pytorch
pip install transformers
pip install -U sentence-transformers
```

How to do it...

The steps for this recipe are as follows:

1. Perform the necessary imports:

```
import re
import string
import pandas as pd
from sklearn.cluster import KMeans
from nltk.probability import FreqDist
from Chapter01.tokenization import tokenize_nltk
from Chapter04.preprocess_bbc_dataset import
get_data
from Chapter04.keyword_classification import
divide_data
from Chapter04.preprocess_bbc_dataset import
get_stopwords
from Chapter04.unsupervised_text_classification
import get_most_frequent_words,
print_most_common_words_by_cluster
from Chapter06.lda_topic_sklearn import stopwords,
bbc_dataset, new_example
from Chapter06.lda_topic_gensim import preprocess
from sentence_transformers import
SentenceTransformer
```

2. Initialize the global variables and read in the stopwords:

```
bbc_dataset = "Chapter04/bbc-text.csv"
stopwords_file_path = "Chapter01/stopwords.csv"
stopwords = get_stopwords(stopwords_file_path)
```

3. Define the **`test_new_example`** function:

```
def test_new_example(km, model, example):
    embedded = model.encode([example])
    topic = km.predict(embedded)[0]
    print(topic)
    return topic
```

4. Read in the data and preprocess it:

```
df = pd.read_csv(bbc_dataset)
```

```
df = preprocess(df)
df['text'] = df['text'].apply(lambda x: "
".join(x))
documents = df['text'].values
```

5. Read in the sentence transformers model and encode the data using it:

```
model = SentenceTransformer('distilbert-base-nli-
mean-tokens')
encoded_data = model.encode(documents)
```

6. Create the **KMeans** model and fit it to the encoded data:

```
km = KMeans(n_clusters=5, random_state=0)
km.fit(encoded_data)
```

7. Print the most common words by cluster:

```
print_most_common_words_by_cluster(documents, km,
5)
```

The result will be as follows:

```
0
['people', 'new', 'mobile', 'technology', 'music',
'users', 'year', 'digital', 'use', 'make', 'get',
'phone', 'net', 'uk', 'games', 'software', 'time',
'tv', …]
1
['government', 'people', 'labour', 'new',
'election', 'party', 'told', 'blair', 'year',
'minister', 'last', 'bn', 'uk', 'public', 'brown',
'time', 'bbc', 'say', 'plans', 'company', …]
2
['year', 'bn', 'growth', 'economy', 'sales',
'economic', 'market', 'prices', 'last', 'bank',
'government', 'new', 'rise', 'dollar', 'figures',
'uk', 'rate', 'years', …]
3
['film', 'best', 'year', 'music', 'new', 'awards',
'first', 'show', 'top', 'award', 'won', 'number',
'last', 'years', 'uk', 'star', 'director', 'world',
'time', 'band', 'three', …]
4
['first', 'year', 'game', 'england', 'win', 'time',
'last', 'world', 'back', 'play', 'new', 'cup',
'team', 'players', 'final', 'wales', 'side',
'ireland', 'good', 'half', 'match', …]
```

8. Now we can test the new sports example using the model:

8. Now we can test the new sports example using the model:

```
test_new_example(km, model, new_example)
```

The output will be as follows:

```
4
```

How it works...

In *step 1*, we import the necessary packages and functions. We reuse several functions from ***Chapter 4***, *Classifying Texts*, and also from previous recipes in this chapter. In *step 2*, we initialize the path to the dataset and get the stopwords.

In *step 3*, we define the **`test_new_example`** function, which is very similar to the **`test_new_example`** function in previous recipes, the only difference being the encoding of the data. We use the sentence transformers model to encode the data and then the **`Kmeans`** model to predict the topic it belongs to.

In *step 4*, we read in the data and preprocess it. The **`preprocess`** function tokenizes the text, puts it into lowercase, and removes the stopwords. We then join the word arrays, since the sentence transformers model takes a string as input.

We read in the sentence transformers model and use it to encode the documents in *step 5*. Then, we read in the DistilBERT-based model, which is smaller than the regular model.

In *step 6*, we create the **`KMeans`** model, initializing it with five clusters and a random state for model reproducibility.

In *step 7*, we print the most common words by cluster, where cluster 0 is **`tech`**, cluster 1 is **`politics`**, cluster 2 is **`business`**, cluster 3 is **`entertainment`**, and cluster 4 is **`sports`**.

In *step 8*, we test the model with a sports example, and it correctly returns cluster 4. Coherence is calculated using Pointwise Mutual Information between each pair of words and then averaging it across all pairs. Pointwise Mutual Information calculates how coincidental it is that two words occur together. Please see more at **https://svn.aksw.org/papers/2015/WSDM_Topic_Evaluation/public.pdf**.

Topic modeling of short texts

In this recipe, we will be using Yelp reviews. These are from the same dataset that we used in ***Chapter 3***, *Representing Text: Capturing Semantics*. We will break the reviews down into sentences and cluster them using the **gsdmm** package. The resulting clusters should be about similar aspects and experience, and while many reviews are about restaurants, there are also other reviews, such as those concerning nail salon ratings.

Getting ready

To install the **gsdmm** package, you will need to create a new folder and then either download the zipped code from GitHub (**https://github.com/rwalk/gsdmm**) or clone it into the created folder using the following command:

```
git clone https://github.com/rwalk/gsdmm.git
```

Then, run the setup script in the folder you installed the package in:

```
python setup.py install
```

How to do it...

In this recipe, we will load the data, divide it into sentences, preprocess it, and then use the **gsdmm** model to cluster the sentences into topics. The steps for this recipe are as follows:

1. Perform the necessary imports:

```
import nltk
import numpy as np
import string
from gsdmm import MovieGroupProcess
from Chapter03.phrases import get_yelp_reviews
from Chapter04.preprocess_bbc_dataset import
get_stopwordss
```

2. Define global variables and load the stopwords:

```
tokenizer = \
nltk.data.load("tokenizers/punkt/english.pickle")
yelp_reviews_file = "Chapter03/yelp-dataset/
review.json"
stopwords_file_path = "Chapter06/
reviews_stopwords.csv"
```

```
stopwords = get_stopwords(stopwords_file_path)
```

3. The **`preprocess`** function cleans up the text:

```
def preprocess(text):
    sentences = tokenizer.tokenize(text)
    sentences =
[nltk.tokenize.word_tokenize(sentence)
                for sentence in sentences]
    sentences = [list(set(word_list)) for word_list
in
                sentences]
    sentences=[[word for word in word_list if word
not
                in stopwords and word not in
                string.punctuation]
               for word_list in sentences]
    return sentences
```

4. The **`top_words_by_cluster`** function prints out the top words by cluster:

```
def top_words(mgp, top_clusters, num_words):
    for cluster in top_clusters:
        sort_dicts = \
        sorted(mgp.cluster_word_distribution[cluste
r].\
               items(), key=lambda k: k[1],
               reverse=True)[:num_words]
        print(f'Cluster {cluster}: {sort_dicts}')
```

5. We now read the reviews and preprocess them:

```
reviews = get_yelp_reviews(yelp_reviews_file)
sentences = preprocess(reviews)
```

6. We then calculate the length of the vocabulary that these sentences contain, as this is a necessary input to the **`GSDMM`** model:

```
vocab = set(word for sentence in sentences for word
in
            sentence)
n_terms = len(vocab)
```

7. Now we can create the model and fit it to the data:

```
mgp = MovieGroupProcess(K=25, alpha=0.1, beta=0.1,
                        n_iters=30)
mgp.fit(sentences, n_terms)
```

8. Now we get the top 15 topics from the model:

```
doc_count = np.array(mgp.cluster_doc_count)
top_clusters = doc_count.argsort()[-15:][::-1]
```

9. We can use the preceding **`top_words_by_cluster`** function to print the most important words for each topic:

```
top_words_by_cluster(mgp, top_clusters, 10)
```

Some of the results will appear as follows:

```
Cluster 6: [('chicken', 1136), ('ordered', 1078),
('sauce', 1073), ('cheese', 863), ('salad', 747),
('delicious', 680), ('fries', 539), ('fresh', 522),
('meat', 466), ('flavor', 465)]
Cluster 4: [('order', 688), ('wait', 626),
('table', 526), ('service', 493), ('people', 413),
('asked', 412), ('server', 342), ('told', 339),
('night', 328), ('long', 316)]
Cluster 5: [('menu', 796), ('prices', 400),
('price', 378), ('service', 355), ('selection',
319), ('quality', 301), ('beer', 289),
('delicious', 277), ('options', 274), ('items',
272)]
Cluster 24: [('room', 456), ('area', 425), ('bar',
419), ('people', 319), ('small', 314),
('restaurant', 312), ('clean', 291), ('tables',
283), ('seating', 268), ('inside', 262)]
Cluster 9: [('service', 1259), ('friendly', 1072),
('staff', 967), ('helpful', 317), ('customer',
310), ('attentive', 250), ('experience', 236),
('server', 210), ('clean', 208), ('people', 166)]
Cluster 3: [('chocolate', 387), ('cream', 339),
('ice', 300), ('tea', 258), ('cake', 219),
('sweet', 212), ('dessert', 186), ('coffee', 176),
('delicious', 176), ('ordered', 175)]
Cluster 18: [('hair', 168), ('nails', 66), ('cut',
60), ('work', 53), ('told', 51), ('massage', 46),
('pain', 46), ('job', 45), ('nail', 43), ('felt',
38)]
…
```

How it works…

In *step 1*, we import the necessary packages and functions. In *step 2*, we define global variables and read in the stopwords. The stopwords used in

this recipe are different from other recipes in that they include a number of words that are common to many reviews, such as *good*, *great*, and *nice*. These adjectives and adverbs are very common in reviews, carry no topic information, and the clustering models frequently create topics around them.

In *step 3*, we define the preprocessing function. This function first splits the text into sentences, tokenizes the sentences into words, and removes duplicates from the word lists. The duplicate removal is necessary for the **`GSDMM`** model, as it requires a list of unique tokens that occur in the text. The preprocessing function then removes stopwords and punctuation from the word lists.

In *step 4*, we define the **`top_words_by_cluster`** function that prints out the most frequent words that appear in each cluster. It sorts the words in each cluster by their frequency and prints out tuples (of word frequency). The number of words per cluster printed is determined by the **`num_words`** parameter.

In *step 5*, we read in the reviews and preprocess them using the **`preprocess`** function defined in *step 3*.

In *step 6*, we get a set of all the unique words in the review sentences by turning the list of words into a set, and then we assign the count of these words to the **`n_terms`** variable to be used later in the creation of the model.

In *step 7*, we create the **`GSDMM`** model. The **`K`** parameter is the *upper bound* on the number of clusters, as the algorithm determines the number of clusters less than or equal to this number. The **`alpha`** parameter controls the probability that a new cluster will be created, and the **`beta`** parameter defines how new text is clustered. If the value of **`beta`** is closer to **`0`**, then text will be clustered more according to similarity, while if it is closer to **`1`**, the clustering will be more based on the frequency of texts. The **`n_iters`** parameter determines the number of passes the algorithm makes through the corpus.

In *step 8*, we get the count of documents by topic and then create a list of the 15 most populous topics. We then use this list in *step 9* to get the **`10`** most frequent words in each cluster.

The results of the clustering make sense for many of the clusters. In the

preceding results, cluster 6 is about food, clusters 4 and 9 relate to the service, cluster 5 is about the selection available, cluster 24 concerns the atmosphere, cluster 3 is about dessert, and cluster 18 is about hair and nail salons.

See also

The **`gsdmm`** package is based on an article by *Yin and Wang*, *A dirichlet multinomial mixture model-based approach for short text clustering*, which can be found online at **https://www.semanticscholar.org/paper/A-dirichlet-multinomial-mixture-model-based-for-Yin-Wang/d03ca28403da15e75bc3e90c21eab44031257e80?p2df**.