CHAPTER 5

Text Summarization

We have come a long way on our journey through the world of text analytics and natural language processing (NLP). You have seen how to process and annotate textual data to use it for various applications. We have also ventured into the world of machine learning (ML) and built our own multi-class text classification system by leveraging various feature-extraction techniques and supervised machine learning algorithms.

In this chapter, we will tackle a slightly different problem in the world of text analytics. The world is rapidly evolving with regard to technology, commerce, business, and media. Gone are the days when we would wait for newspapers to come to our home and be updated about the various events around the world. We now have the Internet and various forms of social media that we consume to stay updated about daily events and stay connected with the world as well as our friends and family. With short messages and statuses, social media websites like Facebook and Twitter have opened up a completely different dimension to sharing and consuming information. We as humans tend to have short attention spans, and this leads us to get bored when consuming or reading large text documents and articles. This brings us to *text summarization*, an extremely important concept in text analytics that is used by businesses and analytical firms to shorten and summarize huge documents of text such that they still retain their key essence or theme and present this summarized information to consumers and clients. This is analogous to an *elevator pitch*, where an executive summary can describe a process, product, service, or business while retaining the core important themes and values in the time it takes to ride an elevator.

Say you have a whole corpus of text documents that ranges from sentences to paragraphs, and you are tasked with trying to derive meaningful insights from it. At first glance, this may seem difficult because you do not even know what to do with these documents, let alone use some analytical or ML techniques on the data. A good way to start would be to use some unsupervised learning approaches specifically aimed at text summarization and information extraction. Here are a few of the things you could do with text documents:

- Extract the key influential phrases from the documents
- Extract various diverse concepts or topics present in the documents
- Summarize the documents to provide a gist that retains the important parts of the whole corpus

This chapter will cover concepts, techniques, and practical implementations of ways to perform all three operations. We can describe our problem formally now, which we will try to solve in this chapter, along with some of the concepts related to it. Given a set of documents, text summarization aims to reduce a document or set of documents in a corpus to a summary of user-specified length such that it retains the key important concepts and themes from the corpus. We will also discuss other ways to summarize documents and extract information from them, including topic models and key phrase extraction.

In this chapter, we will talk about text summarization as well as information extraction from text documents, which captures and summarizes the main themes or concepts of the document corpus. We will start with a detailed discussion of the various types of summarization and information extraction techniques and discuss some concepts essential for understanding the practical implementations later. The chapter will also briefly cover some background dependencies related to text processing and feature extraction before moving on to each technique. We will discuss the three major concepts and techniques of key phrase extraction, topic models, and automated text summarization.

Text Summarization and Information Extraction

Text summarization and information extraction deal with trying to extract key important concepts and themes from a huge corpus of text, essentially reducing it in the process. Before we dive deeper into the concepts and techniques, we should first understand the need for text summarization. The concept of information overload is one of the prime reasons behind the demand for text summarization. Since print and verbal media came into prominence, there has been an abundance of books, articles, audio, and video. This began all the way back in the 3rd or 4th century B.C., when people referred to a huge quantity of books, as there seemed to be no end to the production of books, and this overload of information was often met with disapproval. The Renaissance gave us the invention of the printing press by Gutenberg around 1440 A.D., which led to the mass production of books, manuscripts, articles, and pamphlets. This greatly increased information overload, with scholars complaining about an excess of information, which was becoming extremely difficult to consume, process, and manage.

In the 20th century, advances in computers and technology ushered in the digital age, culminating in the Internet. The Internet opened up a whole window of possibilities into producing and consuming information with social media, news web sites, email, and instant messaging capabilities. This in turn has led to an explosive increase in the amount of information and to unwanted information in the form of spam, unwanted statuses, and tweets—and even to bots posting more unwanted content across the Web.

Information overload, then, is the presence of excess data or information, which consumers find difficult to process in making well-informed decisions. The overload occurs when the amount of information as input to the system starts exceeding the processing capability of the system. We as humans have limited cognitive processing capabilities and are also wired in such a way that we cannot spend a long time reading a single piece of information or data because the mind tends to wander every now and then. Thus when we get loaded with information, it leads to a reduction in making qualitative decisions.

By now you can probably guess where I am going with this concept and why we need summarization and information extraction. Businesses thrive on making key and well-informed decisions and usually they have a huge amount of data and information. Getting insights from it is no piece of cake, and automating it is tough because what to do with all that data is often unclear. Executives rarely have time to listen to long talks or go through pages and pages of important information. The idea of summarization and information extraction is to get an idea of the key important topics and themes of huge documents of information and summarize them into a few lines that can be read, understood, and interpreted easily, thus easing the process of making well-informed decisions in shorter time frames. We need efficient and scalable processes and techniques that can perform this on text data, and the most popular techniques are *keyphrase extraction*, *topic modeling*, and *automated document summarization*. The first two techniques are more into extracting key information in the form of concepts, topics, and themes from documents, thus reducing them, and the last technique is all about summarizing large text documents into a few lines that give the key essence or information which the document is trying to convey. We will cover each technique in detail in future sections along with practical examples but right now, we will briefly talk about what each technique entails and their scope:

- *Keyphrase extraction* is perhaps the simplest out of the three techniques. It involves extracting keywords or phrases from a text document or corpus that capture its main concepts or themes. This can be said to be a simplistic form of topic modeling. You might have seen keywords or phrases described in a research paper or even some product in an online store that describes the entity in a few words or phrases, capturing its main idea or concept.

- *Topic modeling* usually involves using statistical and mathematical modeling techniques to extract main topics, themes, or concepts from a corpus of documents. Note here the emphasis on *corpus* of documents because the more diverse set of documents you have, the more topics or concepts you can generate—unlike with a single document where you will not get too many topics or concepts if it talks about a singular concept. Topic models are also often known as *probabilistic statistical models*, which use specific statistical techniques including singular valued decomposition and latent dirichlet allocation to discover connected latent semantic structures in text data that yield topics and concepts. They are used extensively in text analytics and even bioinformatics.

- *Automated document summarization* is the process of using a computer program or algorithm based on statistical and ML techniques to summarize a document or corpus of documents such that we obtain a short summary that captures all the essential concepts and themes of the original document or corpus. A wide variety of techniques for building automated document summarizers exist, including various extraction- and abstraction-based techniques. The key concept behind all these algorithms is to find a representative subset of the original dataset such that the core essence of the dataset from the semantic and conceptual standpoints is contained in this subset. Document summarization usually involves trying to extract and construct an executive summary from a single document. But the same algorithms can be extended to multiple documents, though usually the idea is not to combine several diverse documents together, which would defeat the purpose of the algorithm. The same concept is not only applied in text analytics but also to image and video summarization.

We will discuss some important mathematical and ML concepts, text normalization, and feature extraction processes in the following sections, before moving to cover each technique in further detail.

Important Concepts

Several important mathematical and ML-based concepts will be useful later on because we will be basing several of our implementations on them. Some will be familiar to you, but I will briefly touch on them again for the sake of completeness so that you can refresh your memory. We will also cover some concepts from natural language processing in this section.

Documents

A *document* is usually an entity containing a whole body of text data with optional headers and other metadata information. A corpus usually consists of a collection of documents. These documents can be simple sentences or complete paragraphs of textual information. *Tokenized corpus* refers to a corpus where each document is tokenized or broken down into *tokens*, which are usually words.

Text Normalization

Text normalization is the process of cleaning, normalizing, and standardizing textual data with techniques like removing special symbols and characters, removing extraneous HTML tags, removing stopwords, correcting spellings, stemming, and lemmatization.

Feature Extraction

Feature extraction is a process whereby we extract meaningful features or attributes from raw textual data for feeding it into a statistical or ML algorithm. This process is also known as *vectorization* because usually the end transformation of this process is numerical vectors from raw text tokens. The reason is that conventional algorithms work on numerical vectors and cannot work directly on raw text data. There are various feature-extraction methods including Bag of Words–based binary features that tell us whether a word or group of words exist or not in the document, Bag of Words–based frequency features that tell us the frequency of occurrence of a word or group of words in a document, and term frequency and inverse document frequency or TF-IDF–weighted features that take into account the term frequency and inverse document frequency when weighing each term. Refer to Chapter 4 for more on feature extraction.

Feature Matrix

A *feature matrix* usually refers to a mapping from a collection of documents to features where each row indicates a document and each column indicates a particular feature, usually a word or a set of words. We will represent collections of documents or sentences through feature matrices after feature extraction and we will often apply statistical and ML techniques on these matrices later on in our practical examples.

Singular Value Decomposition

Singular Value Decomposition (SVD) is a technique from linear algebra that is used quite frequently in summarization algorithms. SVD is the process of factorization of a matrix that is real or complex. Formally we can define SVD as follows. Consider a matrix *M* that has dimensions of $m \times n$ where *m* denotes the number of rows and *n* denotes the number of columns. Mathematically the matrix *M* can be represented using SVD as a factorization such that

$$M_{m\times n} = U_{m\times m}\, S_{m\times n}\, V^T_{n\times n}$$

where we have the following decompositions:

- *U* is an $m \times m$ unitary matrix such that $U^T U = I_{m\times m}$ where I is the identity matrix. The columns of *U* indicate left singular vectors.
- *S* is a diagonal *m x n* matrix with positive real numbers on the diagonal of the matrix. This is also often also represented as a vector of *m* values that indicate the singular values.
- V^T is a $n \times n$ unitary matrix such that $V^T V = I_{n\times n}$ where *I* is the identity matrix. The rows of *V* indicate right singular vectors.

This tells us that *U* and *V* are *orthogonal*. The singular values of *S* are particularly important in summarization algorithms. We will be using SVD particularly for low rank matrix approximation where we approximate the original matrix *M* with a matrix M such that this new matrix is a truncated version of the original matrix *M* with a rank *k* and can be represented by SVD as $M = U S V^T$ where S is a truncated version of the original *S* matrix, which now consists of only the top *k* largest singular values, and the other singular values are represented by zero. We will be using a nice implementation from `scipy` to extract the top *k* singular values and also return the corresponding *U*, *S* and *V* matrices. The following code snippet we will be using is in the `utils.py` file:

```
from scipy.sparse.linalg import svds

def low_rank_svd(matrix, singular_count=2):

    u, s, vt = svds(matrix, k=singular_count)
    return u, s, vt
```

We will be using this implementation in topic modeling as well as document summarization in future sections. Figure 5-1 gives a nice depiction of the preceding process, which yields *k* singular vectors from the original SVD decomposition, and shows how we can get the low rank matrix approximation from the same.

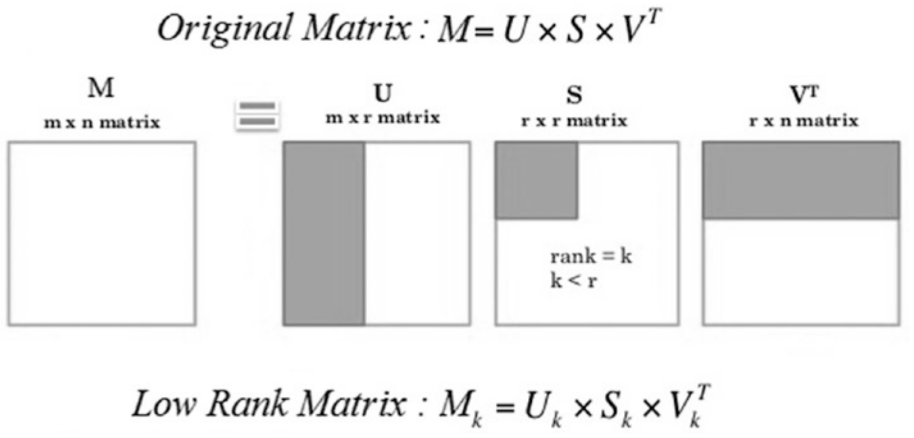

Original Matrix : $M = U \times S \times V^T$
M
m x n matrix
U
m x r matrix
S
r x r matrix
V^T
r x n matrix
rank = k
k < r
Low Rank Matrix : $M_k = U_k \times S_k \times V_k^T$

Figure 5-1. *Singular Value Decomposition with low rank matrix approximation*

You can clearly see that *k* singular values are retained in the low rank matrix approximation and how the original matrix *M* is decomposed into *U*, *S*, and *V* using SVD. In our computations, usually the rows of the matrix *M* will denote terms, and the columns will denote documents. This matrix, also known as the *term-document matrix*, is usually obtained after feature extraction by converting a document-term matrix into its transpose before applying SVD. I will try to keep the math to a minimum in the rest of the chapter unless it is absolutely essential to understand how the algorithms work. The following sections will briefly touch upon text normalization and feature extraction to highlight the techniques and methods that we will be using in this chapter.

Text Normalization

Chapter 3 covered text normalization in detail, and we built our own normalization module in Chapter 4. We will be reusing the same module in this chapter but will be adding a couple of enhancements specifically for the benefit of some of our algorithms. You can find all the text normalization–related code in the `normalization.py` file. The main steps performed in text normalization include the following:

1. Sentence extraction
2. Unescape HTML escape sequences
3. Expand contractions
4. Lemmatize text
5. Remove special characters
6. Remove stopwords

Steps 3–6 remain the same from Chapter 4, except step 5 where we substitute each special character with a blank space depicted by the code `pattern.sub(' ', token)` instead of the empty string in Chapter 4.

Step 1 is a new function where we take in a text document, remove its newlines, parse the text, converting it into ASCII format, and break it down into its sentence constituents. The function is depicted in the following snippet:

```
def parse_document(document):
    document = re.sub('\n', ' ', document)
    if isinstance(document, str):
        document = document
    elif isinstance(document, unicode):
        return unicodedata.normalize('NFKD', document).encode('ascii',
        'ignore')
    else:
        raise ValueError('Document is not string or unicode!')
    document = document.strip()
    sentences = nltk.sent_tokenize(document)
    sentences = [sentence.strip() for sentence in sentences]
    return sentences
```

Step 2 deals with unescaping special HTML characters that are escaped or encoded. The full list at `www.theukwebdesigncompany.com/articles/entity-escape-characters.php` basically shows how some special symbols or even regular characters are escaped into a different code, for example, & is escaped as `&#38;`. So we use the following function to unescape them and bring them back to their original unescaped form so we can normalize them properly in the subsequent stages:

```
from HTMLParser import HTMLParser

html_parser = HTMLParser()
def unescape_html(parser, text):
    return parser.unescape(text)
```

We also parameterize our lemmatization operation in our final normalization function so as to make it optional because in some scenarios it works perfectly while in other scenarios we may not want to use lemmatization. The complete normalization function is depicted as follows:

```
def normalize_corpus(corpus, lemmatize=True, tokenize=False):

    normalized_corpus = []
    for text in corpus:
        text = html_parser.unescape(text)
        text = expand_contractions(text, CONTRACTION_MAP)
        if lemmatize:
            text = lemmatize_text(text)
        else:
            text = text.lower()
        text = remove_special_characters(text)
        text = remove_stopwords(text)
        if tokenize:
            text = tokenize_text(text)
            normalized_corpus.append(text)
        else:
            normalized_corpus.append(text)

    return normalized_corpus
```

We will be using this function for most of our normalization needs. Refer to the `normalization.py` file for all the detailed helper functions we use for normalizing text which we also discussed in Chapter 4.

Feature Extraction

We will use a generic function here to perform various types of feature extraction from text data. The types of features which we will be working with are as follows:

- Binary term occurrence–based features
- Frequency bag of words–based features
- TF-IDF–weighted features

We will use the following function in most of our practical examples in future sections for feature extraction from text documents. You can also find this function in the `utils.py` module in the code files associated with this chapter:

```
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer

def build_feature_matrix(documents, feature_type='frequency'):

    feature_type = feature_type.lower().strip()

    if feature_type == 'binary':
        vectorizer = CountVectorizer(binary=True, min_df=1,
                                     ngram_range=(1, 1))
    elif feature_type == 'frequency':
        vectorizer = CountVectorizer(binary=False, min_df=1,
                                     ngram_range=(1, 1))
    elif feature_type == 'tfidf':
        vectorizer = TfidfVectorizer(min_df=1,
                                     ngram_range=(1, 1))
    else:
        raise Exception("Wrong feature type entered. Possible values:
        'binary', 'frequency', 'tfidf'")

    feature_matrix = vectorizer.fit_transform(documents).astype(float)

    return vectorizer, feature_matrix
```

Now that we have covered the necessary background concepts and dependencies needed for this chapter, we will be deep diving into each text summarization and information extraction technique in detail.

Keyphrase Extraction

One of the simplest yet most powerful techniques of extracting important information from unstructured text documents is keyphrase extraction. *Keyphrase extraction,* also known as *terminology extraction,* is defined as the process or technique of extracting key important and relevant terms or phrases from a body of unstructured text such that the core topics or themes of the text document(s) are captured in these key phrases. This technique falls under the broad umbrella of information retrieval and extraction. Keyphrase extraction finds its uses in many areas, including the following:

- Semantic web
- Query-based search engines and crawlers
- Recommendation systems
- Tagging systems

- Document similarity
- Translation

Keyphrase extraction is often the starting point for carrying out more complex tasks in text analytics or NLP, and the output from this can itself act as features for more complex systems. There are various approaches for keyphrase extraction. We will be covering the following two techniques:

- Collocations
- Weighted tag–based phrase extraction

An important thing to remember here is that we will be extracting phrases that are usually collections of words, though sometimes that can include a single word. If you are extracting keywords, that is also known as *keyword extraction*, and it is a subset of keyphrase extraction.

Collocations

The term *collocation* is actually a concept borrowed from analyzing corpora and linguistics. A collocation is a sequence or group of words that tend to occur frequently such that this frequency tends to be more than what could be termed as a random or chance occurrence. Various types of collocations can be formed based on the parts of speech of the various terms like nouns, verbs, and so on. There are various ways to extract collocations, and one of the best is to use an n-gram grouping or segmentation approach where we construct n-grams out of a corpus, count the frequency of each n-gram, and rank them based on their frequency of occurrence to get the most frequent n-gram collocations.

The idea is to have a corpus of documents, which could be paragraphs or sentences, tokenize them to form sentences, flatten the list of sentences to form one large sentence or string, over which we slide a window of size *n* based on the n-gram range, and compute n-grams across the string. Once computed, we count each n-gram based on its frequency of occurrence and then rank them based on their frequency. This yields the most frequent collocations on the basis of frequency.

We will implement this from scratch initially so that you can understand the algorithm better and then we will use some of `nltk's` built-in capabilities to show the same. We will start by loading some necessary dependencies and a corpus on which we will be computing collocations. We will use the `nltk` Gutenberg corpus's book, Lewis Carroll's *Alice in Wonderland* as our corpus. We also normalize the corpus to standardize the text content using our normalization module specified earlier:

```
from nltk.corpus import gutenberg
from normalization import normalize_corpus
import nltk
from operator import itemgetter

# load corpus
alice = gutenberg.sents(fileids='carroll-alice.txt')
alice = [' '.join(ts) for ts in alice]
norm_alice = filter(None, normalize_corpus(alice, lemmatize=False))
```

```
# print first line
In [772]: print norm_alice[0]
alice adventures wonderland lewis carroll 1865
```

Now that we have loaded our corpus, we will define a function to flatten the corpus into one big string of text. The following function will help us do that for a corpus of documents:

```
def flatten_corpus(corpus):
    return ' '.join([document.strip()
                     for document in corpus])
```

We will define a function to compute n-grams based on some input list of tokens and the parameter *n*, which determines the degree of the n-gram like a unigram, bigram, and so on. The following code snippet computes n-grams for an input sequence:

```
def compute_ngrams(sequence, n):
    return zip(*[sequence[index:]
                 for index in range(n)])
```

This function basically takes in a sequence of tokens and computes a list of lists having sequences where each list contains all items from the previous list except the first item removed from the previous list. It constructs *n* such lists and then zips them all together to give us the necessary n-grams. We can see the function in action on a sample sequence in the following snippet:

```
In [802]: compute_ngrams([1,2,3,4], 2)
Out[802]: [(1, 2), (2, 3), (3, 4)]

In [803]: compute_ngrams([1,2,3,4], 3)
Out[803]: [(1, 2, 3), (2, 3, 4)]
```

The preceding output shows bigrams and trigrams for an input sequence. We will now utilize this function and build upon it to generate the top n-grams based on their frequency of occurrence. The following code snippet helps us in getting the top n-grams:

```
def get_top_ngrams(corpus, ngram_val=1, limit=5):

    corpus = flatten_corpus(corpus)
    tokens = nltk.word_tokenize(corpus)

    ngrams = compute_ngrams(tokens, ngram_val)
    ngrams_freq_dist = nltk.FreqDist(ngrams)
    sorted_ngrams_fd = sorted(ngrams_freq_dist.items(),
                              key=itemgetter(1), reverse=True)
    sorted_ngrams = sorted_ngrams_fd[0:limit]
    sorted_ngrams = [(' '.join(text), freq)
                     for text, freq in sorted_ngrams]

    return sorted_ngrams
```

We make use of `nltk`'s `FreqDist` class to create a counter of all the n-grams based on their frequency and then we sort them based on their frequency and return the top n-grams based on the specified user limit. We will now compute the top bigrams and trigrams on our corpus using the following code snippet:

```
# top 10 bigrams
In [805]: get_top_ngrams(corpus=norm_alice, ngram_val=2,
     ...:                limit=10)
Out[805]:
[(u'said alice', 123),
 (u'mock turtle', 56),
 (u'march hare', 31),
 (u'said king', 29),
 (u'thought alice', 26),
 (u'said hatter', 22),
 (u'white rabbit', 22),
 (u'said mock', 20),
 (u'said gryphon', 18),
 (u'said caterpillar', 18)]

# top 10 trigrams
In [806]: get_top_ngrams(corpus=norm_alice, ngram_val=3,
     ...:                limit=10)
Out[806]:
[(u'said mock turtle', 20),
 (u'said march hare', 10),
 (u'poor little thing', 6),
 (u'white kid gloves', 5),
 (u'little golden key', 5),
 (u'march hare said', 5),
 (u'certainly said alice', 5),
 (u'mock turtle said', 5),
 (u'mouse mouse mouse', 4),
 (u'join dance join', 4)]
```

The preceding output shows sequences of two and three words generated by n-grams along with the number of times they occur throughout the corpus. We can see most of the collocations point to people who are speaking something as *"said <person>".* We also see the people who are popular characters in *"Alice in Wonderland"* like the *mock turtle*, the *king*, the *rabbit*, the *hatter*, and of course *Alice* herself being depicted in the aforementioned collocations.

We will now look at `nltk`'s collocation finders, which enable us to find collocations using various measures like raw frequencies, pointwise mutual information, and so on. Just to explain briefly, *pointwise mutual information* can be computed for two events or terms as the logarithm of the ratio of the probability of them occurring together by the product of their individual probabilities assuming that they are independent of each other. Mathematically we can represent it like this:

$$pmi(x,y) = log\,\frac{p(x,y)}{p(x)p(y)}$$

This measure is symmetric. The following code snippet shows how to compute these collocations using these measures:

```
# bigrams
from nltk.collocations import BigramCollocationFinder
from nltk.collocations import BigramAssocMeasures

finder = BigramCollocationFinder.from_documents([item.split()
                                                 for item
                                                 in norm_alice])
bigram_measures = BigramAssocMeasures()
# raw frequencies
In [813]: finder.nbest(bigram_measures.raw_freq, 10)
Out[813]:
[(u'said', u'alice'),
 (u'mock', u'turtle'),
 (u'march', u'hare'),
 (u'said', u'king'),
 (u'thought', u'alice'),
 (u'said', u'hatter'),
 (u'white', u'rabbit'),
 (u'said', u'mock'),
 (u'said', u'caterpillar'),
 (u'said', u'gryphon')]
# pointwise mutual information
In [814]: finder.nbest(bigram_measures.pmi, 10)
Out[814]:
[(u'abide', u'figures'),
 (u'acceptance', u'elegant'),
 (u'accounting', u'tastes'),
 (u'accustomed', u'usurpation'),
 (u'act', u'crawling'),
 (u'adjourn', u'immediate'),
 (u'adoption', u'energetic'),
 (u'affair', u'trusts'),
 (u'agony', u'terror'),
 (u'alarmed', u'proposal')]

# trigrams
from nltk.collocations import TrigramCollocationFinder
from nltk.collocations import TrigramAssocMeasures

finder = TrigramCollocationFinder.from_documents([item.split()
                                                 for item
                                                 in norm_alice])
```

```
trigram_measures = TrigramAssocMeasures()
# raw frequencies
In [817]: finder.nbest(trigram_measures.raw_freq, 10)
Out[817]:
[(u'said', u'mock', u'turtle'),
 (u'said', u'march', u'hare'),
 (u'poor', u'little', u'thing'),
 (u'little', u'golden', u'key'),
 (u'march', u'hare', u'said'),
 (u'mock', u'turtle', u'said'),
 (u'white', u'kid', u'gloves'),
 (u'beau', u'ootiful', u'soo'),
 (u'certainly', u'said', u'alice'),
 (u'might', u'well', u'say')]
# pointwise mutual information
In [818]: finder.nbest(trigram_measures.pmi, 10)
Out[818]:
[(u'accustomed', u'usurpation', u'conquest'),
 (u'adjourn', u'immediate', u'adoption'),
 (u'adoption', u'energetic', u'remedies'),
 (u'ancient', u'modern', u'seaography'),
 (u'apple', u'roast', u'turkey'),
 (u'arithmetic', u'ambition', u'distraction'),
 (u'brother', u'latin', u'grammar'),
 (u'canvas', u'bag', u'tied'),
 (u'cherry', u'tart', u'custard'),
 (u'circle', u'exact', u'shape')]
```

Now you know how to compute collocations for a corpus using an n-gram generative approach. We will now look at a better way of generating key phrases based on parts of speech tagging and term weighing in the next section.

Weighted Tag–Based Phrase Extraction

Here's a slightly different approach to extracting keyphrases. This method borrows concepts from a couple of papers, namely K. Barker and N. Cornachhia's "Using Noun Phrase Heads to Extract Document Keyphrases" and "KEA: Practical Automatic Keyphrase Extraction" by Ian Witten et al., which you can refer to for further details on their experimentations and approaches. We follow a two-step process in our algorithm here:

1. Extract all noun phrases chunks using shallow parsing
2. Compute TF-IDF weights for each chunk and return the top weighted phrases

For the first step, we will use a simple pattern based on parts of speech (POS) tags to extract noun phrase chunks. You will be familiar with this from Chapter 3 where we explored chunking and shallow parsing. Before discussing our algorithm, let us define the corpus on which we will be testing our implementation. We use a sample description of elephants taken from Wikipedia as shown in the following code:

```
toy_text = """
Elephants are large mammals of the family Elephantidae
and the order Proboscidea. Two species are traditionally recognised,
the African elephant and the Asian elephant. Elephants are scattered
throughout sub-Saharan Africa, South Asia, and Southeast Asia. Male
African elephants are the largest extant terrestrial animals. All
elephants have a long trunk used for many purposes,
particularly breathing, lifting water and grasping objects. Their
incisors grow into tusks, which can serve as weapons and as tools
for moving objects and digging. Elephants' large ear flaps help
to control their body temperature. Their pillar-like legs can
carry their great weight. African elephants have larger ears
and concave backs while Asian elephants have smaller ears
and convex or level backs.
"""
```

Now that we have our corpus ready, we will use the pattern, `" NP: {<DT>? <JJ>* <NN.*>+}"` for extracting all possible noun phrases from our corpus of documents/sentences. You can always experiment with more sophisticated patterns later, incorporating verb, adjective, or even adverb phrases. However, I will keep things simple and concise here to focus on the core logic. Once we have our pattern, we will define a function to parse and extract these phrases using the following snippet (we also load the necessary dependencies at this point):

```
from normalization import parse_document
import itertools
import nltk
from normalization import stopword_list
from gensim import corpora, models

def get_chunks(sentences, grammar = r'NP: {<DT>? <JJ>* <NN.*>+}'):
    # build chunker based on grammar pattern
    all_chunks = []
    chunker = nltk.chunk.regexp.RegexpParser(grammar)

    for sentence in sentences:
        # POS tag sentences
        tagged_sents = nltk.pos_tag_sents(
                            [nltk.word_tokenize(sentence)])
        # extract chunks
        chunks = [chunker.parse(tagged_sent)
                  for tagged_sent in tagged_sents]
        # get word, pos tag, chunk tag triples
        wtc_sents = [nltk.chunk.tree2conlltags(chunk)
                     for chunk in chunks]

        flattened_chunks = list(
                            itertools.chain.from_iterable(
                                wtc_sent for wtc_sent in wtc_sents)
                           )
        # get valid chunks based on tags
        valid_chunks_tagged = [(status, [wtc for wtc in chunk])
                        for status, chunk
                        in itertools.groupby(flattened_chunks,
                                         lambda (word,pos,chunk): chunk
                                         != 'O')]
        # append words in each chunk to make phrases
        valid_chunks = [' '.join(word.lower()
                                for word, tag, chunk
                                in wtc_group
                                    if word.lower()
                                        not in stopword_list)
                                    for status, wtc_group
                                    in valid_chunks_tagged
                                        if status]
        # append all valid chunked phrases
        all_chunks.append(valid_chunks)

    return all_chunks
```

The comments in the preceding function are self-explanatory. Basically, we have a defined grammar pattern for chunking or extracting noun phrases. We define a chunker over the same pattern, and for each sentence in the document, we first annotate it with its POS tags (hence, we should not normalize the text) and then build a shallow parse tree with noun phrases as the chunks and all other POS tag–based words as chinks, which are not parts of any chunks. Once this is done, we use the `tree2conlltags` function to generate (w,t,c) triples, which are words, POS tags, and the IOB-formatted chunk tags discussed in Chapter 3. We remove all tags with chunk tag of `'O'` since they are basically words or terms that do not belong to any chunk (if you remember our discussion of shallow parsing in Chapter 3). Finally, from these valid chunks, we combine the chunked terms to generate phrases from each chunk group. We can see this function in action on our corpus in the following snippet:

```
sentences = parse_document(toy_text)
valid_chunks = get_chunks(sentences)
# print all valid chunks
In [834]: print valid_chunks
 [['elephants', 'large mammals', 'family elephantidae', 'order
proboscidea'], ['species', 'african elephant', 'asian elephant'],
['elephants', 'sub-saharan africa', 'south asia', 'southeast asia'],
['male african elephants', 'extant terrestrial animals'], ['elephants',
'long trunk', 'many purposes', 'breathing', 'water', 'grasping objects'],
['incisors', 'tusks', 'weapons', 'tools', 'objects', 'digging'],
['elephants', 'large ear flaps', 'body temperature'], ['pillar-like legs',
'great weight'], ['african elephants', 'ears', 'backs', 'asian elephants',
'ears', 'convex', 'level backs']]
```

The preceding output shows all the valid keyphrases per sentence of our document. You can already see, since we targeted noun phrases, all phrases talk about noun based entities. We will now build on top of our `get_chunks()` function by implementing the necessary logic for step 2, where we will build a TF-IDF–based model on our keyphrases using `gensim` and then compute TF-IDF–based weights for each keyphrase based on its occurrence in the corpus. Finally, we will sort these keyphrases based on their TF-IDF weights and show the top *n* keyphrases where *n* is specified by the user:

```
def get_tfidf_weighted_keyphrases(sentences,
                                  grammar=r'NP: {<DT>? <JJ>* <NN.*>+}',
                                  top_n=10):
    # get valid chunks
    valid_chunks = get_chunks(sentences, grammar=grammar)
    # build tf-idf based model
    dictionary = corpora.Dictionary(valid_chunks)
    corpus = [dictionary.doc2bow(chunk) for chunk in valid_chunks]
    tfidf = models.TfidfModel(corpus)
    corpus_tfidf = tfidf[corpus]
    # get phrases and their tf-idf weights
    weighted_phrases = {dictionary.get(id): round(value,3)
                        for doc in corpus_tfidf
                        for id, value in doc}
    weighted_phrases = sorted(weighted_phrases.items(),
                              key=itemgetter(1), reverse=True)
    # return top weighted phrases
    return weighted_phrases[:top_n]
```

We can now test this function on our toy corpus from before by using the following code snippet to generate the top ten keyphrases:

```
# top 10 tf-idf weighted keyphrases for toy_text
In [836]: get_tfidf_weighted_keyphrases(sentences, top_n=10)
Out[836]:
[(u'pillar-like legs', 0.707),
 (u'male african elephants', 0.707),
 (u'great weight', 0.707),
 (u'extant terrestrial animals', 0.707),
 (u'large ear flaps', 0.684),
 (u'body temperature', 0.684),
 (u'ears', 0.667),
 (u'species', 0.577),
 (u'african elephant', 0.577),
 (u'asian elephant', 0.577)]
```

Interestingly we see various types of elephants being depicted in the keyphrases, like Asian and African elephants, and also typical attributes of elephants like `"great weight"`, `"large ear flaps"`, and `"pillar like legs"`. Thus you can get an idea of how keyphrase extraction can extract key important concepts from text documents and summarize them. Try out these functions on other corpora to see interesting results!

Topic Modeling

We have seen how keyphrases can be extracted using a couple of techniques. Though these phrases point out key pivotal points from a document or corpus, it is simplistic and often does not portray the various themes or concepts in a corpus, particularly when we have different distinguishing themes or concepts in a corpus of documents. Topic models have been designed specifically for the purpose of extracting various distinguishing concepts or topics from a large corpus containing various types of documents, where each document talks about one or more concepts. These concepts can be anything from thoughts to opinions, facts, outlooks, statements, and so on. The main aim of topic modeling is to use mathematical and statistical techniques to discover hidden and latent semantic structures in a corpus.

Topic modeling involves extracting features from document terms and using mathematical structures and frameworks like matrix factorization and SVD to generate clusters or groups of terms that are distinguishable from each other, and these cluster of words form topics or concepts. These concepts can be used to interpret the main themes of a corpus and also make semantic connections among words that co-occur together frequently in various documents. There are various frameworks and algorithms to build topic models. We will cover the following three methods:

- Latent semantic indexing
- Latent Dirichlet allocation
- Non-negative matrix factorization

The first two methods are quite popular and have been around a long time. The last technique, non-negative matrix factorization, is a very recent technique that is extremely effective and gives excellent results. We will leverage `gensim` and `scikit-learn` for our practical implementations and also look at how to build our own topic model based on latent semantic indexing. This will give you an idea of how these techniques work and also how to convert mathematical frameworks into practical implementations. We will use the following toy corpus initially to test our topic models:

```
toy_corpus = ["The fox jumps over the dog",
"The fox is very clever and quick",
"The dog is slow and lazy",
"The cat is smarter than the fox and the dog",
"Python is an excellent programming language",
"Java and Ruby are other programming languages",
"Python and Java are very popular programming languages",
"Python programs are smaller than Java programs"]
```

You can see that we have eight documents in the preceding corpus: the first four talk about various animals, and the last four are about programming languages. Thus this shows that there are two distinct topics in the corpus. We generalized that using our brains, but the following sections will try to extract that same information using computational methods. Once we build some topic modeling frameworks, we will use the same to generate topics on real product reviews from Amazon.

Latent Semantic Indexing

Our first technique is latent semantic indexing (LSI), which has been around since the 1970s when it was first developed as a statistical technique to correlate and find out semantically linked terms from corpora. LSI is not just used for text summarization but also in information retrieval and search. LSI uses the very popular SVD technique discussed earlier in the "Important Concepts" section. The main principle behind LSI is that similar terms tend to be used in the same context and hence tend to co-occur more. The term *LSI* comes from the fact that this technique has the ability to uncover latent hidden terms which correlate semantically to form topics.

We will now try to implement an LSI by leveraging `gensim` and extract topics from the toy corpus. To start, we load the necessary dependencies and normalize the toy corpus using the following code snippet:

```
from gensim import corpora, models
from normalization import normalize_corpus
import numpy as np

norm_tokenized_corpus = normalize_corpus(toy_corpus, tokenize=True)
# view the normalized tokenized corpus
In [841]: norm_tokenized_corpus
Out[841]:
[[u'fox', u'jump', u'dog'],
 [u'fox', u'clever', u'quick'],
 [u'dog', u'slow', u'lazy'],
 [u'cat', u'smarter', u'fox', u'dog'],
 [u'python', u'excellent', u'programming', u'language'],
 [u'java', u'ruby', u'programming', u'language'],
 [u'python', u'java', u'popular', u'programming', u'language'],
 [u'python', u'program', u'small', u'java', u'program']]
```

We now build a dictionary or vocabulary, which `gensim` uses to map each unique term into a numeric value. Once built, we convert the preceding tokenized corpus into a numeric Bag of Words vector representation where each term and its frequency in a sentence is depicted by a tuple (term, frequency), as seen in the following snippet:

```
# build the dictionary
dictionary = corpora.Dictionary(norm_tokenized_corpus)
```

```
# view the dictionary mappings
In [846]: print dictionary.token2id
{u'program': 17, u'lazy': 5, u'clever': 4, u'java': 13, u'programming': 10,
u'language': 11, u'python': 9, u'smarter': 7, u'fox': 1, u'dog': 2, u'cat':
8, u'jump': 0, u'popular': 15, u'slow': 6, u'excellent': 12, u'quick': 3,
u'small': 16, u'ruby': 14}

# convert tokenized documents into bag of words vectors
corpus = [dictionary.doc2bow(text) for text in norm_tokenized_corpus]
# view the converted vectorized corpus
In [849]: corpus
Out[849]:
[[(0, 1), (1, 1), (2, 1)],
 [(1, 1), (3, 1), (4, 1)],
 [(2, 1), (5, 1), (6, 1)],
 [(1, 1), (2, 1), (7, 1), (8, 1)],
 [(9, 1), (10, 1), (11, 1), (12, 1)],
 [(10, 1), (11, 1), (13, 1), (14, 1)],
 [(9, 1), (10, 1), (11, 1), (13, 1), (15, 1)],
 [(9, 1), (13, 1), (16, 1), (17, 2)]]
```

We will now build a TF-IDF–weighted model over this corpus where each term in each document will contain its TF-IDF weight. This is analogous to feature extraction or vector space transformation where each document is represented by a TF-IDF vector of its terms, as we have done in the past. Once this is done, we build an LSI model on these features and take an input of the number of topics we want to generate. This number is based on intuition and trial and error, so feel free to play around with this parameter when you build topic models on corpora. We will set this parameter to 2, based on the number of topics we expect our toy corpus to contain:

```
# build tf-idf feature vectors
tfidf = models.TfidfModel(corpus)
corpus_tfidf = tfidf[corpus]

# fix the number of topics
total_topics = 2

# build the topic model
lsi = models.LsiModel(corpus_tfidf,
                      id2word=dictionary,
                      num_topics=total_topics)
```

Now that our topic modeling framework is built, we can see the generated topics in the following code snippet:

```
In [855]: for index, topic in lsi.print_topics(total_topics):
     ...:     print 'Topic #'+str(index+1)
     ...:     print topic
     ...:     print
```

```
Topic #1
-0.459*"language" + -0.459*"programming" + -0.344*"java" + -0.344*"python" +
-0.336*"popular" + -0.318*"excellent" + -0.318*"ruby" + -0.148*"program" +
-0.074*"small" + -0.000*"clever"

Topic #2
0.459*"dog" + 0.459*"fox" + 0.444*"jump" + 0.322*"smarter" + 0.322*"cat" +
0.208*"lazy" + 0.208*"slow" + 0.208*"clever" + 0.208*"quick" + -0.000*"ruby"
```

Let's take a moment to understand those results. At first, ignoring the weights, you can see that the first topic contains terms related to programming languages and the second topic contains terms related to animals, which is in line with the main two concepts from our toy corpus mentioned earlier. If you now look at the weights, higher weightage and same sign exists for the terms that contribute toward each of the topics. The first topic has related terms with negative weights, and the second topic has related terms with positive weights. The sign just indicates the direction of the topic, that is, similar correlated terms in the topics will have the same sign or direction. The following function helps display the topics in a better way with or without thresholds:

```
def print_topics_gensim(topic_model, total_topics=1,
                        weight_threshold=0.0001,
                        display_weights=False,
                        num_terms=None):

    for index in range(total_topics):
        topic = topic_model.show_topic(index)
        topic = [(word, round(wt,2))
                 for word, wt in topic
                 if abs(wt) >= weight_threshold]
        if display_weights:
            print 'Topic #'+str(index+1)+' with weights'
            print topic[:num_terms] if num_terms else topic
        else:
            print 'Topic #'+str(index+1)+' without weights'
            tw = [term for term, wt in topic]
            print tw[:num_terms] if num_terms else tw
        print
```

We can try out this function on our toy corpus topic model using the following snippet to see how we can get the topics and play around with the parameters:

```
# print topics without weights
In [860]: print_topics_gensim(topic_model=lsi,
     ...:                     total_topics=total_topics,
     ...:                     num_terms=5,
     ...:                     display_weights=False)
Topic #1 without weights
[u'language', u'programming', u'java', u'python', u'popular']
```

```
Topic #2 without weights
[u'dog', u'fox', u'jump', u'smarter', u'cat']

# print topics with their weights
In [861]: print_topics_gensim(topic_model=lsi,
     ...:                     total_topics=total_topics,
     ...:                     num_terms=5,
     ...:                     display_weights=True)
Topic #1 with weights
[(u'language', -0.46), (u'programming', -0.46), (u'java', -0.34),
(u'python', -0.34), (u'popular', -0.34)]

Topic #2 with weights
[(u'dog', 0.46), (u'fox', 0.46), (u'jump', 0.44), (u'smarter', 0.32),
(u'cat', 0.32)]
```

We have successfully built a topic modeling framework using LSI that can distinguish and show topics from a corpus of documents. Now we will use SVD to build our own LSI topic model framework from the ground up using the mathematical concepts discussed at the beginning of this chapter. We will start by building a TF-IDF feature matrix, which is actually a document-term matrix (if you remember from our classification exercise in Chapter 4). We will transpose this to form a term-document matrix before computing SVD using the following snippet. Besides this, we also fix the number of topics we want to generate and extract the term names from the features so we can map them with their weights:

```
from utils import build_feature_matrix, low_rank_svd

# build the term document tf-idf weighted matrix
norm_corpus = normalize_corpus(toy_corpus)
vectorizer, tfidf_matrix = build_feature_matrix(norm_corpus,
                                                feature_type='tfidf')
td_matrix = tfidf_matrix.transpose()
td_matrix = td_matrix.multiply(td_matrix > 0)

# fix total topics and get the terms used in the term-document matrix
total_topics = 2
feature_names = vectorizer.get_feature_names()
```

Once this is done, we compute the SVD for our term-document matrix using our `low_rank_svd()` function such that we build a low ranked matrix approximation taking only the top *k* singular vectors, which will be equal to our number of topics in this case. Using the *S* and *U* components, we multiply them together to generate each term and its weightage per topic giving us the necessary weights per topic similar to what you saw earlier:

```
u, s, vt = low_rank_svd(td_matrix, singular_count=total_topics)
weights = u.transpose() * s[:, None]
```

Now that we have our term weights, we need to connect them back to our terms. We define two utility functions for generating these topics by connecting the terms with their weights and then printing these topics using a function with configurable parameters:

```
# get topics with their terms and weights
def get_topics_terms_weights(weights, feature_names):
    feature_names = np.array(feature_names)
    sorted_indices = np.array([list(row[::-1])
                           for row
                           in np.argsort(np.abs(weights))])
    sorted_weights = np.array([list(wt[index])
                               for wt, index
                               in zip(weights,sorted_indices)])
    sorted_terms = np.array([list(feature_names[row])
                             for row
                             in sorted_indices])

    topics = [np.vstack((terms.T,
                     term_weights.T)).T
              for terms, term_weights
              in zip(sorted_terms, sorted_weights)]

    return topics

# print all the topics from a corpus
def print_topics_udf(topics, total_topics=1,
                     weight_threshold=0.0001,
                     display_weights=False,
                     num_terms=None):

    for index in range(total_topics):
        topic = topics[index]
        topic = [(term, float(wt))
                 for term, wt in topic]
        topic = [(word, round(wt,2))
                 for word, wt in topic
                 if abs(wt) >= weight_threshold]

        if display_weights:
            print 'Topic #'+str(index+1)+' with weights'
            print topic[:num_terms] if num_terms else topic
        else:
            print 'Topic #'+str(index+1)+' without weights'
            tw = [term for term, wt in topic]
            print tw[:num_terms] if num_terms else tw
        print
```

We are now ready to see our function in action. The following snippet utilizes the previously defined functions to generate topics using our LSI implementation using SVD by connecting the terms with their weights for each topic:

```
In [871]: topics = get_topics_terms_weights(weights, feature_names)
     ...: print_topics_udf(topics=topics,
     ...:                  total_topics=total_topics,
     ...:                  weight_threshold=0,
     ...:                  display_weights=True)
Topic #1 with weights
[(u'dog', 0.72), (u'fox', 0.72), (u'jump', 0.43), (u'smarter', 0.34),
(u'cat', 0.34), (u'slow', 0.23), (u'lazy', 0.23), (u'quick', 0.23),
(u'clever', 0.23), (u'program', 0.0), (u'java', 0.0), (u'excellent', -0.0),
(u'small', 0.0), (u'popular', 0.0), (u'python', 0.0), (u'programming',
-0.0), (u'language', -0.0), (u'ruby', 0.0)]

Topic #2 with weights
[(u'programming', -0.73), (u'language', -0.73), (u'python', -0.56),
(u'java', -0.56), (u'popular', -0.34), (u'ruby', -0.33), (u'excellent',
-0.33), (u'program', -0.21), (u'small', -0.11), (u'fox', 0.0), (u'dog',
0.0), (u'jump', 0.0), (u'clever', 0.0), (u'quick', 0.0), (u'lazy', 0.0),
(u'slow', 0.0), (u'smarter', 0.0), (u'cat', 0.0)]
```

From the preceding output we see that both topics have all the terms, but notice the weights more minutely. Do you see any difference? Of course, the terms in topic one related to programming have *zero* value, indicating they do not contribute to the topic at all. Let us put a proper threshold and get only the relevant terms per topic as follows:

```
# applying a scoring threshold
In [874]: topics = get_topics_terms_weights(weights, feature_names)
     ...: print_topics_udf(topics=topics,
     ...:                  total_topics=total_topics,
     ...:                  weight_threshold=0.15,
     ...:                  display_weights=True)
Topic #1 with weights
[(u'dog', 0.72), (u'fox', 0.72), (u'jump', 0.43), (u'smarter', 0.34),
(u'cat', 0.34), (u'slow', 0.23), (u'lazy', 0.23), (u'quick', 0.23),
(u'clever', 0.23)]

Topic #2 with weights
[(u'programming', -0.73), (u'language', -0.73), (u'python', -0.56),
(u'java', -0.56), (u'popular', -0.34), (u'ruby', -0.33), (u'excellent',
-0.33), (u'program', -0.21)]

 In [875]: topics = get_topics_terms_weights(weights, feature_names)
     ...: print_topics_udf(topics=topics,
     ...:                  total_topics=total_topics,
     ...:                  weight_threshold=0.15,
     ...:                     display_weights=False)
Topic #1 without weights
[u'dog', u'fox', u'jump', u'smarter', u'cat', u'slow', u'lazy', u'quick',
u'clever']

Topic #2 without weights
[u'programming', u'language', u'python', u'java', u'popular', u'ruby',
u'excellent', u'program']
```

This gives us much better depiction of the topics, similar to the ones obtained earlier, where each topic clearly has distinguishable concepts from the other. Thus you can see how simple matrix computations helped us in implementing a powerful topic model framework! We define the following function as a generic reusable topic modeling framework using LSI:

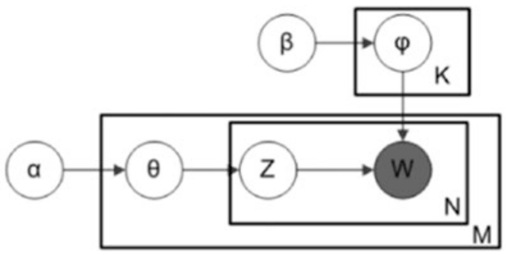
```
def train_lsi_model_gensim(corpus, total_topics=2):

    norm_tokenized_corpus = normalize_corpus(corpus, tokenize=True)
    dictionary = corpora.Dictionary(norm_tokenized_corpus)
    mapped_corpus = [dictionary.doc2bow(text)
                     for text in norm_tokenized_corpus]
    tfidf = models.TfidfModel(mapped_corpus)
    corpus_tfidf = tfidf[mapped_corpus]
    lsi = models.LsiModel(corpus_tfidf,
                          id2word=dictionary,
                          num_topics=total_topics)
    return lsi
```

We will use the preceding function later to extract topics from product reviews. Let us now look at the next technique to build topic models using latent Dirichlet allocation.

Latent Dirichlet Allocation

The latent Dirichlet allocation (LDA) technique is a generative probabilistic model where each document is assumed to have a combination of topics similar to a probabilistic latent semantic indexing model—but in this case, the latent topics contain a *Dirichlet prior* over them. The math behind in this technique is pretty involved, so I will try to summarize it because going it specific detail would be out of the current scope. I recommend readers to go through this excellent talk by Christine Doig available at http://chdoig.github.io/pygotham-topic-modeling/#/, from which we will be borrowing some excellent pictorial representations. The plate notation for the LDA model is depicted in Figure 5-2.



- K is the number of topics
- N is the number of words in the document
- M is the number of documents to analyse
- α is the Dirichlet-prior concentration parameter of the per-document topic distribution
- β is the same parameter of the per-topic word distribution
- φ(k) is the word distribution for topic k
- θ(i) is the topic distribution for document i
- z(i,j) is the topic assignment for w(i,j)
- w(i,j) is the j-th word in the i-th document
- φ and θ are Dirichlet distributions, z and w are multinomials.

Figure 5-2. *LDA plate notation (courtesy of C. Doig, Introduction to Topic Modeling in Python)*

Figure 5-3 shows a good representation of how each of the parameters connects back to the text documents and terms. It is assumed that we have *M* documents, *N* number of words in the documents, and *K* total number of topics we want to generate.

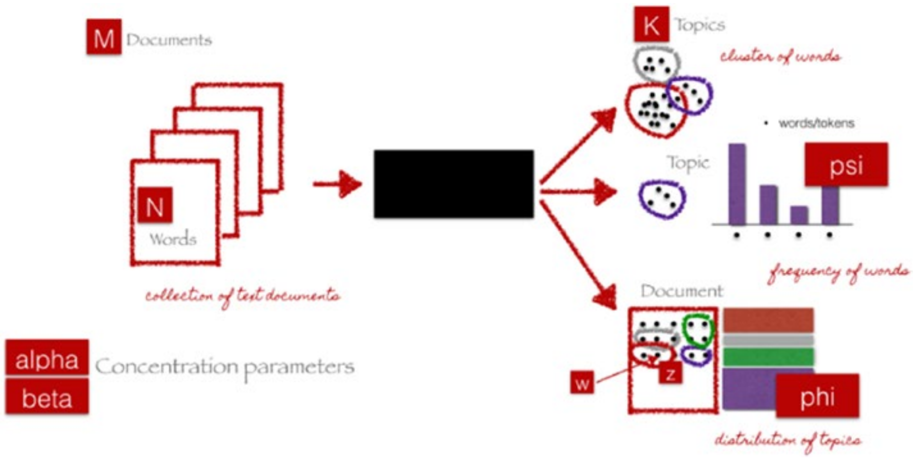


Figure 5-3. *End-to-end LDA framework (courtesy of C. Doig, Introduction to Topic Modeling in Python)*

The black box in the figure represents the core algorithm that makes use of the previously mentioned parameters to extract *K* topics from the documents. The following steps give a very simplistic explanation of what happens in the algorithm for everyone's benefit:

1. Initialize the necessary parameters.
2. For each document, randomly initialize each word to one of the *K* topics.
3. Start an iterative process as follows and repeat it several times.
4. For each document *D*:
    a. For each word *W* in document:
        - For each topic T:
            - Compute $P(T|D)$, which is proportion of words in *D* assigned to topic *T*.
            - Compute $P(W|T)$, which is proportion of assignments to topic *T* over all documents having the word *W*.
        - Reassign word *W* with topic *T* with probability $P(T|D) \times P(W|T)$ considering all other words and their topic assignments.

Once this runs for several iterations, we should have topic mixtures for each document and then generate the constituents of each topic from the terms that point to that topic. We use `gensim` in the following implementation to build an LDA-based topic model:

```
def train_lda_model_gensim(corpus, total_topics=2):

    norm_tokenized_corpus = normalize_corpus(corpus, tokenize=True)
    dictionary = corpora.Dictionary(norm_tokenized_corpus)
    mapped_corpus = [dictionary.doc2bow(text)
                     for text in norm_tokenized_corpus]
    tfidf = models.TfidfModel(mapped_corpus)
    corpus_tfidf = tfidf[mapped_corpus]
    lda = models.LdaModel(corpus_tfidf,
                          id2word=dictionary,
                          iterations=1000,
                          num_topics=total_topics)
    return lda

# use the function to generate topics on toy corpus
In [922]: lda_gensim = train_lda_model_gensim(toy_corpus,
     ...:                                     total_topics=2)
     ...:
     ...: print_topics_gensim(topic_model=lda_gensim,
     ...:                     total_topics=2,
     ...:                     num_terms=5,
     ...:                     display_weights=True)
Topic #1 with weights
[(u'fox', 0.08), (u'dog', 0.08), (u'jump', 0.07), (u'clever', 0.07),
(u'quick', 0.07)]

Topic #2 with weights
[(u'programming', 0.08), (u'language', 0.08), (u'java', 0.07), (u'python',
0.07), (u'ruby', 0.07)]
```

You can play around with various model parameters in the `LdaModel` class, which belongs to `gensim's ldamodel` module. This implementation works best with a corpus that has many documents. We see how the concepts are quite distinguishing across the two topics just as before, but note in this case the weights are positive, making it easier to interpret than LSI. Even `scikit-learn` has finally included an LDA-based topic model implementation in its library. The following snippet makes use of the same to build an LDA topic model:

```
from sklearn.decomposition import LatentDirichletAllocation

# get tf-idf based features
norm_corpus = normalize_corpus(toy_corpus)
vectorizer, tfidf_matrix = build_feature_matrix(norm_corpus,
                                    feature_type='tfidf')
# build LDA model
total_topics = 2
lda = LatentDirichletAllocation(n_topics=total_topics,
                                max_iter=100,
                                learning_method='online',
                                learning_offset=50.,
                                random_state=42)
lda.fit(tfidf_matrix)

# get terms and their weights
feature_names = vectorizer.get_feature_names()
weights = lda.components_

# generate topics from their terms and weights
topics = get_topics_terms_weights(weights, feature_names)
```

In that snippet, the LDA model is applied on the document-term TF-IDF feature matrix, which is decomposed into two matrices, namely a document-topic matrix and a topic-term matrix. We use the topic-term matrix stored in `lda.components_` to retrieve the weights for each term per topic. Once we have these weights, we use our `get_topics_terms_weights()` function from our LSI modeling to build the topics based on the terms and weights per topic. We can now view the topics using our `print_topics_udf()` function, which we implemented earlier:

```
In [926]: topics = get_topics_terms_weights(weights, feature_names)
     ...: print_topics_udf(topics=topics,
     ...:                  total_topics=total_topics,
     ...:                  num_terms=8,
     ...:                  display_weights=True)
Topic #1 with weights
[(u'fox', 1.86), (u'dog', 1.86), (u'jump', 1.19), (u'clever', 1.12),
(u'quick', 1.12), (u'lazy', 1.12), (u'slow', 1.12), (u'cat', 1.06)]

Topic #2 with weights
[(u'programming', 1.8), (u'language', 1.8), (u'java', 1.64), (u'python',
1.64), (u'program', 1.3), (u'ruby', 1.11), (u'excellent', 1.11),
(u'popular', 1.06)]
```

We can now see similar results for the two topics with distinguishable concepts where the first topic is about the animals and their characteristics from the first four documents and the second topic is all about programming languages and their attributes from the last four documents.

Non-negative Matrix Factorization

The last technique we will look at is non-negative matrix factorization (NNMF), which is another matrix decomposition technique similar to SVD, though NNMF operates on non-negative matrices and works well for multivariate data. NNMF can be formally defined like so: Given a non-negative matrix *V*, the objective is to find two non-negative matrix factors *W* and *H* such that when they are multiplied, they can approximately reconstruct *V*. Mathematically this is represented by

$$V \approx WH$$

such that all three matrices are non-negative. To get to this approximation, we usually use a cost function like the Euclidean distance or L2 norm between two matrices, or the Frobenius norm which is a slight modification of the L2 norm. This can be represented as

$$arg\,\min_{W,H} \frac{1}{2} \left\| V - WH \right\|^2$$

where we have our three non-negative matrices *V*, *W*, and *H*. This can be further simplified as follows:

$$\frac{1}{2} \sum_{i,j} \left(V_{ij} - WH_{ij} \right)^2$$

This implementation is available in the NMF class in the `scikit-learn` `decomposition` module that we will be using in the section.

We can build an NNMF-based topic model using the following snippet on our toy corpus which gives us the feature names and their weights just like in LDA:

```
from sklearn.decomposition import NMF
# build tf-idf document-term matrix
norm_corpus = normalize_corpus(toy_corpus)
vectorizer, tfidf_matrix = build_feature_matrix(norm_corpus,
                                    feature_type='tfidf')
# build topic model
total_topics = 2
nmf = NMF(n_components=total_topics,
          random_state=42, alpha=.1, l1_ratio=.5)
nmf.fit(tfidf_matrix)
# get terms and their weights
feature_names = vectorizer.get_feature_names()
weights = nmf.components_
```

Now that we have our terms and their weights, we can use our defined functions from before to print the topics as follows:

```
In [928]: topics = get_topics_terms_weights(weights, feature_names)
     ...: print_topics_udf(topics=topics,
     ...:                  total_topics=total_topics,
     ...:                  num_terms=None,
     ...:                  display_weights=True)
Topic #1 with weights
[(u'programming', 0.55), (u'language', 0.55), (u'python', 0.4), (u'java',
0.4), (u'popular', 0.24), (u'ruby', 0.23), (u'excellent', 0.23),
(u'program', 0.09), (u'small', 0.03)]

Topic #2 with weights
[(u'dog', 0.57), (u'fox', 0.57), (u'jump', 0.35), (u'smarter', 0.26),
(u'cat', 0.26), (u'quick', 0.13), (u'slow', 0.13), (u'clever', 0.13),
(u'lazy', 0.13)]
```

What we have observed is that non-negative matrix factorization works the best even with small corpora with few documents compared to the other methods, but again, this depends on the type of data you are dealing with.

Extracting Topics from Product Reviews

We will now utilize our earlier functions and build topic models using the three techniques on some real-world data. For this, I have extracted some reviews for a particular product from Amazon. Data enthusiasts can get more information about the source of this data from http://jmcauley.ucsd.edu/data/amazon/, which contains various product reviews based on product types and categories. The product of our interest is the very popular video game *The Elder Scrolls V: Skyrim* developed by Bethesda

Softworks. It is perhaps one of the best role-playing games out there. (You can view the product information and its reviews on Amazon at www.amazon.com/dp/B004HYK956 if you are interested.) In our case, the extracted reviews are available in a CSV file named amazon_skyrim_reviews.csv, available along with the code files of this chapter. Let us first load the reviews before extracting topics:

```
import pandas as pd
import numpy as np
# load reviews
CORPUS = pd.read_csv('amazon_skyrim_reviews.csv')
CORPUS = np.array(CORPUS['Reviews'])

# view sample review
In [946]: print CORPUS[12]
I base the value of a game on the amount of enjoyable gameplay I can get out
of it and this one was definitely worth the price!
```

Now that we have our corpus of product reviews loaded, let us set the number of topics to 5 and extract topics using all the three techniques implemented in the earlier sections. The following code snippet achieves the same:

```
# set number of topics
total_topics = 5

# Technique 1: Latent Semantic Indexing
In [958]: lsi_gensim = train_lsi_model_gensim(CORPUS,
     ...:                                      total_topics=total_topics)
     ...: print_topics_gensim(topic_model=lsi_gensim,
     ...:                     total_topics=total_topics,
     ...:                     num_terms=10,
     ...:                     display_weights=False)
Topic #1 without weights
[u'skyrim', u'one', u'quest', u'like', u'play', u'oblivion', u'go', u'get',
u'time', u'level']

Topic #2 without weights
[u'recommend', u'love', u'ever', u'best', u'great', u'level', u'highly',
u'play', u'elder', u'scroll']

Topic #3 without weights
[u'recommend', u'highly', u'fun', u'love', u'ever', u'wonderful', u'best',
u'everyone', u'series', u'scroll']

Topic #4 without weights
[u'fun', u'scroll', u'elder', u'recommend', u'highly', u'wonderful', u'fan',
u'graphic', u'series', u'cool']

Topic #5 without weights
```

```
[u'fun', u'love', u'elder', u'scroll', u'highly', u'5', u'dont', u'hour',
u'series', u'hundred']

# Technique 2a: Latent Dirichlet Allocation (gensim)
In [959]: lda_gensim = train_lda_model_gensim(CORPUS,
     ...:                                     total_topics=total_topics)
     ...: print_topics_gensim(topic_model=lda_gensim,
     ...:                     total_topics=total_topics,
     ...:                     num_terms=10,
     ...:                     display_weights=False)
Topic #1 without weights
[u'quest', u'good', u'skyrim', u'love', u'make', u'best', u'time', u'go',
u'play', u'every']

Topic #2 without weights
[u'good', u'play', u'get', u'really', u'like', u'one', u'hour', u'buy',
u'go', u'skyrim']

Topic #3 without weights
[u'fun', u'gameplay', u'skyrim', u'best', u'want', u'time', u'one', u'play',
u'review', u'like']

Topic #4 without weights
[u'love', u'play', u'one', u'much', u'great', u'ever', u'like', u'fun',
u'recommend', u'level']

Topic #5 without weights
[u'great', u'long', u'love', u'scroll', u'elder', u'oblivion', u'play',
u'month', u'never', u'skyrim']

# Technique 2b: Latent Dirichlet Allocation (scikit-learn)
In [960]: norm_corpus = normalize_corpus(CORPUS)
     ...: vectorizer, tfidf_matrix = build_feature_matrix(norm_corpus,
     ...:                                     feature_type='tfidf')
     ...: feature_names = vectorizer.get_feature_names()
     ...:
     ...:
     ...: lda = LatentDirichletAllocation(n_topics=total_topics,
     ...:                                 max_iter=100,
     ...:                                 learning_method='online',
     ...:                                 learning_offset=50.,
     ...:                                 random_state=42)
     ...: lda.fit(tfidf_matrix)
     ...: weights = lda.components_
     ...: topics = get_topics_terms_weights(weights, feature_names)
     ...: print_topics_udf(topics=topics,
     ...:                  total_topics=total_topics,
     ...:                  num_terms=10,
     ...:                  display_weights=False)
```

```
Topic #1 without weights
[u'statrs', u'expression', u'demand', u'unnecessary', u'mining', u'12yr',
u'able', u'snowy', u'shopkeepers', u'arpg']

Topic #2 without weights
[u'game', u'play', u'get', u'one', u'skyrim', u'great', u'like', u'time',
u'quest', u'much']

Topic #3 without weights
[u'de', u'pagar', u'cr\xe9dito', u'momento', u'responsabilidad', u'compras',
u'para', u'futuras', u'recomiendo', u'skyrimseguridad']

Topic #4 without weights
[u'booklet', u'proudly', u'ending', u'destiny', u'estatic', u'humungous',
u'chirstmas', u'bloodthey', u'accolade', u'scaled']

Topic #5 without weights
[u'game', u'play', u'fun', u'good', u'buy', u'one', u'whatnot', u'titles',
u'haveseen', u'best']

# Technique 3: Non-negative Matrix Factorization
In [961]: nmf = NMF(n_components=total_topics,
     ...:           random_state=42, alpha=.1, l1_ratio=.5)
     ...: nmf.fit(tfidf_matrix)
     ...:
     ...: feature_names = vectorizer.get_feature_names()
     ...: weights = nmf.components_
     ...:
     ...: topics = get_topics_terms_weights(weights, feature_names)
     ...: print_topics_udf(topics=topics,
     ...:                  total_topics=total_topics,
     ...:                  num_terms=10,
     ...:                  display_weights=False)
Topic #1 without weights
[u'game', u'get', u'skyrim', u'play', u'time', u'like', u'quest', u'one',
u'go', u'much']

Topic #2 without weights
[u'game', u'best', u'ever', u'fun', u'play', u'hour', u'great', u'rpg',
u'definitely', u'one']

Topic #3 without weights
[u'write', u'review', u'describe', u'justice', u'word', u'game', u'simply',
u'try', u'period', u'really']

Topic #4 without weights
[u'scroll', u'elder', u'series', u'always', u'love', u'pass', u'buy',
u'franchise', u'game', u'best']

Topic #5 without weights
```

```
[u'recommend', u'love', u'game', u'highly', u'great', u'play', u'wonderful',
u'like', u'oblivion', u'would']
```

The preceding outputs show five topics per technique. If you observe them closely, you will notice that there will always be some overlap between topics, but they bring out distinguishing concepts from the review. We can conclude a few observations:

- All topic modeling techniques bring out concepts related to people describing this game with adjectives like *wonderful*, *great*, and *highly recommendable*.
- They also describe the game's genre as RPG (role-playing game) or ARPG (action role-playing game).
- Game features like *gameplay* and *graphics* are associated with positive words like *good*, *great*, *fun*, and *cool*.
- The word *oblivion* comes up in many of the topic models. This is in reference to the previous game of the *Elder Scrolls* series, called *The Elder Scrolls IV: Oblivion*. This is an indication of customers comparing this game with its predecessor in the reviews.

Go ahead and play around with these functions and the data. You might even try building topic models on new data sources. Remember, topic modeling often acts as a starting point to digging deeper into the data to uncover patterns by querying with specific topic concepts or even clustering and grouping text documents and analyzing their similarity.

Automated Document Summarization

We briefly talked about document summarization at the beginning of this chapter, in trying to extract the gist from a large document or corpus such that it retains the core essence or meaning of the corpus. The idea of document summarization is a bit different from keyphrase extraction or topic modeling. The end result is still in the form of some document, but with a few sentences based on the length we might want the summary to be. This is similar to having a research paper with an abstract or an executive summary. The main objective of automated document summarization is to perform this summarization without involving human inputs except for running any computer programs. Mathematical and statistical models help in building and automating the task of summarizing documents by observing their content and context.

There are mainly two broad approaches towards document summarization using automated techniques:

- *Extraction-based techniques*: These methods use mathematical and statistical concepts like SVD to extract some key subset of content from the original document such that this subset of content contains the core information and acts as the focal point of the entire document. This content could be words, phrases, or sentences. The end result from this approach is a short executive summary of a couple of lines are taken or extracted from the original document. No new content is generated in this technique—hence the name *extraction-based*.
- *Abstraction-based techniques*: These methods are more complex and sophisticated and leverage language semantics to create representations. They also make use of NLG techniques where the machine uses knowledge bases and semantic representations to generate text on its own and creates summaries just like a human would write them.

Most research today exists for extraction-based techniques because it is comparatively harder to build abstraction-based summarizers. But some advances have been made in that area with regard to creating abstract summaries mimicking humans. Let us look at an implementation of document summarization by leveraging `gensim`'s summarization module. We will be using our Wikipedia description of elephants as the document on which we will test all our summarization techniques. We start by loading the necessary dependencies and the corpus as follows:

```
from normalization import normalize_corpus, parse_document
from utils import build_feature_matrix, low_rank_svd
import numpy as np

toy_text = """
Elephants are large mammals of the family Elephantidae
and the order Proboscidea. Two species are traditionally recognised,
the African elephant and the Asian elephant. Elephants are scattered
throughout sub-Saharan Africa, South Asia, and Southeast Asia. Male
African elephants are the largest extant terrestrial animals. All
elephants have a long trunk used for many purposes,
particularly breathing, lifting water and grasping objects. Their
incisors grow into tusks, which can serve as weapons and as tools
for moving objects and digging. Elephants' large ear flaps help
to control their body temperature. Their pillar-like legs can
carry their great weight. African elephants have larger ears
and concave backs while Asian elephants have smaller ears
and convex or level backs.
"""
```

We now define a function to summarize an input document to a fraction of its original size, which will be taken as a user input parameter `summary_ratio` in the following function. The output will be the summarized document:

```
from gensim.summarization import summarize, keywords

def text_summarization_gensim(text, summary_ratio=0.5):

    summary = summarize(text, split=True, ratio=summary_ratio)
    for sentence in summary:
        print sentence
```

We will now parse our input document to remove the newlines and extract sentences and then pass the complete document to the preceding function where `gensim` takes care of normalization and summarizes the document, as shown in the following snippet:

```
In [978]: docs = parse_document(toy_text)
     ...: text = ' '.join(docs)
     ...: text_summarization_gensim(text, summary_ratio=0.4)
Two species are traditionally recognised,  the African elephant and the
Asian elephant.
All  elephants have a long trunk used for many purposes,  particularly
breathing, lifting water and grasping objects.
African elephants have larger ears  and concave backs while Asian elephants
have smaller ears  and convex or level backs.
```

If you observe the preceding output and compare it with the original document, we had a total of nine sentences in the original document, and it has been summarize to a total of three sentences. But if you read the summarized document, you will see the core meaning and themes of the document have been retained, which include the two species of elephants, how they are distinguishable from each other, and their common characteristics. This summarization implementation from `gensim` is based on a popular algorithm called TextRank.

Now that we have seen how interesting text summarization can be, let us look at a couple of extraction-based summarization algorithms. We will be mainly focusing on the following two techniques:

- Latent semantic analysis
- TextRank

We will first explore the concepts and math behind each technique and then implement those using Python. Finally, we will test them on our toy document from before. Before we deep dive into the techniques, let us prepare our toy document by parsing and normalizing it as follows:

```
# parse and normalize document
sentences = parse_document(toy_text)
norm_sentences = normalize_corpus(sentences,lemmatize=True)
```

```
# check total sentences in document
In [992]: total_sentences = len(norm_sentences)
     ...: print 'Total Sentences in Document:', total_sentences
Total Sentences in Document: 9
```

Once we have a working summarization algorithm, we will also construct a generic function for each technique and test it on a real product description from Wikipedia in a future section.

Latent Semantic Analysis

Here, we will be looking at summarizing text documents by utilizing document sentences, the terms in each sentence of the document, and applying SVD to them using some sort of feature weights like Bag of Words or TF-IDF weights. The core principle behind latent semantic analysis (LSA) is that in any document, there exists a latent structure among terms which are related contextually and hence should also be correlated in the same singular space. The approach we follow in our implementation is taken from the popular paper published in 2004 by J. Steinberger and K. Jezek, "Using latent semantic analysis in text summarization and summary evaluation," which proposes some improvements over some excellent work done by Y. Gong and X. Liu's "Generic Text Summarization Using Relevance Measure and Latent Semantic Analysis," published in 2001. I recommend you to read these two papers if you are interested in learning more about this technique.

The main idea in our implementation is to use SVD such that, if you remember the equation from SVD where $M = USV^T$ such that *U* and *V* are the orthogonal matrices and S was the diagonal matrix, which can also be represented as a vector of the singular values. The original matrix can be represented as a term-document matrix, where the rows will be terms and each column will be a document, that is, a sentence from our document in this case. The values can be any type of weighting, like Bag of Words model-based frequencies, TF-IDFS, or binary occurrences. We will use our `low_rank_svd()` function to create a low rank matrix approximation for *M* based on the number of concepts *k*, which will be our number of singular values. The same *k* columns from matrix *U* will point to the term vectors for each of the *k* concepts, and in case of matrix *V*, the *k* rows based on the top *k* singular values point to sentence vectors. Once we have *U*, *S*, and V^T from the SVD for the top *k* singular values based on the number of concepts *k*, we perform the following computations. Remember, the input parameters we need are the number of concepts *k* and the number of sentences *n* which we want the final summary to contain:

- Get the sentence vectors from the matrix *V* (*k* rows).
- Get the top *k* singular values from *S*.
- Apply a threshold-based approach to remove singular values that are less than half of the largest singular value if any exist. This is a heuristic, and you can play around with this value if you want. Mathematically, $S_i = 0 \; iff \; S_i < \frac{1}{2}\max(S)$.
- Multiply each term sentence column from *V* squared with its corresponding singular value from *S* also squared, to get sentence weights per topic.

- Compute the sum of the sentence weights across the topics and take the square root of the final score to get the salience scores for each sentence in the document.

The preceding salience score computations for each sentence can be mathematically represented as

$$SS = \sqrt{\sum_{i=1}^{k} S_i V_i^T}$$

where *SS* denotes the saliency score for each sentence by taking the dot product between the singular values and the sentence vectors from V^T. Once we have these scores, we sort them in descending order, pick the top *n* sentences corresponding to the highest scores, and combine them to form our final summary based on the order in which they were present in the original document. Let us implement the above steps in our code using the following snippet:

```
# set the number of sentences and topics for summarized document
num_sentences = 3
num_topics = 3

# build document term matrix based on bag of words features
vec, dt_matrix = build_feature_matrix(sentences,
                                      feature_type='frequency')
# convert to term document matrix
td_matrix = dt_matrix.transpose()
td_matrix = td_matrix.multiply(td_matrix > 0)

# get low rank SVD components
u, s, vt = low_rank_svd(td_matrix, singular_count=num_topics)

# remove singular values below threshold
sv_threshold = 0.5
min_sigma_value = max(s) * sv_threshold
s[s < min_sigma_value] = 0

# compute salience scores for all sentences in document
salience_scores = np.sqrt(np.dot(np.square(s), np.square(vt)))

# print salience score for each sentence
In [996]: print np.round(salience_scores, 2)
[ 2.93  3.28  1.67  1.8   2.24  4.51  0.71  1.22  5.24]

# rank sentences based on their salience scores
top_sentence_indices = salience_scores.argsort()[-num_sentences:][::-1]
top_sentence_indices.sort()
```

```
# view top sentence index positions
In [997]: print top_sentence_indices
[1 5 8]

# get document summary by combining above sentences
In [998]: for index in top_sentence_indices:
     ...:     print sentences[index]
Two species are traditionally recognised,  the African elephant and the
Asian elephant.
Their  incisors grow into tusks, which can serve as weapons and as
tools  for moving objects and digging.
African elephants have larger ears  and concave backs while Asian elephants
have smaller ears  and convex or level backs.
```

You can see how a few matrix operations give us a concise and excellent summarized document that covers the main topics from the document about elephants. Compare it with the one generated earlier using `gensim`. Do you see some similarity between the summaries?

We will now build a generic reusable function for LSA using the previous algorithm so that we can use it on our product description document later on and you can also use this function on your own data:

```
def lsa_text_summarizer(documents, num_sentences=2,
                        num_topics=2, feature_type='frequency',
                        sv_threshold=0.5):

    vec, dt_matrix = build_feature_matrix(documents,
                                          feature_type=feature_type)

    td_matrix = dt_matrix.transpose()
    td_matrix = td_matrix.multiply(td_matrix > 0)

    u, s, vt = low_rank_svd(td_matrix, singular_count=num_topics)
    min_sigma_value = max(s) * sv_threshold
    s[s < min_sigma_value] = 0

    salience_scores = np.sqrt(np.dot(np.square(s), np.square(vt)))
    top_sentence_indices = salience_scores.argsort()[-num_sentences:][::-1]
    top_sentence_indices.sort()

    for index in top_sentence_indices:
        print sentences[index]
```

This concludes our discussion on LSA, and we will move on to the next technique for extraction-based document summarization.

TextRank

The TextRank summarization algorithm internally uses the popular PageRank algorithm, which is used by Google for ranking web sites and pages and measures their importance. It is used by the Google search engine when providing relevant web pages based on search queries. To understand TextRank better, we need to understand some of the concepts surrounding PageRank.

The core algorithm in PageRank is a graph-based scoring or ranking algorithm, where pages are scored or ranked based on their importance. Web sites and pages contain further links embedded in them, which link to more pages with more links, and this continues across the Internet. This can be represented as a graph-based model where vertices indicate the web pages, and edges indicate links among them. This can be used to form a voting or recommendation system such that when one vertex links to another one in the graph, it is basically casting a vote. Vertex importance is decided not only on the number of votes or edges but also the importance of the vertices that are connected to it and their importance. This helps in determining the score or rank for each vertex or page. This is evident from Figure 5-4, which represents a sample of pages with their importance.

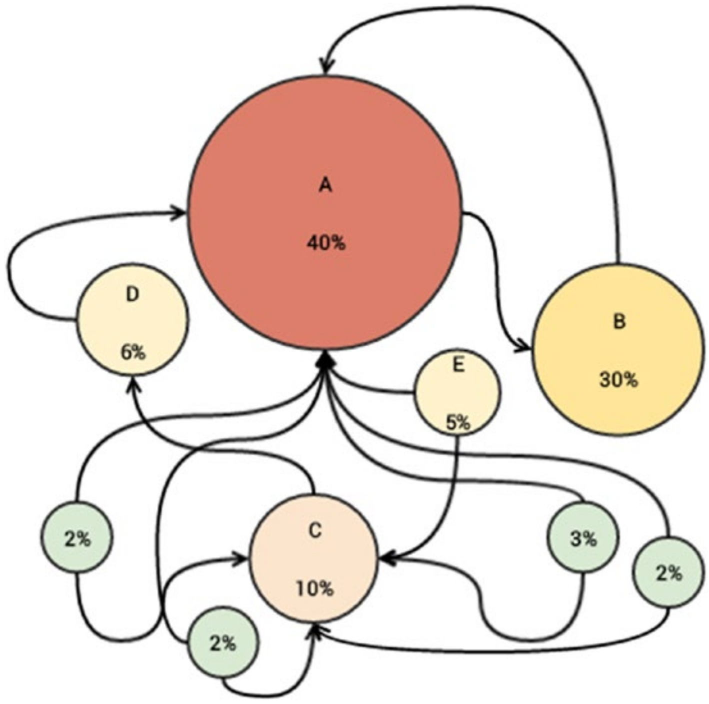


Figure 5-4. *PageRank scores for a simple network*

In Figure 5-4, we can see that vertex denoting Page B has a higher score than Page C, even if it has fewer edges compared to Page C, because Page A is an important page which is connected to Page B. Thus we can now formally define PageRank as follows.

Consider a directed graph represented as $G=(V,E)$ such that *V* represents the set of vertices or pages and *E* represents the set of edges or links, and *E* is a subset of $V \times V$. Assuming we have a given page V_i for which we want to compute the PageRank, we can mathematically define it as

$$PR(V_i)=(1-d)+d\times\sum_{j\in In(V_i)}\frac{PR(V_j)}{|Out(V_j|}$$

where for the vertex/page V_i we have $PR(V_i)$, which indicates the PageRank score, $In(V_i)$ represents the set of pages which point to this vertex/page, $Out(V_i)$ represents the set of pages which the vertex/page V_i points to, and *d* is the damping factor usually having a value between 0 to 1—ideally it is set to 0.85.

Coming back to the TextRank algorithm, when summarizing a document, we will have sentences, keywords, or phrases as the vertices of the algorithm based on the type of summarization we are trying to do. We might have multiple links between these vertices, and the modification which we make from the original PageRank algorithm is to have a weight coefficient say w_{ij} between the edge connecting two vertices V_i and V_j such that this weight indicates the strength of this connection between them. Thus we now formally define the new function for computing TextRank of vertices as

$$TR(V_i)=(1-d)+d\times\sum_{V_j\in In(V_i)}\frac{w_{ji}\,TR(V_j)}{\sum_{V_k\in Out(V_j)} w_{jk}}$$

where *TR* indicates the weighted PageRank score for a vertex now defined as the TextRank for that vertex. Thus we can now formulate the algorithm and identify the main steps we will be following:

1. Tokenize and extract sentences from the document to be summarized.
2. Decide on the number of sentences *k* that we want in the final summary.
3. Build document term feature matrix using weights like TF-IDF or Bag of Words.
4. Compute a document similarity matrix by multiplying the matrix with its transpose.
5. Use these documents (sentences in our case) as the vertices and the similarities between each pair of documents as the weight or score coefficient mentioned earlier and feed them to the PageRank algorithm.
6. Get the score for each sentence.
7. Rank the sentences based on score and return the top *k* sentences.

The following code snippet shows how to construct the connected graph among all the sentences from our toy document by making use of the document similarity scores and the documents themselves as the vertices. We will use the networkx library to help us plot this graph. Remember, each document is a sentence in our case which will also be the vertices in the graph:

```
import networkx

# define number of sentences in final summary
num_sentences = 3

# construct weighted document term matrix
vec, dt_matrix = build_feature_matrix(norm_sentences,
                                      feature_type='tfidf')

# construct the document similarity matrix
similarity_matrix = (dt_matrix * dt_matrix.T)
# view the document similarity matrix
In [1011]: print np.round(similarity_matrix.todense(), 2)
[[ 1.    0.    0.03  0.05  0.03  0.    0.15  0.    0.06]
 [ 0.    1.    0.    0.07  0.    0.    0.    0.    0.11]
 [ 0.03  0.    1.    0.03  0.02  0.    0.03  0.    0.04]
 [ 0.05  0.07  0.03  1.    0.03  0.    0.04  0.    0.11]
 [ 0.03  0.    0.02  0.03  1.    0.07  0.03  0.    0.04]
 [ 0.    0.    0.    0.    0.07  1.    0.    0.    0.  ]
 [ 0.15  0.    0.03  0.04  0.03  0.    1.    0.    0.05]
 [ 0.    0.    0.    0.    0.    0.    0.    1.    0.  ]
 [ 0.06  0.11  0.04  0.11  0.04  0.    0.05  0.    1.  ]]

# build the similarity graph
similarity_graph = networkx.from_scipy_sparse_matrix(similarity_matrix)
# view the similarity graph
In [1013]: networkx.draw_networkx(similarity_graph)
Out [1013]:
```

In Figure 5-5, we can see how the sentences of our toy document are now linked to each other based on document similarities. The graph gives an idea how well connected some sentences are to other sentences.

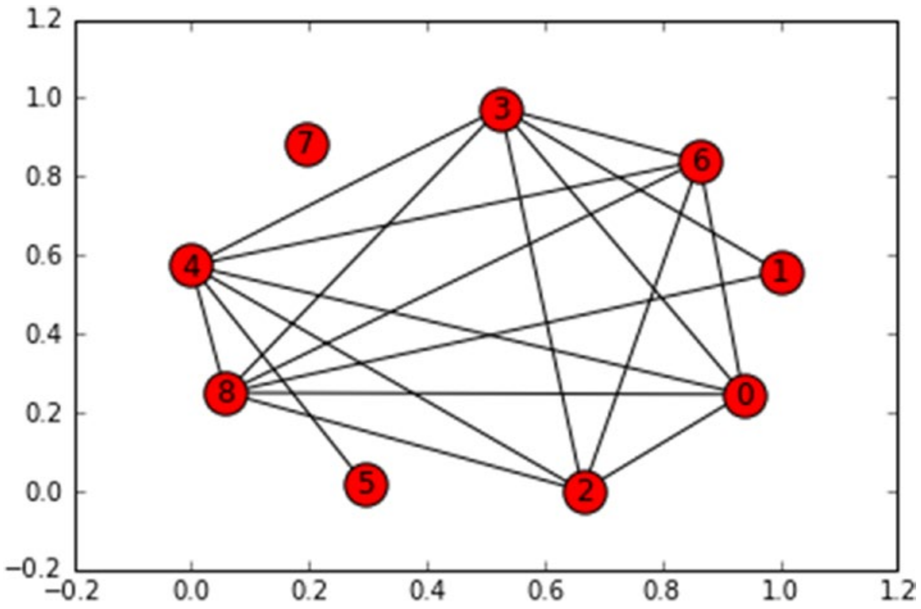


Figure 5-5. *Similarity graph showing connections between sentences*

We will now compute the PageRank scores for all the sentences, rank them, and build our summary using the top three sentences:

```
# compute pagerank scores for all the sentences
scores = networkx.pagerank(similarity_graph)

# rank sentences based on their scores
ranked_sentences = sorted(((score, index)
                            for index, score
                            in scores.items()),
                          reverse=True)
# view the ranked sentences
In [1030]: ranked_sentences
Out[1030]:
```

```
[(0.11889477617125277, 8),
 (0.11456045476451866, 3),
 (0.11285293843138654, 0),
 (0.11210156056437962, 6),
 (0.11139550507847462, 4),
 (0.111111111111111, 7),
 (0.10709498606197024, 5),
 (0.10610242758495998, 2),
 (0.10588624023194664, 1)]

# get the top sentence indices for our summary
top_sentence_indices = [ranked_sentences[index][1]
                        for index in range(num_sentences)]
top_sentence_indices.sort()

# view the top sentence indices
In [1032]: print top_sentence_indices
 [0, 3, 8]

# construct the document summary
In [1033]: for index in top_sentence_indices:
      ...:     print sentences[index]
Elephants are large mammals of the family Elephantidae  and the order
Proboscidea.
Male  African elephants are the largest extant terrestrial animals.
African elephants have larger ears  and concave backs while Asian elephants
have smaller ears  and convex or level backs.
```

We finally get our desired summary by using the TextRank algorithm. The content is also quite meaningful where it talks about elephants being mammals, their taxonomy, and how Asian and African elephants can be distinguished.

We will now define a generic function as follows to compute TextRank-based summaries on any document:

```
def textrank_text_summarizer(documents, num_sentences=2,
                             feature_type='frequency'):

    vec, dt_matrix = build_feature_matrix(norm_sentences,
                                          feature_type='tfidf')
    similarity_matrix = (dt_matrix * dt_matrix.T)

    similarity_graph = networkx.from_scipy_sparse_matrix(similarity_matrix)
    scores = networkx.pagerank(similarity_graph)

    ranked_sentences = sorted(((score, index)
                                for index, score
                                in scores.items()),
                              reverse=True)
    top_sentence_indices = [ranked_sentences[index][1]
                            for index in range(num_sentences)]
    top_sentence_indices.sort()

    for index in top_sentence_indices:
        print sentences[index]
```

We have covered two document-summarization techniques and also built generic reusable functions to compute automated document summaries for any text document. In the following section, we will summarize a product description from a wiki page.

Summarizing a Product Description

Building on what we talked about in the product reviews from the topic modeling section, here we will be summarizing a description for the same product—a role-playing video game named *The Elder Scrolls V: Skyrim*. We have taken several lines from the Wikipedia page containing the product's detailed description. In this section, we will perform automated document summarization on the product description utilizing our functions from the previous section. We will start with loading the product description and normalizing the content:

```
# load the document
DOCUMENT = """
The Elder Scrolls V: Skyrim is an open world action role-playing video game
developed by Bethesda Game Studios and published by Bethesda Softworks.
It is the fifth installment in The Elder Scrolls series, following
The Elder Scrolls IV: Oblivion. Skyrim's main story revolves around
the player character and their effort to defeat Alduin the World-Eater,
a dragon who is prophesied to destroy the world.
The game is set two hundred years after the events of Oblivion
and takes place in the fictional province of Skyrim. The player completes
quests
and develops the character by improving skills.
Skyrim continues the open world tradition of its predecessors by allowing the
player to travel anywhere in the game world at any time, and to
ignore or postpone the main storyline indefinitely. The player may freely roam
over the land of Skyrim, which is an open world environment consisting
of wilderness expanses, dungeons, cities, towns, fortresses and villages.
Players may navigate the game world more quickly by riding horses,
or by utilizing a fast-travel system which allows them to warp to previously
Players have the option to develop their character. At the beginning of the game,
players create their character by selecting one of several races,
including humans, orcs, elves and anthropomorphic cat or lizard-like
creatures,
and then customizing their character's appearance.discovered locations. Over the
course of the game, players improve their character's skills, which are
numerical
representations of their ability in certain areas. There are eighteen skills
divided evenly among the three schools of combat, magic, and stealth.
Skyrim is the first entry in The Elder Scrolls to include Dragons in the game's
wilderness. Like other creatures, Dragons are generated randomly in the world
and will engage in combat.
"""

# normalize the document
In [1045]: sentences = parse_document(DOCUMENT)
      ...: norm_sentences = normalize_corpus(sentences,lemmatize=True)
      ...: print "Total Sentences:", len(norm_sentences)
Total Sentences: 13
```

We can see that there are a total of 13 sentences in this description. Let us now generate the document summaries using our functions in the following code snippet:

```
# LSA document summarization
In [1053]: lsa_text_summarizer(norm_sentences, num_sentences=3,
      ...:                     num_topics=5, feature_type='frequency',
      ...:                     sv_threshold=0.5)
The Elder Scrolls V: Skyrim is an open world action role-playing video
game  developed by Bethesda Game Studios and published by Bethesda
Softworks.
Players may navigate the game world more quickly by riding horses,  or
by utilizing a fast-travel system which allows them to warp to
previously  Players have the option to develop their character.
At the beginning of the game,  players create their character by selecting
one of several races,  including humans, orcs, elves and anthropomorphic
cat or lizard-like creatures,  and then customizing their character's
appearance.discovered locations.

# TextRank document summarization
In [1054]: textrank_text_summarizer(norm_sentences, num_sentences=3,
      ...:                          feature_type='tfidf')
The Elder Scrolls V: Skyrim is an open world action role-playing video
game  developed by Bethesda Game Studios and published by Bethesda
Softworks.
Players may navigate the game world more quickly by riding horses,  or
by utilizing a fast-travel system which allows them to warp to
previously  Players have the option to develop their character.
Skyrim is the first entry in The Elder Scrolls to include Dragons in the
game's  wilderness.
```

You can see from the preceding outputs that we were successfully able to summarize our product description from 13 to 3 lines, and this short summary depicts the core essence of the product description, like the name of the game and its various features regarding its gameplay and characters.

This concludes our discussion on automated text summarization. I encourage you to try out these techniques on more documents and test it with various different parameters like more number of topics, different feature types like TF-IDF, Bag of Words, binary occurrences, and even word vectors.

Summary

In this chapter, we covered some interesting areas in NLP and text analytics with regard to information extraction, document summarization, and topic modeling. We started with an overview of the evolution of information and learned about concepts like information overload leading to the need for text summarization and information retrieval. We talked about the various ways we can extract key information from textual data and ways of summarizing large documents. We covered important mathematical concepts like SVD and low rank matrix approximation and utilized them in several of our algorithms. We mainly covered three approaches towards reducing information overload, including keyphrase extraction, topic models, and automated document summarization. Keyphrase extraction includes methods like collocations and weighted tagged term–based approaches for getting keyphrases or terms from corpora. We built several topic modeling techniques, including latent semantic indexing, latent Dirichlet allocation, and the very recently implemented non-negative matrix factorization. Finally, we looked at two extraction-based techniques for automated document summarization: LSA and TextRank. We implemented each method and observed results on real-world data to get a good idea of how these methods worked and how effective simple mathematical operations can be in generating actionable insights.