15
Word Embeddings

In the two previous chapters, we applied the bag-of-words model to convert text data into a numerical format. The results were sparse, fixed-length vectors that represent documents in a high-dimensional word space. This allows evaluating the similarity of documents and creates features to train a machine learning algorithm and classify a document's content or rate the sentiment expressed in it. However, these vectors ignore the context in which a term is used so that, for example, a different sentence containing the same words would be encoded by the same vector.

In this chapter, we will introduce an alternative class of algorithms that use neural networks to learn a vector representation of individual semantic units such as a word or a paragraph. These vectors are dense rather than sparse, and have a few hundred real-valued rather than tens of thousands of binary or discrete entries. They are called **embeddings** because they assign each semantic unit a location in a continuous vector space.

Embeddings result from training a model to relate tokens to their context with the benefit that similar usage implies a similar vector. Moreover, we will see how the embeddings encode semantic aspects, such as relationships among words by means of their relative location. As a result, they are powerful features for use in the deep learning models that we will introduce in the following chapters.

More specifically, in this chapter, we will cover the following topics:

- What word embeddings are and how they work and capture semantic information
- How to use trained word vectors
- Which network architectures are useful to train Word2vec models
- How to train a Word2vec model using Keras, gensim, and TensorFlow
- How to visualize and evaluate the quality of word vectors
- How to train a Word2vec model using SEC filings
- How `Doc2vec` extends Word2vec

How word embeddings encode semantics

The bag-of-words model represents documents as vectors that reflect the tokens they contain. Word embeddings represent tokens as lower dimensional vectors so that their relative location reflects their relationship in terms of how they are used in context. They embody the distributional hypothesis from linguistics that claims words are best defined by the company they keep.

Word vectors are capable of capturing numerous semantic aspects; not only are synonyms close to each other, but words can have multiple degrees of similarity, for example, the word driver could be similar to motorist or to cause. Furthermore, embeddings reflect relationships among pairs of words such as analogies (Tokyo is to Japan what Paris is to France, or went is to go what saw is to see) as we will illustrate later in this section.

Embeddings result from training a machine learning model to predict words from their context or vice versa. In the following section, we will introduce how these neural language models work and present successful approaches including Word2vec, `Doc2vec`, and fastText.

How neural language models learn usage in context

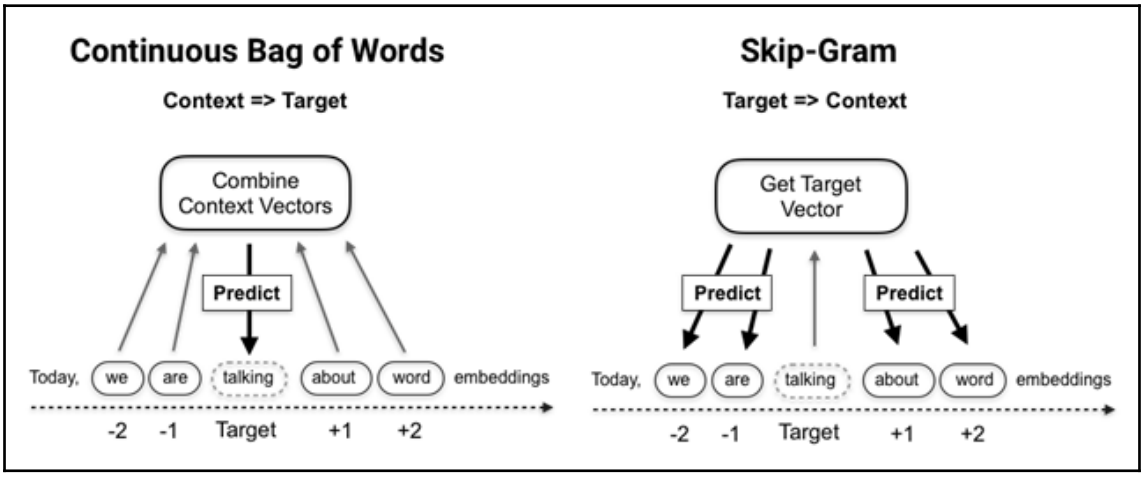
Word embeddings result from training a shallow neural network to predict a word given its context. Whereas traditional language models define context as the words preceding the target, word-embedding models use the words contained in a symmetric window surrounding the target. In contrast, the bag-of-words model uses the entirety of documents as context and uses (weighted) counts to capture the cooccurrence of words rather than predictive vectors.

Earlier neural language models that were used included nonlinear hidden layers that increased the computational complexity. Word2vec and its extensions simplified the architecture to enable training on large datasets (Wikipedia, for example, contains over two billion tokens; see `Chapter 17`, *Deep Learning*, for additional details on feed-forward networks).

The Word2vec model – learn embeddings at scale

A Word2vec model is a two-layer neural net that takes a text corpus as input and outputs a set of embedding vectors for words in that corpus. There are two different architectures to learn word vectors efficiently using shallow neural networks depicted in the following figure:

- The **Continuous-Bag-Of-Words** (**CBOW**) model predicts the target word using the average of the context word vectors as input so that their order does not matter. A CBOW model trains faster and tends to be slightly more accurate for frequent terms, but pays less attention to infrequent words.
- The **Skip-Gram** (**SG**) model, by contrast, uses the target word to predict words sampled from the context. It works well with small datasets and finds good representations even for rare words or phrases:



Hence, the Word2vec model receives an embedding vector as input and computes the dot product with another embedding vector. Note that, assuming normed vectors, the dot product is maximized (in absolute terms) when vectors are equal, and minimized when they are orthogonal.

It then uses backpropagation to adjust the embedding weights in response to the loss computed by an objective function due to any classification errors. We will see in the next section how Word2vec computes the loss.

Training proceeds by sliding the context window over the documents, typically segmented into sentences. Each complete iteration over the corpus is called an **epoch**. Depending on the data, several dozen epochs may be necessary for vector quality to converge.

Technically, the SG model has been shown to factorize a word-context matrix that contains the pointwise mutual information of the respective word and context pairs implicitly (see references on GitHub).

Model objective – simplifying the softmax

Word2vec models aim to predict a single word out of the potentially very large vocabulary. Neural networks often use the softmax function that maps any number of real values to an equal number of probabilities to implement the corresponding multiclass objective, where *h* refers to the embedding and *v* to the input vectors, and *c* is the context of word *w*:

$$p(\ w \mid c) = \frac{exp(h^T v'_w)}{\sum_{w_i \epsilon V} exp(h^T v'_{w_i})}$$

However, the softmax complexity scales with the number of classes, as the denominator requires the computation of the dot product for all words in the vocabulary to standardize the probabilities. Word2vec models gain efficiency by using a simplified version of the softmax or sampling-based approaches (see references for details):

- The **hierarchical softmax** organizes the vocabulary as a binary tree with words as leaf nodes. The unique path to each node can be used to compute the word probability.
- **Noise-contrastive estimation** (**NCE**) samples out-of-context "noise words" and approximates the multiclass task by a binary classification problem. The NCE derivative approaches the softmax gradient as the number of samples increases, but as few as 25 samples can yield convergence similar to the softmax, at a rate that is 45 times faster.
- **Negative sampling** (**NEG**) omits the noise word samples to approximate NCE and directly maximizes the probability of the target word. Hence, NEG optimizes the semantic quality of embedding vectors (similar vectors for similar usage) rather than the accuracy on a test set. It may, however, produce poorer representations for infrequent words than the hierarchical softmax objective.

Automatic phrase detection

Preprocessing typically involves phrase detection, that is, the identification of tokens that are commonly used together and should receive a single vector representation (for example, New York City, see the discussion of n-grams in `Chapter 13`, *Working with Text Data*).

The original Word2vec authors use a simple lift scoring method that identifies two words w_i, w_j as a bigram if their joint occurrence exceeds a given threshold relative to each word's individual appearance, corrected by a discount factor δ:

$$score(w_i, w_j) = \frac{count(w_i, w_j) - \delta}{count(w_i)count(w_j)}$$

The scorer can be applied repeatedly to identify successively longer phrases.

An alternative is the normalized point-wise mutual information score that is more accurate, but also more costly to compute. It uses the relative word frequency *P(w)* and varies between +1 and -1:

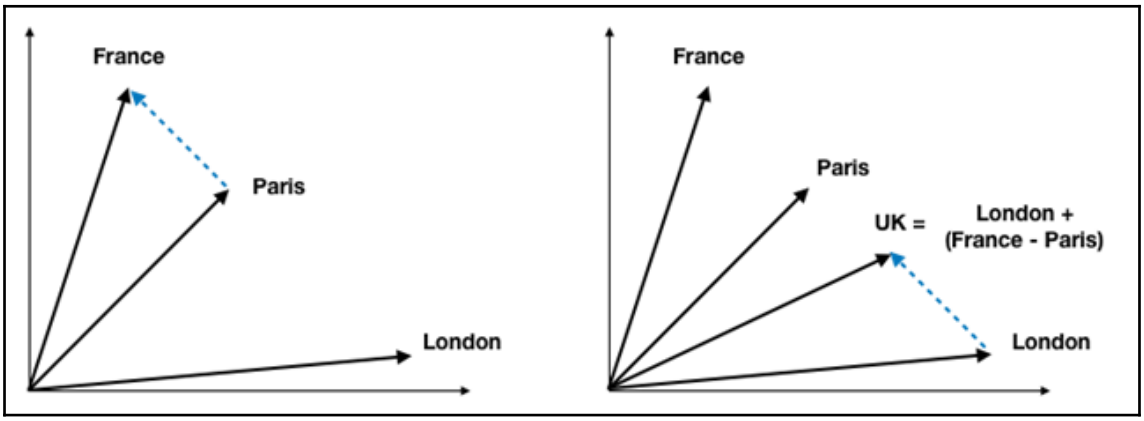
$$NPMI = \frac{ln(P(w_i, w_j)/P(w_i)P(w_j))}{-ln(P(w_i, w_j))}$$

How to evaluate embeddings – vector arithmetic and analogies

The bag-of-words model creates document vectors that reflect the presence and relevance of tokens to the document. **Latent semantic analysis** reduces the dimensionality of these vectors and identifies what can be interpreted as latent concepts in the process. **Latent Dirichlet allocation** represents both documents and terms as vectors that contain the weights of latent topics.

The dimensions of the word and phrase vectors do not have an explicit meaning. However, the embeddings encode similar usage as proximity in the latent space in a way that carries over to semantic relationships. This results in the interesting properties that analogies can be expressed by adding and subtracting word vectors.

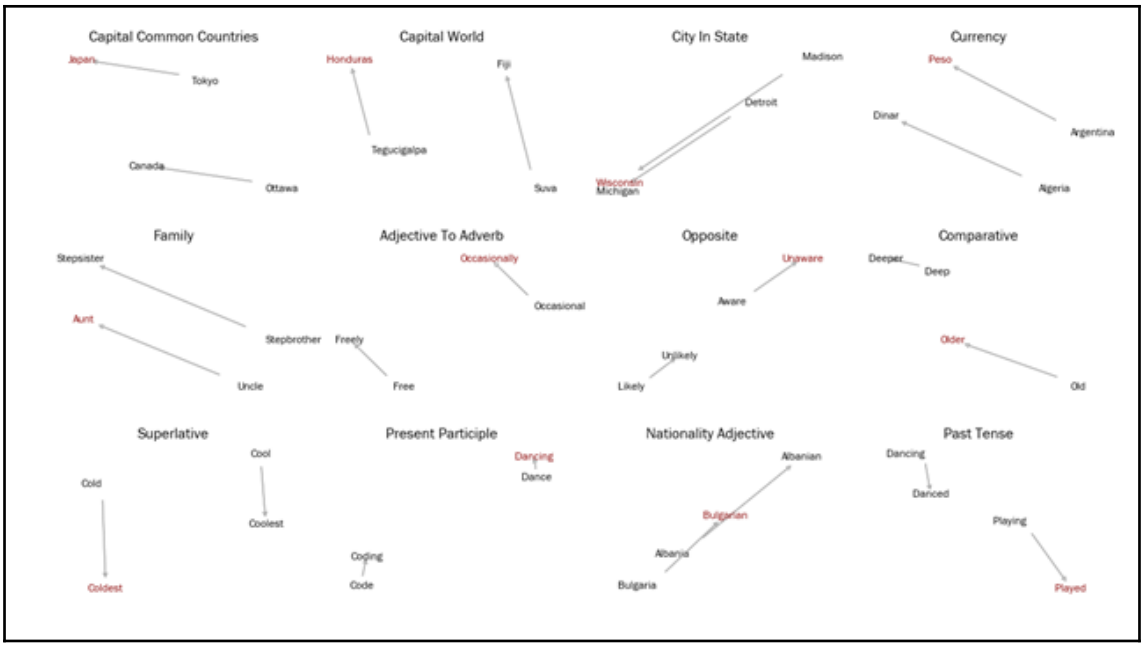
The following figure shows how the vector connecting Paris and France (that is, the difference of their embeddings) reflects the capital of relationship. The analogous relationship, London: UK, corresponds to the same vector, that is, the UK is very close to the location obtained by adding the capital of vector to London:



Just as words can be used in different contexts, they can be related to other words in different ways, and these relationships correspond to different directions in the latent space. Accordingly, there are several types of analogies that the embeddings should reflect if the training data permits.

The Word2vec authors provide a list of several thousand relationships spanning aspects of geography, grammar and syntax, and family relationships to evaluate the quality of embedding vectors. As illustrated above, the test validates that the target word (UK) is closest to the result of adding the vector that represents an analogous relationship (Paris: France) to the target's complement (London).

The following figure projects the 300-dimensional embeddings of the most closely related analogies for a Word2vec model trained on the Wikipedia corpus, with over 2 billion tokens, into two dimensions using **principal component analysis** (**PCA**). A test of over 24,400 analogies from the following categories achieved an accuracy of over 73.5% (see notebook):



Working with embedding models

Similar to other unsupervised learning techniques, the goal of learning embedding vectors is to generate features for other tasks such as text classification or sentiment analysis.

There are several options to obtain embedding vectors for a given corpus of documents:

- Use embeddings learned from a generic large corpus such as Wikipedia or Google News
- Train your own model using documents that reflect a domain of interest

The less generic and more specialized the content of the subsequent text modeling task is, the more preferable is the second approach. However, quality word embeddings are data-hungry and require informative documents containing hundreds of millions of words.

How to use pre-trained word vectors

There are several sources for pretrained word embeddings. Popular options include Stanford's GloVE and spaCy's built-in vectors (see the notebook `using_trained_vectors` for details).

GloVe – global vectors for word representation

GloVe is an unsupervised algorithm developed at the Stanford NLP lab that learns vector representations for words from aggregated global word-word co-occurrence statistics (see references). Vectors pretrained on the following web-scale sources are available:

- Common Crawl with 42B or 840B tokens and a vocabulary or 1.9M or 2.2M tokens
- Wikipedia 2014 + Gigaword 5 with 6B tokens and a vocabulary of 400K tokens
- Twitter using 2B tweets, 27B tokens and a vocabulary of 1.2M tokens

We can use gensim to convert and load the vector text files into the `KeyedVector` object:

```
from gensim.models import Word2vec, KeyedVectors
 from gensim.scripts.glove2Word2vec import glove2Word2vec
glove2Word2vec(glove_input_file=glove_file, Word2vec_output_file=w2v_file)
 model = KeyedVectors.load_Word2vec_format(w2v_file, binary=False)
```

The Word2vec authors provide text files containing over 24,000 analogy tests that gensim uses to evaluate word vectors.

The word vectors trained on the Wikipedia corpus cover all analogies and achieve an overall accuracy of 75.5% with some variation across categories:

Category	Samples	Accuracy	Category	Samples	Accuracy
capital-common-countries	506	94.86%	comparative	1,332	88.21%
capital-world	8,372	96.46%	superlative	1,056	74.62%
city-in-state	4,242	60.00%	present-participle	1,056	69.98%
currency	752	17.42%	nationality-adjective	1,640	92.50%
family	506	88.14%	past-tense	1,560	61.15%
adjective-to-adverb	992	22.58%	plural	1,332	78.08%
opposite	756	28.57%	plural-verbs	870	58.51%

The Common Crawl vectors for the 100,000 most common tokens cover about 80% of the analogies and achieve slightly higher accuracy at 78%, whereas the Twitter vectors cover only 25% with 62% accuracy.

How to train your own word vector embeddings

Many tasks require embeddings or domain-specific vocabulary that pretrained models based on a generic corpus may not represent well or at all. Standard Word2vec models are not able to assign vectors to out-of-vocabulary words and instead use a default vector that reduces their predictive value.

For example, when working with industry-specific documents, the vocabulary or its usage may change over time as new technologies or products emerge. As a result, the embeddings need to evolve as well. In addition, corporate earnings releases use nuanced language not fully reflected in GloVe vectors pretrained on Wikipedia articles.

We will illustrate the Word2vec architecture using the Keras library that we will introduce in more detail in the next chapter and the more performant gensim adaptation of the code provided by the Word2vec authors. The notebook Word2vec contains additional implementation detail, including a reference of a TensorFlow implementation.

The Skip-Gram architecture in Keras

To illustrate the Word2vec network architecture, we use the TED Talk dataset with aligned English and Spanish subtitles that we first introduced in `Chapter 13`, *Working with Text Data*.

The notebook contains the code to tokenize the documents and assign a unique ID to each item in the vocabulary. We require at least five occurrences in the corpus and keep a vocabulary of 31,300 tokens.

Noise-contrastive estimation

Keras includes a `make_sampling_table` method that allows us to create a training set as pairs of context and noise words with corresponding labels, sampled according to their corpus frequencies.

The result is 27 million positive and negative examples of context and target pairs.

The model components

The *Skip-Gram* model contains a 200-dimensional embedding vector for each vocabulary item, resulting in 31,300 x 200 trainable parameters, plus two for the sigmoid output.

In each iteration, the model computes the dot product of the context and the target-embedding vectors, passes the result through the sigmoid to produce a probability and adjusts the embedding based on the gradient of the loss.

Visualizing embeddings using TensorBoard

TensorBoard is a visualization tool that permits the projection of the embedding vectors into three dimensions to explore the word and phrase locations.

Word vectors from SEC filings using gensim

In this section, we will learn word and phrase vectors from annual US **Securities and Exchange Commission** (**SEC**) filings using gensim to illustrate the potential value of word embeddings for algorithmic trading. In the following sections, we will combine these vectors as features with price returns to train neural networks to predict equity prices from the content of security filings.

In particular, we use a dataset containing over 22,000 10-K annual reports from the period 2013-2016 that are filed by listed companies and contain both financial information and management commentary (see `Chapter 3`, *Alternative Data for Finance*). For about half of the 11-K filings for companies, we have stock prices to label the data for predictive modeling (see references about data sources and the notebooks in the `sec-filings` folder for details).

Preprocessing

Each filing is a separate text file and a master index contains filing metadata. We extract the most informative sections, namely, the following:

- **Items 1 and 1A**: Business and Risk Factors
- **Items 7 and 7A**: Management's Discussion and Disclosures about Market Risks

The notebook preprocessing shows how to parse and tokenize the text using spaCy, similar to the approach taken in `Chapter 14`, *Topic Modeling*. We do not lemmatize the tokens to preserve the nuances of word usage.

Automatic phrase detection

We use `gensim` to detect phrases as previously introduced. The `Phrases` module scores the tokens and the `Phraser` class transforms the text data accordingly. The notebook shows how to repeat the process to create longer phrases:

```
sentences = LineSentence(f'ngrams_1.txt')
phrases = Phrases(sentences=sentences,
                  min_count=25,  # ignore terms with a lower count
                  threshold=0.5,  # only phrases with higher score
                  delimiter=b'_',  # how to join ngram tokens
                  scoring='npmi')  # alternative: default
grams = Phraser(phrases)
sentences = grams[sentences]
```

The most frequent bigrams include `common_stock`, `united_states`, `cash_flows`, `real_estate`, and `interest_rates`.

Model training

The `gensim.models.Word2vec` class implements the SG and CBOW architectures introduced previously. The Word2vec notebook contains additional implementation detail.

To facilitate memory-efficient text ingestion, the `LineSentence` class creates a generator from individual sentences contained in the provided text file:

```
sentence_path = Path('data', 'ngrams', f'ngrams_2.txt')
sentences = LineSentence(sentence_path)
```

The `Word2vec` class offers the configuration options previously introduced:

```
model = Word2vec(sentences,
                 sg=1,     # 1=skip-gram; otherwise CBOW
                 hs=0,     # hier. softmax if 1, neg. sampling if 0
                 size=300,       # Vector dimensionality
                 window=3,       # Max dist. btw target and context word
                 min_count=50,  # Ignore words with lower frequency
                 negative=10,  # noise word count for negative sampling
                 workers=8,      # no threads
                 iter=1,         # no epochs = iterations over corpus
                 alpha=0.025,    # initial learning rate
                 min_alpha=0.0001 # final learning rate
                )
```

The notebook shows how to persist and reload models to continue training, or how to store the embedding vectors separately, for example, for use in ML models.

Model evaluation

Basic functionality includes identifying similar words:

```
model.wv.most_similar(positive=['iphone'],
                      restrict_vocab=15000)
                  term  similarity
0              android    0.600454
1           smartphone    0.581685
2                  app    0.559129
```

We can also validate individual analogies using positive and negative contributions accordingly:

```
model.wv.most_similar(positive=['france', 'london'],
                      negative=['paris'],
                      restrict_vocab=15000)

             term  similarity
0  united_kingdom    0.606630
1         germany    0.585644
2     netherlands    0.578868
```

Performance impact of parameter settings

We can use the analogies to evaluate the impact of different parameter settings. The following results stand out (see detailed results in the `models` folder):

- Negative sampling outperforms the hierarchical softmax, while also training faster
- The Skip-Gram architecture outperforms CBOW given the objective function
- Different `min_count` settings have a smaller impact, with the midpoint of 50 performing best

Further experiments with the best performing SG model, using negative sampling and a `min_count` of 50, show the following:

- Smaller context windows than five lower the performance
- A higher negative sampling rate improves performance at the expense of slower training
- Larger vectors improve performance, with a size of 600 yielding the best accuracy at 38.5%

Sentiment analysis with Doc2vec

Text classification requires combining multiple word embeddings. A common approach is to average the embedding vectors for each word in the document. This uses information from all embeddings and effectively uses vector addition to arrive at a different location point in the embedding space. However, relevant information about the order of words is lost.

By contrast, the state-of-the-art generation of embeddings for pieces of text such as a paragraph or a product review is to use the document-embedding model `Doc2vec`. This model was developed by the Word2vec authors shortly after publishing their original contribution.

Similar to Word2vec, there are also two flavors of `Doc2vec`:

- The **distributed bag of words** (**DBOW**) model corresponds to the Word2vec CBOW model. The document vectors result from training a network in the synthetic task of predicting a target word based on both the context word vectors and the document's doc vector.
- The **distributed memory** (**DM**) model corresponds to the Word2vec Skip-Gram architecture. The doc vectors result from training a neural net to predict a target word using the full document's doc vector.

Gensim's `Doc2vec` class implements this algorithm.

Training Doc2vec on yelp sentiment data

We use a random sample of 500,000 Yelp (see `Chapter 13`, *Working with Text Data*) reviews with their associated star ratings (see notebook `yelp_sentiment`):

```
df = (pd.read_parquet('yelp_reviews.parquet', engine='fastparquet')
          .loc[:, ['stars', 'text']])
stars = range(1, 6)
sample = pd.concat([df[df.stars==s].sample(n=100000) for s in stars])
```

We apply use simple pre-processing to remove stopwords and punctuation using `NLTK`'s tokenizer and drop reviews with fewer than 10 tokens:

```
import nltk
nltk.download('stopwords')
from nltk import RegexpTokenizer
from nltk.corpus import stopwords
tokenizer = RegexpTokenizer(r'\w+')
stopword_set = set(stopwords.words('english'))

def clean(review):
    tokens = tokenizer.tokenize(review)
    return ' '.join([t for t in tokens if t not in stopword_set])

sample.text = sample.text.str.lower().apply(clean)
sample = sample[sample.text.str.split().str.len()>10]
```

Create input data

The `gensim.models.doc2vec` class processes documents in the `TaggedDocument` format that contains the tokenized documents alongside a unique tag that permits accessing the document vectors after training:

```
sentences = []
for i, (_, text) in enumerate(sample.values):
    sentences.append(TaggedDocument(words=text.split(), tags=[i]))
```

The training interface works similar to `word2vec` with additional parameters to specify the Doc2vec algorithm:

```
model = Doc2vec(documents=sentences,
                dm=1,          # algorithm: use distributed memory
                dm_concat=0,   # 1: concat, not sum/avg context vectors
                dbow_words=0,  # 1: train word vectors, 0: only doc
                                    vectors
                alpha=0.025,   # initial learning rate
```

```
                  size=300,
                  window=5,
                  min_count=10,
                  epochs=5,
                  negative=5)
model.save('test.model')
```

You can also use the `train()` method to continue the learning process and, for example, iteratively reduce the learning rate:

```
for _ in range(10):
    alpha *= .9
    model.train(sentences,
                total_examples=model.corpus_count,
                epochs=model.epochs,
                alpha=alpha)
```

As a result, we can access the document vectors as features to train a sentiment classifier:

```
X = np.zeros(shape=(len(sample), size))
y = sample.stars.sub(1) # model needs [0, 5) labels
for i in range(len(sample)):
 X[i] = model[i]
```

We will train a `lightgbm` gradient boosting machine as follows:

1. Create `lightgbm` `Dataset` objects from the train and test sets:

```
train_data = lgb.Dataset(data=X_train, label=y_train)
test_data = train_data.create_valid(X_test, label=y_test)
```

2. Define the training parameters for a multiclass model with five classes (using defaults otherwise):

```
params = {'objective'  : 'multiclass',
          'num_classes': 5}
```

3. Train the model for 250 iterations and monitor the validation set error:

```
lgb_model = lgb.train(params=params,
                      train_set=train_data,
                      num_boost_round=250,
                      valid_sets=[train_data, test_data],
                      verbose_eval=25)
```

4. Lightgbm predicts probabilities for all five classes. We obtain class predictions using `np.argmax()` to obtain the column index with the highest predicted probability:

```
y_pred = np.argmax(lgb_model.predict(X_test), axis=1)
```

5. We compute the accuracy score to evaluate the result and see an improvement of more than 100% over the baseline of 20% for five balanced classes:

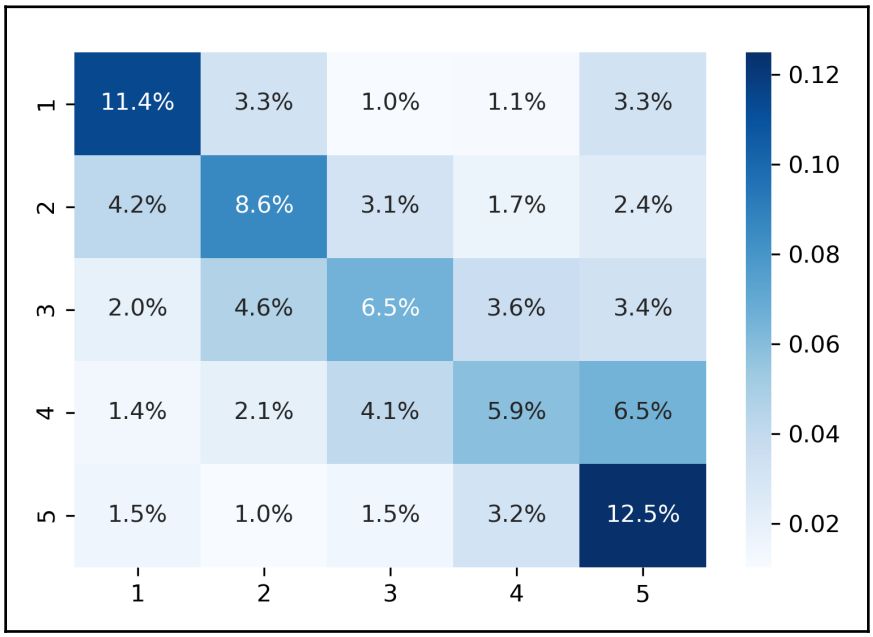
```
accuracy_score(y_true=y_test, y_pred=y_pred)
0.44955063467061984
```

6. Finally, we take a closer look at predictions for each class using the confusion matrix:

```
cm = confusion_matrix(y_true=y_test, y_pred=y_pred)
cm = pd.DataFrame(cm / np.sum(cm), index=stars, columns=stars)
```

7. And visualize the result as a `seaborn` heatmap:

```
sns.heatmap(cm, annot=True, cmap='Blues', fmt='.1%')
```



In sum, the `doc2vec` method allowed us to achieve a very substantial improvement in test accuracy over a naive benchmark without much tuning. If we only select top and bottom reviews (with five and one stars, respectively) and train a binary classifier, the AUC score achieves over 0.86 using 250,000 samples from each class.

Bonus – Word2vec for translation

The notebook translation demonstrates that the relationships encoded in one language often correspond to similar relationships in another language.

It illustrates how word vectors can be used to translate words and phrases by projecting word vectors from the embedding space of one language into the space of another language using a translation matrix.

Summary

This chapter started with how word embeddings encode semantics for individual tokens more effectively than the bag-of-words model that we used in `Chapter 13`, *Working with Text Data.* We also saw how to evaluated embedding by validating if semantic relationships among words are properly represented using linear vector arithmetic.

To learn word embeddings, we use shallow neural networks that used to be slow to train at the scale of web data containing billions of tokens. The `word2vec` model combines several algorithmic innovations to dramatically speed up training and has established a new standard for text feature generation. We saw how to use pretrained word vectors using `spaCy` and `gensim`, and learned to train our own word vector embeddings. We then applied a `word2vec` model to SEC filings. Finally, we covered the `doc2vec` extension that learns vector representations for documents in a similar fashion as word vectors and applied it to Yelp business reviews.

Now, we will begin part 4 on deep learning (available online as mentioned in the Preface), starting with an introduction to feed-forward networks, popular deep learning frameworks and techniques for efficient training at scale.