16
Word Embeddings for Earnings Calls and SEC Filings

In the two previous chapters, we converted text data into a numerical format using the **bag-of-words model**. The result is sparse, fixed-length vectors that represent documents in high-dimensional word space. This allows the similarity of documents to be evaluated and creates features to train a model with a view to classifying a document's content or rating the sentiment expressed in it. However, these vectors ignore the context in which a term is used so that two sentences containing the same words in a different order would be encoded by the same vector, even if their meaning is quite different.

This chapter introduces an alternative class of algorithms that use **neural networks** to learn a vector representation of individual semantic units like a word or a paragraph. These vectors are dense rather than sparse, have a few hundred real-valued entries, and are called **embeddings** because they assign each semantic unit a location in a continuous vector space. They result from training a model to **predict tokens from their context** so that similar usage implies a similar embedding vector. Moreover, the embeddings encode semantic aspects like relationships among words by means of their relative location. As a result, they are powerful features for deep learning models for solving tasks that require semantic information, such as machine translation, question answering, or maintaining a dialogue.

To develop a **trading strategy based on text data**, we are usually interested in the meaning of documents rather than individual tokens. For example, we might want to create a dataset that uses features representing a tweet or a news article with sentiment information (refer to *Chapter 14, Text Data for Trading – Sentiment Analysis*), or an asset's return for a given horizon after publication. Although the bag-of-words model loses plenty of information when encoding text data, it has the advantage of representing an entire document. However, word embeddings have been further developed to represent more than individual tokens. Examples include the **doc2vec** extension, which resorts to weighting word embeddings. More recently, the **attention** mechanism emerged to produce more context-sensitive sentence representations, resulting in **transformer** architectures such as the **BERT** family of models that has dramatically improved performance on numerous natural language tasks.

More specifically, after working through this chapter and the companion notebooks, you will know about the following:

- What word embeddings are, how they work, and why they capture semantic information
- How to obtain and use pretrained word vectors
- Which network architectures are most effective at training word2vec models
- How to train a word2vec model using Keras, Gensim, and TensorFlow
- Visualizing and evaluating the quality of word vectors
- How to train a word2vec model on SEC filings to predict stock price moves
- How doc2vec extends word2vec and can be used for sentiment analysis
- Why the transformer's attention mechanism had such an impact on natural language processing
- How to fine-tune pretrained BERT models on financial data and extract high-quality embeddings

You can find the code examples and links to additional resources in the GitHub directory for this chapter. This chapter uses neural networks and deep learning; if unfamiliar, you may want to first read *Chapter 17*, *Deep Learning for Trading*, which introduces key concepts and libraries.

How word embeddings encode semantics

The bag-of-words model represents documents as sparse, high-dimensional vectors that reflect the tokens they contain. Word embeddings represent tokens as dense, lower-dimensional vectors so that the relative location of words reflects 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 assigned nearby embeddings, but words can have multiple degrees of similarity. For example, the word "driver" could be similar to "motorist" or to "factor." Furthermore, embeddings encode relationships among pairs of words like 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 neural network to predict words from their context or vice versa. In this section, we will introduce how these models work and present successful approaches, including word2vec, doc2vec, and the more recent transformer family of models.

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 entire document as context and relies on (weighted) counts to capture the co-occurrence of words.

Earlier neural language models used included nonlinear hidden layers that increased the computational complexity. **word2vec**, introduced by Mikolov, Sutskever, et al. (2013) and its extensions simplified the architecture to enable training on large datasets. The Wikipedia corpus, for example, contains over 2 billion tokens. (Refer to *Chapter 17*, *Deep Learning for Trading*, for additional details on feedforward networks.)

word2vec – scalable word and phrase embeddings

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, shown in the following diagram, to efficiently learn word vectors using shallow neural networks (Mikolov, Chen, et al., 2013):

- 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. CBOW trains faster and tends to be slightly more accurate for frequent terms, but pays less attention to infrequent words.
- The **skip-gram** (**SG**) model, in 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.

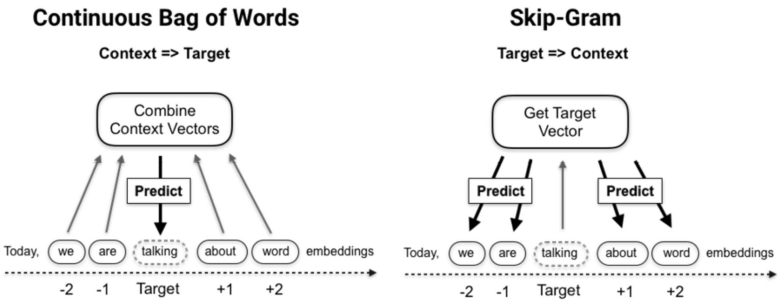

Figure 16.1: Continuous-bag-of-words versus skip-gram processing logic

The 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.

During training, the **backpropagation algorithm** adjusts the embedding weights in response to the loss computed by an objective function based on 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.

The skip-gram model implicitly factorizes a word-context matrix that contains the pointwise mutual information of the respective word and context pairs (Levy and Goldberg, 2014).

Model objective – simplifying the softmax

Word2vec models aim to predict a single word out of a potentially very large vocabulary. Neural networks often use the softmax function as an output unit in the final layer to implement the multiclass objective because it maps an arbitrary number of real values to an equal number of probabilities. The softmax function is defined as follows, where h refers to the embedding and v to the input vectors, and c is the context of word w:

$$p(w|c) = \frac{\exp\left(h^{\mathrm{T}} v'_w\right)}{\sum_{w_i \in V} \exp\left(h^{\mathrm{T}} v'_{w_i}\right)}$$

However, the softmax complexity scales with the number of classes because the denominator requires computing the dot product for all words in the vocabulary to standardize the probabilities. Word2vec gains efficiency by using a modified version of the softmax or sampling-based approximations:

- 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 (Morin and Bengio, 2005).
- **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 45 times faster (Mnih and Kavukcuoglu, 2013).
- **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 (Mikolov et al., 2013).

Automating phrase detection

Preprocessing typically involves phrase detection, that is, the identification of tokens that are commonly used together and should receive a sin-

gle vector representation (for example, New York City; refer to the discussion of n-grams in *Chapter 13*, *Data-Driven Risk Factors and Asset Allocation with Unsupervised Learning*).

The original word2vec authors (Mikolov et al., 2013) 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:

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

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

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

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

Evaluating embeddings using semantic arithmetic

The bag-of-words model creates document vectors that reflect the presence and relevance of tokens to the document. As discussed in *Chapter 15*, *Topic Modeling – Summarizing Financial News*, **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 word and phrase vectors produced by word2vec do not have an explicit meaning. However, the **embeddings encode similar usage as proximity** in the latent space created by the model. The embeddings also capture semantic relationships so that analogies can be expressed by adding and subtracting word vectors.

Figure 16.2 shows how the vector that points from "Paris" to "France" (which measures the difference between their embedding vectors) reflects the "capital of" relationship. The analogous relationship between London and the UK corresponds to the same vector: the embedding for the term "UK" is very close to the location obtained by adding the "capital of" vector to the embedding for the term "London":

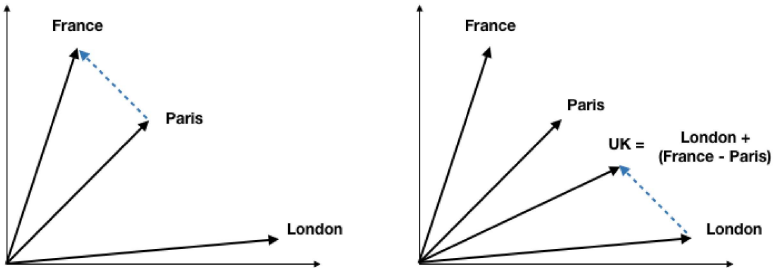

Figure 16.2: Embedding vector arithmetic

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 over 25,000 relationships in 14 categories spanning aspects of geography, grammar and syntax, and family relationships to evaluate the quality of embedding vectors. As illustrated in the preceding diagram, 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 table shows the number of samples and illustrates some of the analogy categories. The test checks how close the embedding for *d* is to the location determined by *c* + *(b-a)*. Refer to the `evaluating_embeddings` notebook for implementation details.

Category	# Samples	a	b	c	d
Capital-Country	506	athens	greece	baghdad	iraq
City-State	4,242	chicago	illinois	houston	texas
Past Tense	1,560	dancing	danced	decreasing	decreased
Plural	1,332	banana	bananas	bird	birds
Comparative	1,332	bad	worse	big	bigger
Opposite	812	acceptable	unacceptable	aware	unaware

Superlative	1,122	bad	worst	big	biggest
Plural (Verbs)	870	de-crease	de-creases	de-scribe	de-scribes
Currency	866	algeria	dinar	angola	kwanza
Family	506	boy	girl	brother	sister

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 a couple of options to obtain embedding vectors for a given corpus of documents:

- Use pretrained embeddings learned from a generic large corpus like 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, the more preferable the second approach. However, quality word embeddings are data-hungry and require informative documents containing hundreds of millions of words.

We will first look at how you can use pretrained vectors and then demonstrate examples of how to build your own word2vec models using financial news and SEC filings data.

How to use pretrained word vectors

There are several sources for pretrained word embeddings. Popular options include Stanford's GloVE and spaCy's built-in vectors (refer to the `using_pretrained_vectors` notebook for details). In this section, we will focus on GloVe.

GloVe – Global vectors for word representation

GloVe (*Global Vectors for Word Representation*, Pennington, Socher, and Manning, 2014) 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 resources linked on GitHub). Vectors pretrained on the following web-scale sources are available:

- **Common Crawl** with 42 billion or 840 billion tokens and a vocabulary or 1.9 million or 2.2 million tokens
- **Wikipedia** 2014 + Gigaword 5 with 6 billion tokens and a vocabulary of 400,000 tokens

- **Twitter** using 2 billion tweets, 27 billion tokens, and a vocabulary of 1.2 million tokens

We can use Gensim to convert the vector text files using `glove2word2vec` and then load them 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)
```

Gensim uses the **word2vec analogy tests** described in the previous section using text files made available by the authors to evaluate word vectors. For this purpose, the library has the `wv.accuracy` function, which we use to pass the path to the analogy file, indicate whether the word vectors are in binary format, and whether we want to ignore the case. We can also restrict the vocabulary to the most frequent to speed up testing:

```
accuracy = model.wv.accuracy(analogies_path,
                             restrict_vocab=300000,
                             case_insensitive=True)
```

The word vectors trained on the Wikipedia corpus cover all analogies and achieve an overall accuracy of 75.44 percent with some variation across categories:

Category	# Samples	Accuracy	Category	# Samples	Accuracy
Capital-Country	506	94.86%	Comparative	1,332	88.21%
Capitals RoW	8,372	96.46%	Opposite	756	28.57%
City-State	4,242	60.00%	Superlative	1,056	74.62%
Currency	752	17.42%	Present-Participle	1,056	69.98%
Family	506	88.14%	Past Tense	1,560	61.15%
Nationality	1,640	92.50%	Plural	1,332	78.08%
Adjective-Adverb	992	22.58%	Plural Verbs	870	58.51%

Figure 16.3 compares the performance for the three GloVe sources for the 100,000 most common tokens. It shows that Common Crawl vectors,

which cover about 80 percent of the analogies, achieve slightly higher accuracy at 78 percent. The Twitter vectors cover only 25 percent, with 56.4 percent accuracy:

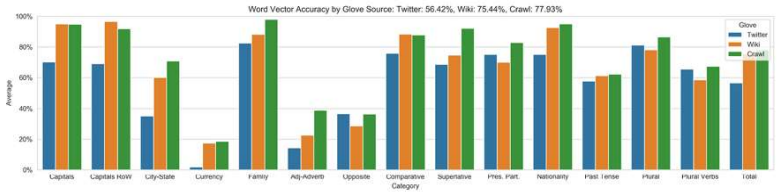

Figure 16.3: GloVe accuracy on word2vec analogies

Figure 16.4 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 PCA. A test of over 24,400 analogies from the following categories achieved an accuracy of over 73.5 percent:

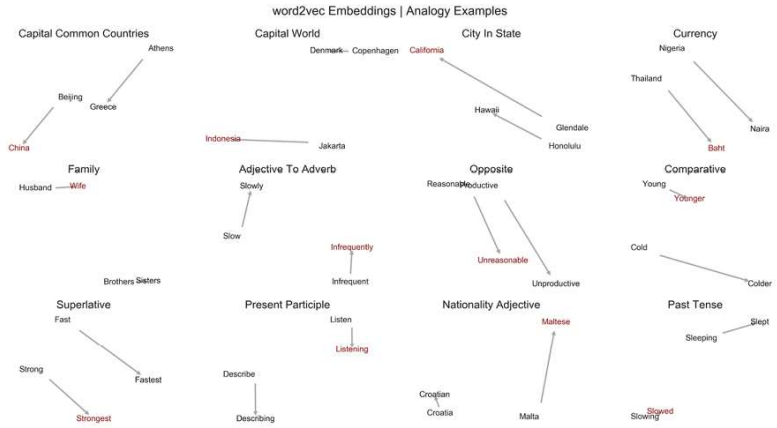

Figure 16.4: 2D visualization of selected analogy embeddings

Custom embeddings for financial news

Many tasks require embeddings of domain-specific vocabulary that models pretrained on a generic corpus may not be able to capture. 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, documents like corporate earnings releases use nuanced language that GloVe vectors pretrained on Wikipedia articles are unlikely to properly reflect.

In this section, we will train and evaluate domain-specific embeddings using financial news. We'll first show how to preprocess the data for this task, then demonstrate how the skip-gram architecture outlined in the first section works, and finally visualize the results. We also will introduce alternative, faster training methods.

Preprocessing – sentence detection and n-grams

To illustrate the word2vec network architecture, we'll use the financial news dataset with over 125,000 relevant articles that we introduced in *Chapter 15*, *Topic Modeling – Summarizing Financial News*, on topic modeling. We'll load the data as outlined in the `lda_financial_news.ipynb` notebook in that chapter. The `financial_news_preprocessing.ipynb` notebook contains the code samples for this section.

We use spaCy's built-in **sentence boundary detection** to split each article into sentences, remove less informative items, such as numbers and punctuation, and keep the result if it is between 6 and 99 tokens long:

```
def clean_doc(d):
    doc = []
    for sent in d.sents:
        s = [t.text.lower() for t in sent if not
        any([t.is_digit, not t.is_alpha, t.is_punct, t.is_space])]
        if len(s) > 5 or len(sent) < 100:
            doc.append(' '.join(s))
    return doc
nlp = English()
sentencizer = nlp.create_pipe("sentencizer")
nlp.add_pipe(sentencizer)
clean_articles = []
iter_articles = (article for article in articles)
for i, doc in enumerate(nlp.pipe(iter_articles, batch_size=100, n_process=8), 1):
    clean_articles.extend(clean_doc(doc))
```

We end up with 2.43 million sentences that, on average, contain 15 tokens.

Next, we create n-grams to capture composite terms. Gensim lets us identify n-grams based on the relative frequency of joint versus individual occurrence of the components. The `Phrases` module scores the tokens, and the `Phraser` class transforms the text data accordingly.

It transforms our list of sentences into a new dataset that we can write to file as follows:

```
sentences = LineSentence((data_path / f'articles_clean.txt').as_posix())
phrases = Phrases(sentences=sentences,
                          min_count=10,  # 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]
with (data_path / f'articles_ngrams.txt').open('w') as f:
        for sentence in sentences:
            f.write(' '.join(sentence) + '\n')
```

The notebook illustrates how we can repeat this process using the 2-gram file as input to create 3-grams. We end up with some 25,000 2-grams and 15,000 3- or 4-grams. Inspecting the result shows that the highest-scoring terms are names of companies or individuals, suggesting that we might want to tighten our initial cleaning criteria. Refer to the notebook for additional details on the dataset.

The skip-gram architecture in TensorFlow 2

In this section, we will illustrate how to build a word2vec model using the Keras interface of TensorFlow 2 that we will introduce in much more detail in the next chapter. The `financial_news_word2vec_tensorflow` notebook contains the code samples and additional implementation details.

We start by tokenizing the documents and assigning a unique ID to each item in the vocabulary. First, we sample a subset of the sentences created in the previous section to limit the training time:

```
SAMPLE_SIZE=.5
sentences = file_path.read_text().split('\n')
words = ' '.join(np.random.choice(sentences, size=int(SAMLE_SIZE* l en(sentences)), replace=Fal
```

We require at least 10 occurrences in the corpus, keep a vocabulary of 31,300 tokens, and begin with the following steps:

1. Extract the top *n* most common words to learn embeddings.
2. Index these *n* words with unique integers.
3. Create an `{index: word}` dictionary.
4. Replace the *n* words with their index, and a dummy value `'UNK'` elsewhere:

```
# Get (token, count) tuples for tokens meeting MIN_FREQ
MIN_FREQ = 10
token_counts = [t for t in Counter(words).most_common() if t[1] >= MIN_FREQ]
tokens, counts = list(zip(*token_counts))
# create id-token dicts & reverse dicts
id_to_token = pd.Series(tokens, index=range(1, len(tokens) + 1)).to_dict()
id_to_token.update({0: 'UNK'})
token_to_id = {t:i for i, t in id_to_token.items()}
data = [token_to_id.get(word, 0) for word in words]
```

We end up with 17.4 million tokens and a vocabulary of close to 60,000 tokens, including up to 3-grams. The vocabulary covers around 72.5 percent of the analogies.

Noise-contrastive estimation – creating validation samples

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. A lower factor increases the probability of selecting less frequent tokens; a chart in the notebook shows that the value of 0.1 limits sampling to the top 10,000 tokens:

```
SAMPLING_FACTOR =  1e-4
sampling_table = make_sampling_table(vocab_size,
                                     sampling_factor=SAMPLING_FACTOR)
```

Generating target-context word pairs

To train our model, we need pairs of tokens where one represents the target and the other is selected from the surrounding context window, as shown previously in the right panel of *Figure 16.1*. We can use Keras' `skipgrams()` function as follows:

```
pairs, labels = skipgrams(sequence=data,
                          vocabulary_size=vocab_size,
                          window_size=WINDOW_SIZE,
                          sampling_table=sampling_table,
                          negative_samples=1.0,
                          shuffle=True)
```

The result is 120.4 million context-target pairs, evenly split between positive and negative samples. The negative samples are generated according to the `sampling_table` probabilities we created in the previous step. The first five target and context word IDs with their matching labels appear as follows:

```
pd.DataFrame({'target': target_word[:5],
              'context': context_word[:5],
              'label': labels[:5]})
   target context label
0   30867    2117     1
1     196     359     1
2   17960   32467     0
3     314    1721     1
4   28387    7811     0
```

Creating the word2vec model layers

The word2vec model contains the following:

- An input layer that receives the two scalar values representing the target-context pair
- A shared embedding layer that computes the dot product of the vector for the target and context word

- A sigmoid output layer

The **input layer** has two components, one for each element of the target-context pair:

```
input_target = Input((1,), name='target_input')
input_context = Input((1,), name='context_input')
```

The **shared embedding layer** contains one vector for each element of the vocabulary that is selected according to the index of the target and context tokens, respectively:

```
embedding = Embedding(input_dim=vocab_size,
                      output_dim=EMBEDDING_SIZE,
                      input_length=1,
                      name='embedding_layer')
target = embedding(input_target)
target = Reshape((EMBEDDING_SIZE, 1), name='target_embedding')(target)
context = embedding(input_context)
context = Reshape((EMBEDDING_SIZE, 1), name='context_embedding')(context)
```

The **output layer** measures the similarity of the two embedding vectors by their dot product and transforms the result using the `sigmoid` function that we encountered when discussing logistic regression in *Chapter 7, Linear Models – From Risk Factors to Return Forecasts*:

```
# similarity measure
dot_product = Dot(axes=1)([target, context])
dot_product = Reshape((1,), name='similarity')(dot_product)
output = Dense(units=1, activation='sigmoid', name='output')(dot_product)
```

This skip-gram model contains a 200-dimensional embedding layer that will assume different values for each vocabulary item. As a result, we end up with 59,617 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 two or three dimensions to explore the word and phrase locations. After loading the embedding metadata file we created (refer to the notebook), you can also search for specific terms to view and explore its neighbors, projected into two or three dimensions using UMAP, t-SNE, or PCA (refer to *Chapter 13, Data-Driven Risk Factors and*

Asset Allocation with Unsupervised Learning). Refer to the notebook for a higher-resolution color version of the following screenshot:

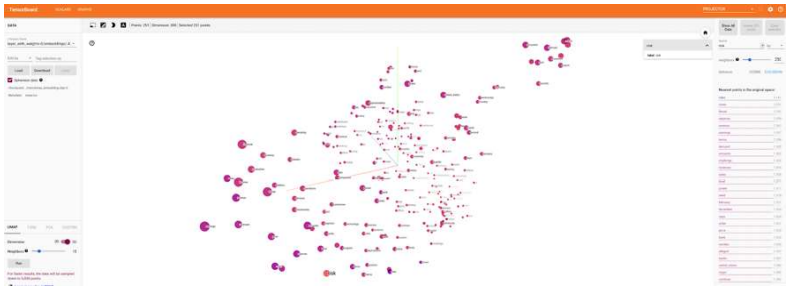

Figure 16.5: 3D embeddings and metadata visualization

How to train embeddings faster with Gensim

The TensorFlow implementation is very transparent in terms of its architecture, but it is not particularly fast. The **natural language processing (NLP)** library Gensim, which we also used for topic modeling in the last chapter, offers better performance and more closely resembles the C-based word2vec implementation provided by the original authors.

Usage is very straightforward. We first create a sentence generator that just takes the name of the file we produced in the preprocessing step as input (we'll work with 3-grams again):

```
sentence_path = data_path / FILE_NAME
sentences = LineSentence(str(sentence_path))
```

In a second step, we configure the word2vec model with the familiar parameters concerning the sizes of the embedding vector and the context window, the minimum token frequency, and the number of negative samples, among others:

```
model = Word2Vec(sentences,
                 sg=1, # set to 1 for skip-gram; CBOW otherwise
                 size=300,
                 window=5,
                 min_count=20,
                 negative=15,
                 workers=8,
                 iter=EPOCHS,
                 alpha=0.05)
```

One epoch of training takes a bit over 2 minutes on a modern 4-core i7 processor.

We can persist both the model and the word vectors, or just the word vectors, as follows:

```
# persist model
model.save(str(gensim_path / 'word2vec.model'))
# persist word vectors
model.wv.save(str(gensim_path / 'word_vectors.bin'))
```

We can validate model performance and continue training until we are satisfied with the results like so:

```
model.train(sentences, epochs=1, total_examples=model.corpus_count)
```

In this case, training for six additional epochs yields the best results with an accuracy of 41.75 percent across all analogies covered by the vocabulary. The left panel of *Figure 16.6* shows the correct/incorrect predictions and accuracy breakdown per category.

Gensim also allows us to evaluate custom semantic algebra. We can check the popular `"woman"+"king"-"man" ~ "queen"` example as follows:

```
most_sim = best_model.wv.most_similar(positive=['woman', 'king'], negative=['man'], topn=10)
```

The right panel of the figure shows that "queen" is the third token, right after "monarch" and the less obvious "lewis", followed by several royalties:

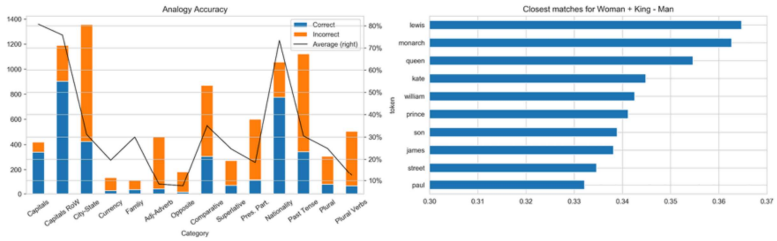

Figure 16.6: Analogy accuracy by category and for a specific example

We can also evaluate the tokens most similar to a given target to gain a better understanding of the embedding characteristics. We randomly select based on log corpus frequency:

```
counter = Counter(sentence_path.read_text().split())
most_common = pd.DataFrame(counter.most_common(), columns=['token', 'count'])
most_common['p'] = np.log(most_common['count'])/np.log(most_common['count']).sum()similars = pd
for token in np.random.choice(most_common.token, size=10, p=most_common.p):
    similars[token] = [s[0] for s in best_model.wv.most_similar(token)]
```

The following table exemplifies the results that include several n-grams:

Target	Closest Match

	0	1	2	3
profiles	profile	users	political_consultancy_cambridge_analytica	sop cate
divestments	divestitures	acquisitions	takeovers	bay
readiness	training	military	command	air_
arsenal	nuclear_weapons	russia	ballistic_missile	wea
supply_disruptions	disruptions	raw_material	disruption	pric

We will now proceed to develop an application more closely related to real-life trading using SEC filings.

word2vec for trading with SEC filings

In this section, we will learn word and phrase vectors from annual 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 will use a dataset containing over **22,000 10-K annual reports** from the period **2013-2016** that are filed by over 6,500 listed companies and contain both financial information and management commentary (see *Chapter 2*, *Market and Fundamental Data – Sources and Techniques*).

For about 3,000 companies corresponding to 11,000 filings, we have stock prices to label the data for predictive modeling. (See data source details and download instructions and preprocessing code samples in the `sec_preprocessing` notebook in the `sec-filings` folder.)

Preprocessing – sentence detection and n-grams

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

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

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

Automatic phrase detection

As in the previous section, we use Gensim to detect phrases that consist of multiple tokens, or n-grams. The notebook shows that the most frequent bigrams include `common_stock`, `united_states`, `cash_flows`, `real_estate`, and `interest_rates`.

We end up with a vocabulary of slightly over 201,000 tokens with a median frequency of 7, suggesting substantial noise that we can remove by increasing the minimum frequency when training our word2vec model.

Labeling filings with returns to predict earnings surprises

The dataset comes with a list of tickers and filing dates associated with the 10,000 documents. We can use this information to select stock prices for a certain period surrounding the filing publication. The goal would be to train a model that uses word vectors for a given filing as input to predict post-filing returns.

The following code example shows how to label individual filings with the 1-month return for the period after filing:

```
with pd.HDFStore(DATA_FOLDER / 'assets.h5') as store:
    prices = store['quandl/wiki/prices'].adj_close
sec = pd.read_csv('sec_path/filing_index.csv').rename(columns=str.lower)
sec.date_filed = pd.to_datetime(sec.date_filed)
sec = sec.loc[sec.ticker.isin(prices.columns), ['ticker', 'date_filed']]
price_data = []
for ticker, date in sec.values.tolist():
    target = date + relativedelta(months=1)
    s = prices.loc[date: target, ticker]
    price_data.append(s.iloc[-1] / s.iloc[0] - 1)
df = pd.DataFrame(price_data,
                  columns=['returns'],
                  index=sec.index)
```

We will come back to this when we work with deep learning architectures in the following chapters.

Model training

The `gensim.models.word2vec` class implements the skip-gram and CBOW architectures introduced previously. The notebook `word2vec` contains additional implementation details.

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

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

The `Word2Vec` class offers the configuration options introduced earlier in this chapter:

```
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 machine learning models.

Model evaluation

Basic functionality includes identifying similar words:

```
sims=model.wv.most_similar(positive=['iphone'], restrict_vocab=15000)
                  term   similarity
0                 ipad    0.795460
1              android    0.694014
2           smartphone    0.665732
```

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 (refer to the detailed results in the `models` folder):

- Negative sampling outperforms the hierarchical softmax, while also training faster.
- The skip-gram architecture outperforms CBOW.
- Different `min_count` settings have a smaller impact; the midpoint of 50 performs best.

Further experiments with the best-performing skip-gram model using negative sampling and a `min_count` of 50 show the following:

- Context windows smaller than 5 reduce 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 percent.

Sentiment analysis using doc2vec embeddings

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.

In contrast, the document embedding model, doc2vec, developed by the word2vec authors shortly after publishing their original contribution, produces embeddings for pieces of text like a paragraph or a product review directly. 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 on 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 word2wec 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. We'll illustrate the use of doc2vec by applying it to the Yelp sentiment dataset that we introduced in *Chapter 14*. To speed up training, we limit the data to a stratified random sample of 0.5 million Yelp reviews with their associated star ratings. The `doc2vec_yelp_sentiment` notebook contains the code examples for this section.

Creating doc2vec input from Yelp sentiment data

We load the combined Yelp dataset containing 6 million reviews, as created in *Chapter 14*, *Text Data for Trading – Sentiment Analysis*, and sample 100,000 reviews for each star rating:

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

We use nltk's `RegexpTokenizer` for simple and quick text cleaning:

```
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)
```

After we filter out reviews shorter than 10 tokens, we are left with 485,825 samples. The left panel of *Figure 16.6* shows the distribution of the number of tokens per review.

The `gensim.models.Doc2Vec` class processes documents in the `TaggedDocument` format that contains the tokenized documents alongside a unique tag that permits the document vectors to be accessed after training:

```
sample = pd.read_parquet('yelp_sample.parquet')
sentences = []
for i, (stars, text) in df.iterrows():
    sentences.append(TaggedDocument(words=text.split(), tags=[i]))
```

Training a doc2vec model

The training interface works in a similar fashion to word2vec and also allows continued training and persistence:

```
model = Doc2Vec(documents=sentences,
                dm=1,           # 1=distributed memory, 0=dist.BOW
                epochs=5,
                size=300,       # vector size
                window=5,       # max. distance betw. target and context
                min_count=50,   # ignore tokens w. lower frequency
                negative=5,     # negative training samples
                dm_concat=0,    # 1=concatenate vectors, 0=sum
                dbow_words=0,   # 1=train word vectors as well
                workers=4)
model.save((results_path / 'sample.model').as_posix())
```

We can query the *n* terms most similar to a given token as a quick way to evaluate the resulting word vectors as follows:

```
model.most_similar('good')
```

The right panel of *Figure 16.7* displays the returned tokens and their similarity:

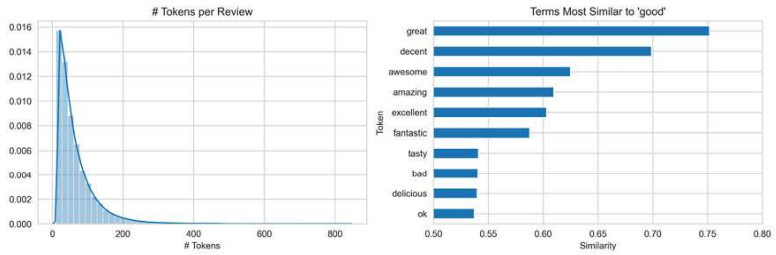

Figure 16.7: Histogram of the number of tokens per review (left) and terms most similar to the token 'good'

Training a classifier with document vectors

Now, we can access the document vectors to create features for a sentiment classifier:

```
y = sample.stars.sub(1)
X = np.zeros(shape=(len(y), size)) # size=300
for i in range(len(sample)):
    X[i] = model.docvecs[i]
X.shape
(485825, 300)
```

We create training and test sets as usual:

```
X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    test_size=0.2,
                                                    random_state=42,
                                                    stratify=y)
```

Now, we proceed to train a `RandomForestClassifier`, a LightGBM gradient boosting model, and a multinomial logistic regression. We use 500 trees for the random forest:

```
rf = RandomForestClassifier(n_jobs=-1, n_estimators=500)
rf.fit(X_train, y_train)
rf_pred = rf.predict(X_test)
```

We use early stopping with the LightGBM classifier, but it runs for the full 5,000 rounds because it continues to improve its validation performance:

```
train_data = lgb.Dataset(data=X_train, label=y_train)
test_data = train_data.create_valid(X_test, label=y_test)
params = {'objective': 'multiclass',
          'num_classes': 5}
lgb_model = lgb.train(params=params,
                      train_set=train_data,
```

```
                        num_boost_round=5000,
                        valid_sets=[train_data, test_data],
                        early_stopping_rounds=25,
                        verbose_eval=50)
# generate multiclass predictions
lgb_pred = np.argmax(lgb_model.predict(X_test), axis=1)
```

Finally, we build a multinomial logistic regression model as follows:

```
lr = LogisticRegression(multi_class='multinomial', solver='lbfgs',
                        class_weight='balanced')
lr.fit(X_train, y_train)
lr_pred = lr.predict(X_test)
```

When we compute the accuracy for each model on the validation set, gradient boosting performs significantly better at 62.24 percent. *Figure 16.8* shows the confusion matrix and accuracy for each model:

Figure 16.8: Confusion matrix and test accuracy for alternative models

The sentiment classification result in *Chapter 14*, *Text Data for Trading – Sentiment Analysis*, produced better accuracy for LightGBM (73.6 percent), but we used the full dataset and included additional features. You may want to test whether increasing the sample size or tuning the model parameters makes doc2vec perform equally well.

Lessons learned and next steps

This example applied sentiment analysis using doc2vec to **product reviews rather than financial documents**. We selected product reviews because it is very difficult to find financial text data that is large enough for training word embeddings from scratch and also has useful sentiment labels or sufficient information for us to assign them labels, such as asset returns, ourselves.

While product reviews allow us to demonstrate the workflow, we need to keep in mind **important structural differences**: product reviews are often short, informal, and specific to one particular object. Many financial documents, in contrast, are longer, more formal, and the target object may or may not be clearly identified. Financial news articles could concern multiple targets, and while corporate disclosures may have a clear source, they may also discuss competitors. An analyst report, for instance,

may also discuss both positive and negative aspects of the same object or topic.

In short, the interpretation of sentiment expressed in financial documents often requires a more sophisticated, nuanced, and granular approach that builds up an understanding of the content's meaning from different aspects. Decision makers also often care to understand how a model arrives at its conclusion.

These challenges have not yet been solved and remain an area of very active research, complicated not least by the scarcity of suitable data sources. However, recent breakthroughs that significantly boosted performance on various NLP tasks since 2018 suggest that financial sentiment analysis may also become more robust in the coming years. We will turn to these innovations next.

New frontiers – pretrained transformer models

Word2vec and GloVe embeddings capture more semantic information than the bag-of-words approach. However, they allow only a single fixed-length representation of each token that does not differentiate between context-specific usages. To address unsolved problems such as multiple meanings for the same word, called **polysemy**, several new models have emerged that build on the **attention mechanism** designed to learn more contextualized word embeddings (Vaswani et al., 2017). The key characteristics of these models are as follows:

- The use of **bidirectional language models** that process text both left-to-right and right-to-left for a richer context representation
- The use of **semi-supervised pretraining** on a large generic corpus to learn universal language aspects in the form of embeddings and network weights that can be used and fine-tuned for specific tasks (a form of **transfer learning** that we will discuss in more detail in *Chapter 18, CNNs for Financial Time Series and Satellite Images*)

In this section, we briefly describe the attention mechanism, outline how the recent transformer models—starting with **Bidirectional Encoder Representation from Transformers** (**BERT**)—use it to improve performance on key NLP tasks, reference several sources for pretrained language models, and explain how to use them for financial sentiment analysis.

Attention is all you need

The **attention mechanism** explicitly models the relationships between words in a sentence to better incorporate the context. It was first applied to machine translation (Bahdanau, Cho, and Bengio, 2016), but has since become integral to neural language models for a wide variety of tasks.

Until 2017, **recurrent neural networks** (**RNNs**), which sequentially process text left-to-right or right-to-left, represented the state of the art for NLP tasks like translation. Google, for example, has employed such a model in production since late 2016. Sequential processing implies several steps to semantically connect words at distant locations and precludes parallel processing, which greatly speeds up computation on modern, specialized hardware like GPUs. (For more information on RNNs, refer to *Chapter 19*, *RNNs for Multivariate Time Series and Sentiment Analysis*.)

In contrast, the **Transformer** model, introduced in the seminal paper *Attention is all you need* (Vaswani et al., 2017), requires only a constant number of steps to identify semantically related words. It relies on a self-attention mechanism that captures links between all words in a sentence, regardless of their relative position. The model learns the representation of a word by assigning an attention score to every other word in the sentence that determines how much each of the other words should contribute to the representation. These scores then inform a weighted average of all words' representations, which is fed into a fully connected network to generate a new representation for the target word.

The Transformer model uses an encoder-decoder architecture with several layers, each of which uses several attention mechanisms (called **heads**) in parallel. It yielded large performance improvements on various translation tasks and, more importantly, inspired a wave of new research into neural language models addressing a broader range of tasks. The resources linked on GitHub contain various excellent visual explanations of how the attention mechanism works, so we won't go into more detail here.

BERT – towards a more universal language model

In 2018, Google released the **BERT** model, which stands for **Bidirectional Encoder Representations from Transformers** (Devlin et al., 2019). In a major breakthrough for NLP research, it achieved groundbreaking results on eleven natural language understanding tasks, ranging from question answering and named entity recognition to paraphrasing and sentiment analysis, as measured by the **General Language Understanding Evaluation** (**GLUE**) benchmark (see GitHub for links to task descriptions and a leaderboard).

The new ideas introduced by BERT unleashed a flurry of new research that produced dozens of improvements that soon surpassed non-expert humans on the GLUE tasks and led to the more challenging **SuperGLUE** benchmark designed by DeepMind (Wang et al., 2019). As a result, 2018 is now considered a turning point for NLP research; both Google Search and Microsoft's Bing are now using variations of BERT to interpret user queries and provide more accurate results.

We will briefly outline BERT's key innovations and provide indications on how to get started using it and its subsequent enhancements with one of

several open source libraries providing pretrained models.

Key innovations – deeper attention and pretraining

The BERT model builds on **two key ideas**, namely, the **transformer architecture** described in the previous section and **unsupervised pretraining** so that it doesn't need to be trained from scratch for each new task; rather, its weights are fine-tuned:

- BERT takes the **attention mechanism** to a new (deeper) level by using 12 or 24 layers, depending on the architecture, each with 12 or 16 attention heads. This results in up to 24 × 16 = 384 attention mechanisms to learn context-specific embeddings.
- BERT uses **unsupervised, bidirectional pretraining** to learn its weights in advance on two tasks: **masked language modeling** (predicting a missing word given the left and right context) and **next sentence prediction** (predicting whether one sentence follows another).

Context-free models such as word2vec or GloVe generate a single embedding for each word in the vocabulary: the word "bank" would have the same context-free representation in "bank account" and "bank of the river." In contrast, BERT learns to represent each word based on the other words in the sentence. As a **bidirectional model**, BERT is able to represent the word "bank" in the sentence "I accessed the bank account," not only based on "I accessed the" as a unidirectional contextual model, but also based on "account."

BERT and its successors can be **pretrained on a generic corpus** like Wikipedia before adapting its final layers to a specific task and **fine-tuning its weights**. As a result, you can use large-scale, state-of-the-art models with billions of parameters, while only incurring a few hours rather than days or weeks of training costs. Several libraries offer such pretrained models that you can build on to develop a custom sentiment classifier for your dataset of choice.

Using pretrained state-of-the-art models

The recent NLP breakthroughs described in this section have shown how to acquire linguistic knowledge from unlabeled text with networks large enough to represent the long tail of rare usage phenomena. The resulting Transformer architectures make fewer assumptions about word order and context; instead, they learn a much more subtle understanding of language from very large amounts of data, using hundreds of millions or even billions of parameters.

We will highlight several libraries that make pretrained networks, as well as excellent Python tutorials available.

The Hugging Face Transformers library

Hugging Face is a US start-up developing chatbot applications designed to offer personalized AI-powered communication. It raised $15 million in

late 2019 to further develop its very successful open source NLP library, Transformers.

The library provides general-purpose architectures for natural language understanding and generation with more than 32 pretrained models in more than 100 languages and deep interoperability between TensorFlow 2 and PyTorch. It has excellent documentation.

The spacy-transformers library includes wrappers to facilitate the inclusion of the pretrained transformer models in a spaCy pipeline. Refer to the reference links on GitHub for more information.

AllenNLP

AllenNLP is built and maintained by the Allen Institute for AI, started by Microsoft cofounder Paul Allen, in close collaboration with researchers at the University of Washington. It has been designed as a research library for developing state-of-the-art deep learning models on a wide variety of linguistic tasks, built on PyTorch.

It offers solutions for key tasks from question answering to sentence annotation, including reading comprehension, named entity recognition, and sentiment analysis. A pretrained **RoBERTa** model (a more robust version of BERT; Liu et al., 2019) achieves over 95 percent accuracy on the Stanford sentiment treebank and can be used with just a few lines of code (see links to the documentation on GitHub).

Trading on text data – lessons learned and next steps

As highlighted at the end of the section *Sentiment analysis using doc2vec embeddings*, there are important structural characteristics of financial documents that often complicate their interpretation and undermine simple dictionary-based methods.

In a recent survey of financial sentiment analysis, Man, Luo, and Lin (2019) found that most existing approaches only identify high-level polarities, such as positive, negative, or neutral. However, practical applications that lead to real decisions typically require a more nuanced and transparent analysis. In addition, the lack of large financial text datasets with relevant labels limits the potential for using traditional machine learning methods or neural networks for sentiment analysis.

The pretraining approach just described, which, in principle, yields a deeper understanding of textual information, thus offers substantial promise. However, most applied research using transformers has focused on NLP tasks such as translation, question answering, logic, or dialog systems. Applications in relation to financial data are still in their infancy (see, for example, Araci 2019). This is likely to change soon given the availability of pretrained models and their potential to extract more valuable information from financial text data.

Summary

In this chapter, we discussed a new way of generating text features that use shallow neural networks for unsupervised machine learning. We saw how the resulting word embeddings capture interesting semantic aspects beyond the meaning of individual tokens by capturing some of the context in which they are used. We also covered how to evaluate the quality of word vectors using analogies and linear algebra.

We used Keras to build the network architecture that produces these features and applied the more performant Gensim implementation to financial news and SEC filings. Despite the relatively small datasets, the word2vec embeddings did capture meaningful relationships. We also demonstrated how appropriate labeling with stock price data can form the basis for supervised learning.

We applied the doc2vec algorithm, which produces a document rather than token vectors, to build a sentiment classifier based on Yelp business reviews. While this is unlikely to yield tradeable signals, it illustrates the process of how to extract features from relevant text data and train a model to predict an outcome that may be informative for a trading strategy.

Finally, we outlined recent research breakthroughs that promise to yield more powerful natural language models due to the availability of pre-trained architectures that only require fine-tuning. Applications to financial data, however, are still at the research frontier.

In the next chapter, we will dive into the final part of this book, which covers how various deep learning architectures can be useful for algorithmic trading.