

for this reason, SHAP values provide differentiated insights into how the impact of a feature varies across samples, which is important, given the role of interaction effects in these nonlinear models.

How to summarize SHAP values by feature

To get a high-level overview of the feature importance across a number of samples, there are two ways to plot the SHAP values: a simple average across all samples that resembles the global feature-importance measures computed previously (as shown in the left-hand panel of *Figure 12.15*), or a scatterplot to display the impact of every feature for every sample (as shown in the right-hand panel of the figure). They are very straightforward to produce using a trained model from a compatible library and matching input data, as shown in the following code:

```
# Load JS visualization code to notebook
shap.initjs()
# explain the model's predictions using SHAP values
explainer = shap.TreeExplainer(model)
shap_values = explainer.shap_values(X_test)
shap.summary_plot(shap_values, X_test, show=False)
```

The scatterplot sorts features by their total SHAP values across all samples and then shows how each feature impacts the model output, as measured by the SHAP value, as a function of the feature's value, represented by its color, where red represents high values and blue represents low values relative to the feature's range:

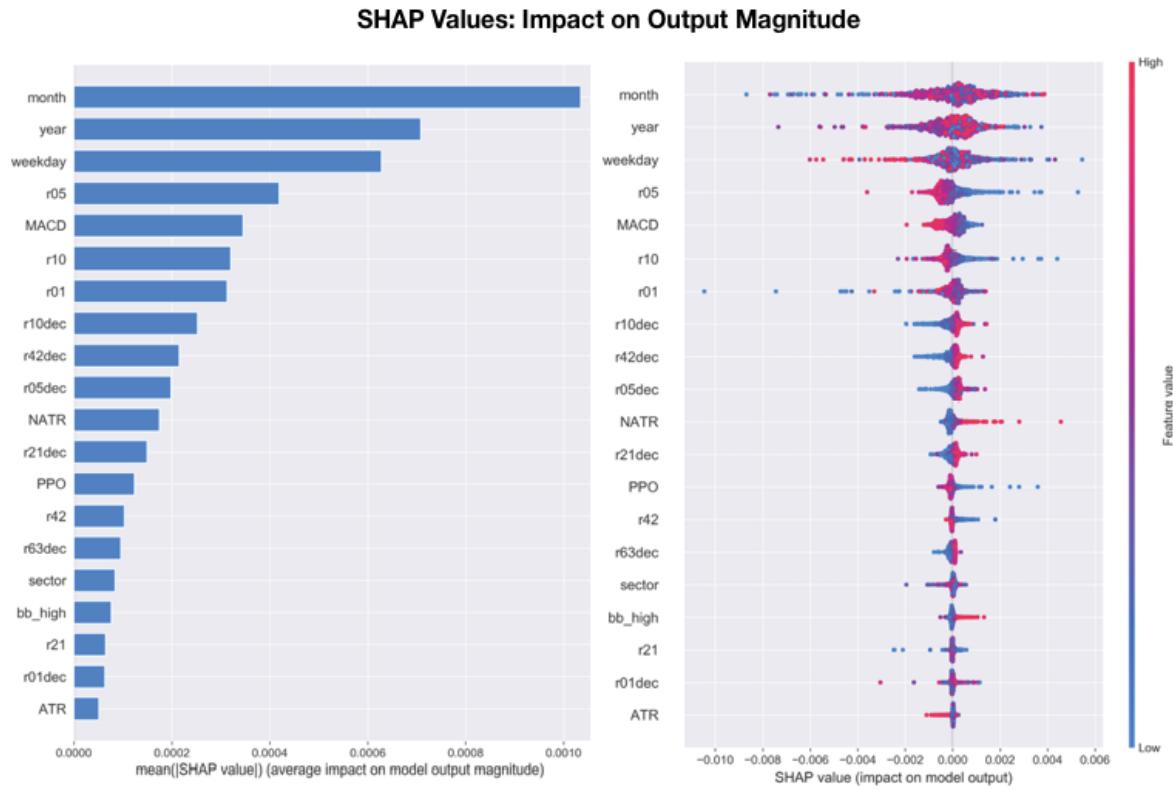


Figure 12.15: SHAP summary plots

There are some interesting differences compared to the conventional feature importance shown in *Figure 12.12*; namely, the MACD indicator turns out more important, as well as the relative return measures.

How to use force plots to explain a prediction

The force plot in the following image shows the **cumulative impact of various features and their values** on the model output, which in this case was 0.6, quite a bit higher than the base value of 0.13 (the average model output over the provided dataset). Features highlighted in red with arrows pointing to the right increase the output. The month being October is the most important feature and increases the output from 0.338 to 0.537, whereas the year being 2017 reduces the output.

Hence, we obtain a detailed breakdown of how the model arrived at a specific prediction, as shown in the following plot:

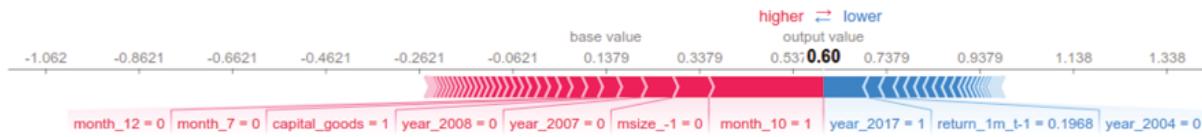


Figure 12.16: SHAP force plot

We can also compute **force plots for multiple data points** or predictions at a time and use a **clustered visualization** to gain insights into how prevalent certain influence patterns are across the dataset. The following plot shows the force plots for the first 1,000 observations rotated by 90 degrees, stacked horizontally, and ordered by the impact of different features on the outcome for the given observation.

The implementation uses hierarchical agglomerative clustering of data points on the feature SHAP values to identify these patterns, and displays the result interactively for exploratory analysis (see the notebook), as shown in the following code:

```
shap.force_plot(explainer.expected_value, shap_values[:1000,:],  
                 X_test.iloc[:1000])
```

This produces the following output:

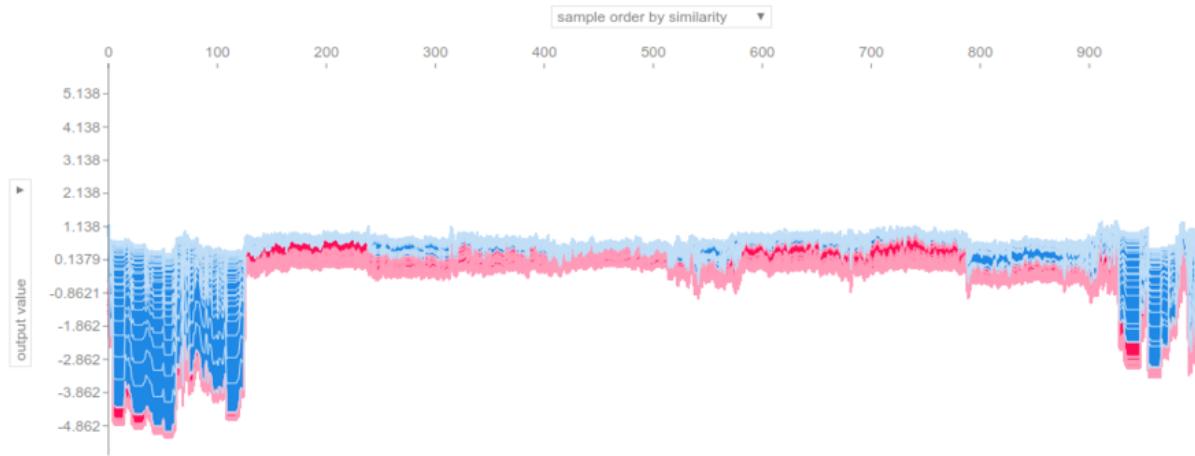


Figure 12.17: SHAP clustered force plot

How to analyze feature interaction

Lastly, SHAP values allow us to gain additional insights into the interaction effects between different features by separating these interactions from the main effects. `shap.dependence_plot` can be defined as follows:

```
shap.dependence_plot(ind='r01',
                      shap_values=shap_values,
                      features=X,
                      interaction_index='r05',
                      title='Interaction between 1- and 5-Day Returns')
```

It displays how different values for 1-month returns (on the x-axis) affect the outcome (SHAP value on the y-axis), differentiated by 3-month returns (see the following plot):

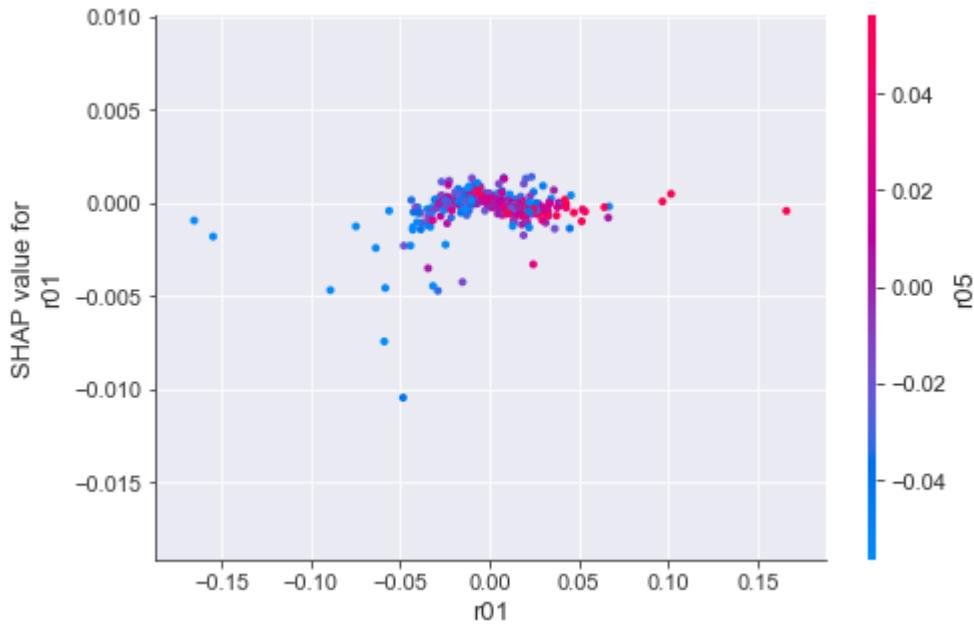


Figure 12.18: SHAP interaction plot

SHAP values provide granular feature attribution at the level of each individual prediction and enable much richer inspection of complex models through (interactive) visualization. The SHAP summary dot plot displayed earlier in this section (*Figure 12.15*) offers much more differentiated insights than a global feature-importance bar chart. Force plots of individual clustered predictions allow more detailed analysis, while SHAP dependence plots capture interaction effects and, as a result, provide more accurate and detailed results than partial dependence plots.

The limitations of SHAP values, as with any current feature-importance measure, concern the attribution of the influence of variables that are highly correlated because their similar impact can be broken down in arbitrary ways.

Backtesting a strategy based on a boosting ensemble

In this section, we'll use Zipline to evaluate the performance of a long-short strategy that enters 25 long and 25 short positions based on a daily return forecast signal. To this end, we'll select the best-performing models, generate forecasts, and design trading rules that act on these predictions.

Based on our evaluation of the cross-validation results, we'll select one or several models to generate signals for a new out-of-sample period. For this example, we'll combine predictions for the best 10 LightGBM models to reduce variance for the 1-day forecast horizon based on its solid mean quantile spread computed by Alphalens.

We just need to obtain the parameter settings for the best-performing models and then train accordingly. The notebook

`making_out_of_sample_predictions` contains the requisite code. Model training uses the hyperparameter settings of the best-performing models and data for the test period, but otherwise follows the logic used during cross-validation very closely, so we'll omit the details here.

In the notebook `backtesting_with_zipline`, we've combined the predictions of the top 10 models for the validation and test periods, as follows:

```
def load_predictions(bundle):
    predictions = (pd.read_hdf('predictions.h5', 'train/01')
                  .append(pd.read_hdf('predictions.h5', 'test/01'))
                  .drop('y_test', axis=1)))
    predictions = (predictions.loc[~predictions.index.duplicated()]
                  .iloc[:, :10]
                  .mean(1)
                  .sort_index())
```

```
.dropna()  
.to_frame('prediction'))
```

We'll use the custom ML factor that we introduced in *Chapter 8, The ML4T Workflow – From Model to Strategy Backtesting*, to import the predictions and make it accessible in a pipeline.

We'll execute `Pipeline` from the beginning of the validation period to the end of the test period. *Figure 12.19* shows (unsurprisingly) solid in-sample performance with annual returns of 27.3 percent, compared to 8.0 percent out-of-sample. The right panel of the image shows the cumulative returns relative to the S&P 500:

Metric	All	In-sample	Out-of-sample
Annual return	20.60%	27.30%	8.00%
Cumulative returns	75.00%	62.20%	7.90%
Annual volatility	19.40%	21.40%	14.40%
Sharpe ratio	1.06	1.24	0.61
Max drawdown	-17.60%	-17.60%	-9.80%
Sortino ratio	1.69	2.01	0.87
Skew	0.86	0.95	-0.16
Kurtosis	8.61	7.94	3.07

Daily value at risk	-2.40%	-2.60%	-1.80%
Daily turnover	115.10%	108.60%	127.30%
Alpha	0.18	0.25	0.05
Beta	0.24	0.24	0.22

The Sharpe ratio is 1.24 in-sample and 0.61 out-of-sample; the right panel shows the quarterly rolling value. Alpha is 0.25 in-sample versus 0.05 out-of-sample, with beta values of 0.24 and 0.22, respectively. The worst drawdown leads to losses of 17.59 percent in the second half of 2015:

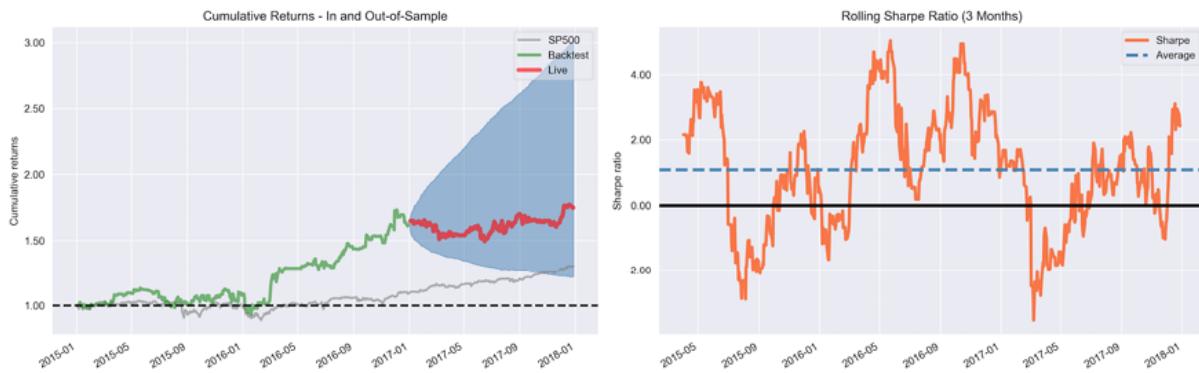


Figure 12.19: Strategy performance—cumulative returns and rolling Sharpe ratio

Long trades are slightly more profitable than short trades, which lose on average:

Summary stats	All trades	Short trades	Long trades
Total number of round_trips	22,352	11,631	10,721

Percent profitable	50.0%	48.0%	51.0%
Winning round_trips	11,131	5,616	5,515
Losing round_trips	11,023	5,935	5,088
Even round_trips	198	80	118

Lessons learned and next steps

Overall, we can see that despite using only market data in a highly liquid environment, the gradient boosting models manage to deliver predictions that are significantly better than random guesses.

Clearly, profits are anything but guaranteed, not least since we made very generous assumptions regarding transaction costs (note the high turnover).

However, there are several ways to improve on this basic framework, that is, by varying parameters from more general and strategic to more specific and tactical aspects, such as:

1. Try a different investment universe (for example, fewer liquid stocks or other assets).
2. Be creative about adding complementary data sources.
3. Engineer more sophisticated features.
4. Vary the experiment setup using, for example, longer or shorter holding and lookback periods.
5. Come up with more interesting trading rules and use several rather than a single ML signal.

Hopefully, these suggestions inspire you to build on the template we laid out and come up with an effective ML-driven trading strategy!

Boosting for an intraday strategy

We introduced **high-frequency trading (HFT)** in *Chapter 1, Machine Learning for Trading – From Idea to Execution*, as a key trend that accelerated the adoption of algorithmic strategies. There is no objective definition of HFT that pins down the properties of the activities it encompasses, including holding periods, order types (for example, passive versus aggressive), and strategies (momentum or reversion, directional or liquidity provision, and so on). However, most of the more technical treatments of HFT seem to agree that the data driving HFT activity tends to be the most granular available. Typically, this would be microstructure data directly from the exchanges such as the NASDAQ ITCH data that we introduced in *Chapter 2, Market and Fundamental Data – Sources and Techniques*, to demonstrate how it details every order placed, every execution, and every cancelation, and thus permits the reconstruction of the full limit order book, at least for equities and except for certain hidden orders.

The application of ML to HFT includes the optimization of trade execution both on official exchanges and in dark pools. ML can also be used to generate trading signals, as we will show in this section; see also Kearns and Nevyvaka (2013) for additional details and examples of how ML can add value in the HFT context.

This section uses the **AlgoSeek NASDAQ 100 dataset** from the Consolidated Feed produced by the Securities Information Processor. The data includes information on the National Best Bid and Offer quotes and trade prices at **minute bar frequency**. It also contains some features on the price dynamic, such as the number of trades at the bid or ask price, or those following positive and negative price moves at the tick level (see *Chapter 2, Market and Fundamental Data – Sources and Techniques*, for additional background and the download and preprocessing instructions in the data directory in the GitHub repository).

We'll first describe how we can engineer features for this dataset, then train a gradient boosting model to predict the volume-weighted average price for the next minute, and then evaluate the quality of the resulting trading signals.

Engineering features for high-frequency data

The dataset that AlgoSeek generously made available for this book contains over 50 variables on 100 tickers for any given day at minute frequency for the period 2013-2017. The data also covers pre-market and after-hours trading, but we'll limit this example to official market hours to the 390 minutes from 9:30 a.m. to 4:00 p.m. to somewhat restrict the size of the data, as well as to avoid having to deal with periods of irregular trading activity. See the notebook `intraday_features` for the code examples in this section.

We'll select 12 variables with over 51 million observations as raw material to create features for an ML model. This will aim predict the 1-min forward return for the volume-weighted average price:

```

MultiIndex: 51242505 entries, ('AAL', Timestamp('2014-12-22 09:30:00'))
Data columns (total 12 columns):
 #   Column  Non-Null Count   Dtype  
--- 
 0   first    51242500 non-null   float64
 1   high     51242500 non-null   float64
 2   low      51242500 non-null   float64
 3   last     51242500 non-null   float64
 4   price    49242369 non-null   float64
 5   volume   51242505 non-null   int64  
 6   up       51242505 non-null   int64  
 7   down     51242505 non-null   int64  
 8   rup      51242505 non-null   int64  
 9   rdown    51242505 non-null   int64  
 10  atask    51242505 non-null   int64  
 11  atbid    51242505 non-null   int64  
dtypes: float64(5), int64(7)
memory usage: 6.1+ GB

```

Due to the large memory footprint of the data, we only create 20 simple features, namely:

- The lagged returns for each of the last 10 minutes.
- The number of shares traded with upticks and downticks during a bar, divided by the total number of shares.
- The number of shares traded where the trade price is the same (repeated) following and upticks or downticks during a bar, divided by the total number of shares.
- The difference between the number of shares traded at the ask versus the bid price, divided by total volume during the bar.
- Several technical indicators, including the Balance of Power, the Commodity Channel Index, and the Stochastic RSI (see the *Appendix, Alpha Factor Library*, for details).

We'll make sure that we shift the data to avoid lookahead bias, as exemplified by the computation of the Money Flow Index, which

uses the TA-Lib implementation:

```
data['MFI'] = (by_ticker
    .apply(lambda x: talib.MFI(x.high,
                                x.low,
                                x['last'],
                                x.volume,
                                timeperiod=14))
    .shift())
```

The following graph shows a standalone evaluation of the individual features' predictive content using their rank correlation with the 1-minute forward returns. It reveals that the recent lagged returns are presumably the most informative variables:

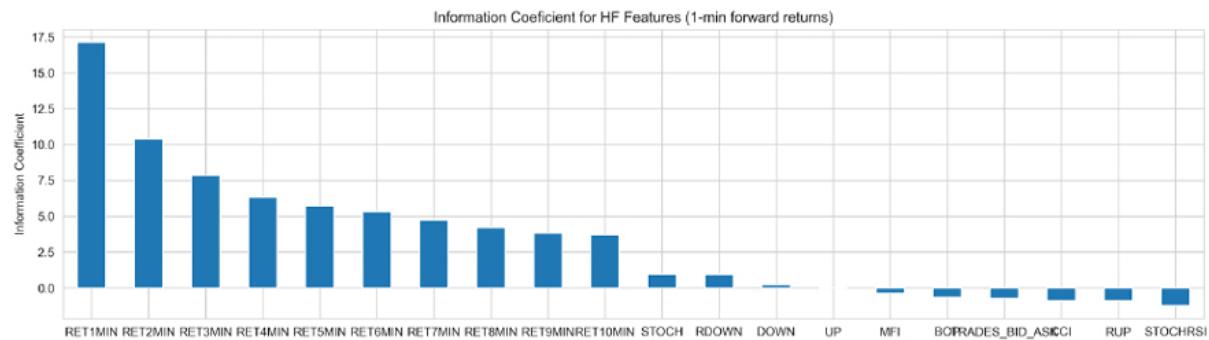


Figure 12.20: Information coefficient for high-frequency features

We can now proceed to train a gradient boosting model using these features.

Minute-frequency signals with LightGBM

To generate predictive signals for our HFT strategy, we'll train a LightGBM boosting model to predict the 1-min forward returns. The model receives 12 months of minute data during training the model and generates out-of-sample forecasts for the subsequent 21 trading

days. We'll repeat this for 24 train-test splits to cover the last 2 years of our 5-year sample.

The training process follows the preceding LightGBM example closely; see the notebook `intraday_model` for the implementation details.

One key difference is the adaptation of the custom `MultipleTimeSeriesCV` to minute frequency; we'll be referencing the `date_time` level of `MultiIndex` (see notebook for implementation). We compute the lengths of the train and test periods based on 390 observations per ticker and day as follows:

```
DAY = 390    # minutes; 6.5 hrs (9:30 - 15:59)
MONTH = 21   # trading days
n_splits = 24
cv = MultipleTimeSeriesCV(n_splits=n_splits,
                          lookahead=1,
                          test_period_length=MONTH * DAY,
                          train_period_length=12 * MONTH * DAY,
                          date_idx='date_time')
```

The large data size significantly drives up training time, so we use default settings but set the number of trees per ensemble to 250. We track the IC on the test set using the following `ic_lgbm()` custom metric definition that we pass to the model's `.train()` method.

The custom metric receives the model prediction and the binary training dataset, which we can use to compute any metric of interest; note that we set `is_higher_better` to `True` since the model minimizes loss functions by default (see the LightGBM documentation for additional information):

```
def ic_lgbm(preds, train_data):
    """Custom IC eval metric for lightgbm"""
    . . .
```

```
    is_higher_better = True
    return 'ic', spearmanr(preds, train_data.get_label())[0], is_higher_better
model = lgb.train(params=params,
                   train_set=lgb_train,
                   valid_sets=[lgb_train, lgb_test],
                   feval=ic_lgbm,
                   num_boost_round=num_boost_round,
                   early_stopping_rounds=50,
                   verbose_eval=50)
```

At 250 iterations, the validation IC is still improving for most folds, so our results are not optimal, but training already takes several hours this way. Let's now take a look at the predictive content of the signals generated by our model.

Evaluating the trading signal quality

Now, we would like to know how accurate the model's out-of-sample predictions are, and whether they could be the basis for a profitable trading strategy.

First, we compute the IC, both for all predictions and on a daily basis, as follows:

```
ic = spearmanr(cv_preds.y_test, cv_preds.y_pred)[0]
by_day = cv_preds.groupby(cv_preds.index.get_level_values('date_time'))
ic_by_day = by_day.apply(lambda x: spearmanr(x.y_test, x.y_pred)[0])
daily_ic_mean = ic_by_day.mean()
daily_ic_median = ic_by_day.median()
```

For the 2 years of rolling out-of-sample tests, we obtain a statistically significant, positive 1.90. On a daily basis, the mean IC is 1.98 and the median IC equals 1.91.

These results clearly suggest that the predictions contain meaningful information about the direction and size of short-term price movements that we could use for a trading strategy.

Next, we calculate the average and cumulative forward returns for each decile of the predictions:

```
dates = cv_preds.index.get_level_values('date_time').date
cv_preds['decile'] = (cv_preds.groupby(dates, group_keys=False)
min_ret_by_decile = cv_preds.groupby(['date_time', 'decile']).y_test.\
    .apply(lambda x: pd.qcut(x.y_pred, q=10)))
cumulative_ret_by_decile = (min_ret_by_decile
    .unstack('decile')
    .add(1)
    .cumprod()
    .sub(1))
```

Figure 12.21 displays the results. The left panel shows the average 1-min return per decile and indicates an average spread of 0.5 basis points per minute. The right panel shows the cumulative return of an equal-weighted portfolio invested in each decile, suggesting that —before transaction costs—a long-short strategy appears attractive:

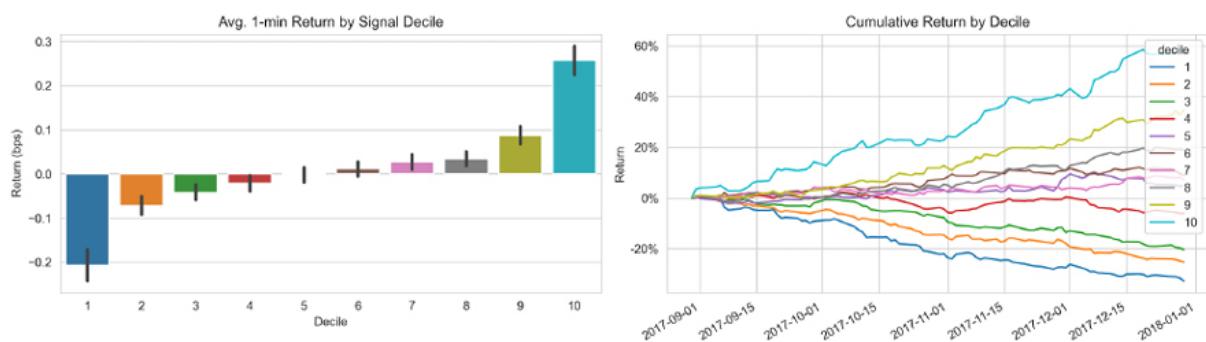


Figure 12.21: Average 1-min returns and cumulative returns by decile

The backtest with minute data is quite time-consuming, so we've omitted this step; however, feel free to experiment with Zipline or

backtrader to evaluate this strategy under more realistic assumptions regarding transaction costs or using proper risk controls.

Summary

In this chapter, we explored the gradient boosting algorithm, which is used to build ensembles in a sequential manner, adding a shallow decision tree that only uses a very small number of features to improve on the predictions that have been made. We saw how gradient boosting trees can be very flexibly applied to a broad range of loss functions, as well as offer many opportunities to tune the model to a given dataset and learning task.

Recent implementations have greatly facilitated the use of gradient boosting. They've done this by accelerating the training process and offering more consistent and detailed insights into the importance of features and the drivers of individual predictions.

Finally, we developed a simple trading strategy driven by an ensemble of gradient boosting models that was actually profitable, at least before significant trading costs. We also saw how to use gradient boosting with high-frequency data.

In the next chapter, we will turn to Bayesian approaches to machine learning.

Data-Driven Risk Factors and Asset Allocation with Unsupervised Learning

Chapter 6, The Machine Learning Process, introduced how unsupervised learning adds value by uncovering structures in data without the need for an outcome variable to guide the search process. This contrasts with supervised learning, which was the focus of the last several chapters: instead of predicting future outcomes, unsupervised learning aims to learn an informative representation of the data that helps explore new data, discover useful insights, or solve some other task more effectively.

Dimensionality reduction and clustering are the main tasks for unsupervised learning:

- **Dimensionality reduction** transforms the existing features into a new, smaller set while minimizing the loss of information. Algorithms differ by how they measure the loss of information, whether they apply linear or nonlinear transformations or which constraints they impose on the new feature set.
- **Clustering algorithms** identify and group similar observations or features instead of identifying new features. Algorithms differ in how they define the similarity of observations and their assumptions about the resulting groups.

These unsupervised algorithms are useful when a **dataset does not contain an outcome**. For instance, we may want to extract tradeable information from a large body of financial reports or news articles. In *Chapter 14, Text Data for Trading – Sentiment Analysis*, we'll use topic modeling to discover hidden themes that allow us to explore and summarize content more effectively, and identify meaningful relationships that can help us to derive signals.

The algorithms are also useful when we want to **extract information independently from an outcome**. For example, rather than using third-party industry classifications, clustering allows us to identify synthetic groupings based on the attributes of assets useful for our purposes, such as returns over a certain time horizon, exposure to risk factors, or similar fundamentals. In this chapter, we will learn how to use clustering to manage portfolio risks by identifying hierarchical relationships among asset returns.

More specifically, after reading this chapter, you will understand:

- How **principal component analysis (PCA)** and **independent component analysis (ICA)** perform linear dimensionality reduction
- Identifying data-driven risk factors and eigenportfolios from asset returns using PCA
- Effectively visualizing nonlinear, high-dimensional data using manifold learning
- Using T-SNE and UMAP to explore high-dimensional image data
- How k-means, hierarchical, and density-based clustering algorithms work

- Using agglomerative clustering to build robust portfolios with hierarchical risk parity



You can find the code samples for this chapter and links to additional resources in the corresponding directory of the GitHub repository. The notebooks include color versions of the images.

Dimensionality reduction

In linear algebra terms, the features of a dataset create a **vector space** whose dimensionality corresponds to the number of linearly independent rows or columns, whichever is larger. Two columns are linearly dependent when they are perfectly correlated so that one can be computed from the other using the linear operations of addition and multiplication.

In other words, they are parallel vectors that represent the same direction rather than different ones in the data and thus only constitute a single dimension. Similarly, if one variable is a linear combination of several others, then it is an element of the vector space created by those columns and does not add a new dimension of its own.

The number of dimensions of a dataset matters because each new dimension can add a signal concerning an outcome. However, there is also a downside known as the **curse of dimensionality**: as the number of independent features grows while the number of observations remains constant, the average distance between data points also grows, and the density of the feature space drops exponentially, with dramatic implications for **machine learning**

(ML). **Prediction becomes much harder** when observations are more distant, that is, different from each other. Alternative data sources, like text and images, typically are of high dimensionality, but they generally affect models that rely on a large number of features. The next section addresses the resulting challenges.

Dimensionality reduction seeks to **represent the data more efficiently** by using fewer features. To this end, algorithms project the data to a lower-dimensional space while discarding any variation that is not informative, or by identifying a lower-dimensional subspace or manifold on or near to where the data lives.

A **manifold** is a space that locally resembles Euclidean space. One-dimensional manifolds include a line or a circle, but not the visual representation of the number eight due to the crossing point.

The manifold hypothesis maintains that high-dimensional data often resides in a lower-dimensional space, which, if identified, permits a faithful representation of the data in this subspace. Refer to Fefferman, Mitter, and Narayanan (2016) for background information and the description of an algorithm that tests this hypothesis.

Dimensionality reduction, therefore, compresses the data by finding a different, smaller set of variables that capture what matters most in the original features to minimize the loss of information.

Compression helps counter the curse of dimensionality, economizes on memory, and permits the visualization of salient aspects of higher-dimensional data that is otherwise very difficult to explore.

Dimensionality reduction algorithms differ by the constraints they impose on the new variables and how they aim to minimize the loss of information (see Burges 2010 for an excellent overview):

- **Linear algorithms** like PCA and ICA constrain the new variables to be linear combinations of the original features; for example, hyperplanes in a lower-dimensional space. Whereas PCA requires the new features to be uncorrelated, ICA goes further and imposes statistical independence, implying the absence of both linear and nonlinear relationships.
- **Nonlinear algorithms** are not restricted to hyperplanes and can capture a more complex structure in the data. However, given the infinite number of options, the algorithms still need to make assumptions in order to arrive at a solution. Later in this section, we will explain how **t-distributed Stochastic Neighbor Embedding (t-SNE)** and **Uniform Manifold Approximation and Projection (UMAP)** are very useful to visualize higher-dimensional data. *Figure 13.1* illustrates how manifold learning identifies a two-dimensional subspace in the three-dimensional feature space. (The notebook `manifold_learning` illustrates the use of additional algorithms, including local linear embedding.)

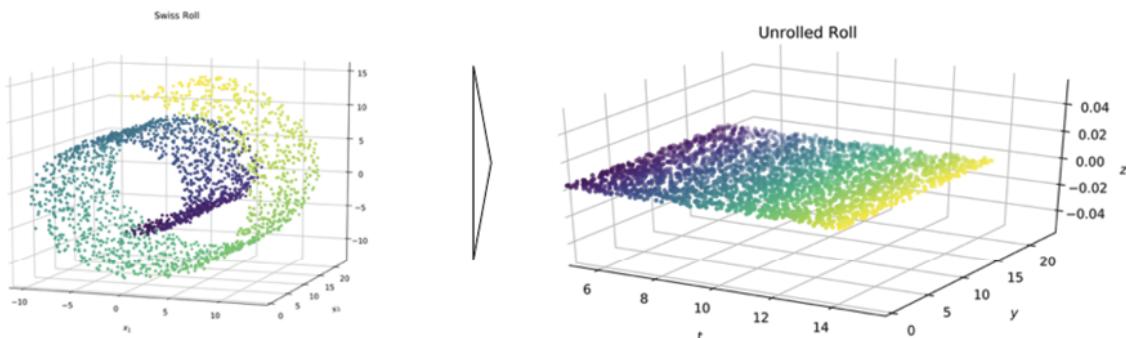


Figure 13.1: Nonlinear dimensionality reduction

The curse of dimensionality

An increase in the number of dimensions of a dataset means that there are more entries in the vector of features that represents each

observation in the corresponding Euclidean space.

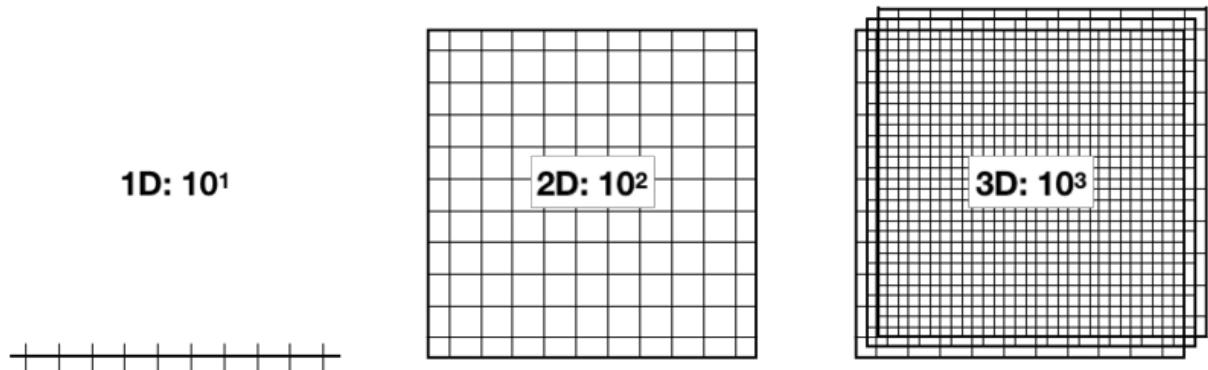
We measure the distance in a vector space using the Euclidean distance, also known as the L^2 norm, which we applied to the vector of linear regression coefficients to train a regularized ridge regression.

The Euclidean distance between two n -dimensional vectors with Cartesian coordinates $p = (p_1, p_2, \dots, p_n)$ and $q = (q_1, q_2, \dots, q_n)$ is computed using the familiar formula developed by Pythagoras:

$$d(p, q) = \sqrt{\sum_{i=1}^n (p_i - q_i)^2}$$

Therefore, each new dimension adds a non-negative term to the sum so that the distance increases with the number of dimensions for distinct vectors. In other words, as the number of features grows for a given number of observations, the feature space becomes increasingly sparse, that is, less dense or emptier. On the flip side, the lower data density requires more observations to keep the average distance between the data points the same.

Figure 13.2 illustrates the exponential growth in the number of data points needed to maintain the average distance among observations as the number of dimensions increases. 10 points uniformly distributed on a line correspond to 10^2 points in two dimensions and 10^3 points in three dimensions in order to keep the density constant.



The number of features required to keep average distance constant grows exponentially with the number of dimensions.

Figure 13.2: The number of features required to keep the average distance constant grows exponentially with the number of dimensions

The notebook `the_curse_of_dimensionality` in the GitHub repository folder for this section simulates how the average and minimum distances between data points increase as the number of dimensions grows (see *Figure 13.3*).

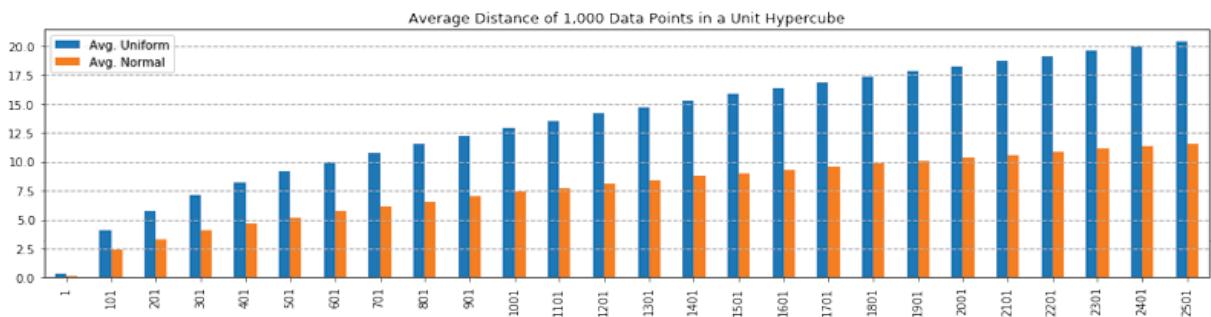


Figure 13.3: Average distance of 1,000 data points in a unit hypercube

The **simulation** randomly samples up to 2,500 features in the range $[0, 1]$ from an uncorrelated uniform or a correlated normal distribution. The average distance between data points increases to over 11 times the unitary feature range for the normal distribution, and to over 20 times in the (extreme) case of an uncorrelated uniform distribution.

When the **distance between observations** grows, supervised ML becomes more difficult because predictions for new samples are less likely to be based on learning from similar training features. Put simply, the number of possible unique rows grows exponentially as the number of features increases, making it much harder to efficiently sample the space. Similarly, the complexity of the functions learned by flexible algorithms that make fewer assumptions about the actual relationship grows exponentially with the number of dimensions.

Flexible algorithms include the tree-based models we saw in *Chapter 11, Random Forests – A Long-Short Strategy for Japanese Stocks*, and *Chapter 12, Boosting Your Trading Strategy*. They also include the deep neural networks that we will cover later in the book, starting with *Chapter 16, Word Embeddings for Earnings Calls and SEC Filings*. The variance of these algorithms increases as **more dimensions add opportunities to overfit** to noise, resulting in poor generalization performance.

Dimensionality reduction leverages the fact that, in practice, features are often correlated or exhibit little variation. If so, it can compress data without losing much of the signal and complements the use of regularization to manage prediction error due to variance and model complexity.

The critical question that we take on in the following section then becomes: what are the best ways to find a lower-dimensional representation of the data?

Linear dimensionality reduction

Linear dimensionality reduction algorithms compute linear combinations that **translate, rotate, and rescale the original features** to capture significant variations in the data, subject to constraints on the characteristics of the new features.

PCA, invented in 1901 by Karl Pearson, finds new features that reflect directions of maximal variance in the data while being mutually uncorrelated. ICA, in contrast, originated in signal processing in the 1980s with the goal of separating different signals while imposing the stronger constraint of statistical independence.

This section introduces these two algorithms and then illustrates how to apply PCA to asset returns in order to learn risk factors from the data, and build so-called eigenportfolios for systematic trading strategies.

Principal component analysis

PCA finds linear combinations of the existing features and uses these principal components to represent the original data. The number of components is a hyperparameter that determines the target dimensionality and can be, at most, equal to the lesser of the number of rows or columns.

PCA aims to capture most of the variance in the data to make it easy to recover the original features and ensures that each component adds information. It reduces dimensionality by projecting the original data into the principal component space.

The PCA algorithm works by identifying a sequence of components, each of which aligns with the direction of maximum variance in the data after accounting for variation captured by previously computed components. The sequential optimization ensures that new

components are not correlated with existing components and produces an orthogonal basis for a vector space.

This new basis is a rotation of the original basis, such that the new axes point in the direction of successively decreasing variance. The decline in the amount of variance of the original data explained by each principal component reflects the extent of correlation among the original features. In other words, the share of components that captures, for example, 95 percent of the original variation provides insight into the linearly independent information in the original data.

Visualizing PCA in 2D

Figure 13.4 illustrates several aspects of PCA for a two-dimensional random dataset (refer to the notebook `pca_key_ideas`):

- The left panel shows how the first and second principal components align with the **directions of maximum variance** while being orthogonal.
- The central panel shows how the first principal component minimizes the **reconstruction error**, measured as the sum of the distances between the data points and the new axis.
- The right panel illustrates **supervised OLS** (refer to *Chapter 7, Linear Models – From Risk Factors to Return Forecasts*), which approximates the outcome (x_2) by a line computed from the single feature x_1 . The vertical lines highlight how OLS minimizes the distance along the outcome axis, whereas PCA minimizes the distances that are orthogonal to the hyperplane.

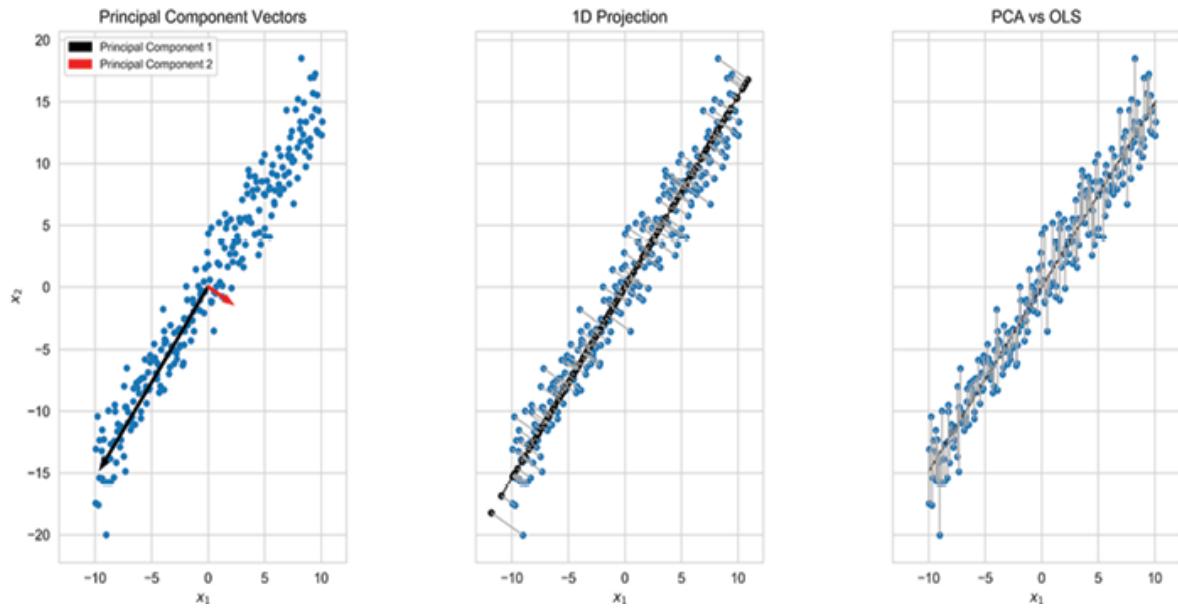


Figure 13.4: PCA in 2D from various perspectives

Key assumptions made by PCA

PCA makes several assumptions that are important to keep in mind. These include:

- High variance implies a high signal-to-noise ratio.
- The data is standardized so that the variance is comparable across features.
- Linear transformations capture the relevant aspects of the data.
- Higher-order statistics beyond the first and second moments do not matter, which implies that the data has a normal distribution.

The emphasis on the first and second moments aligns with standard risk/return metrics, but the normality assumption may conflict with the characteristics of market data. Market data often exhibits skew or kurtosis (fat tails) that differ from those of the normal distribution and will not be taken into account by PCA.

How the PCA algorithm works

The algorithm finds vectors to create a hyperplane of target dimensionality that minimizes the reconstruction error, measured as the sum of the squared distances of the data points to the plane. As illustrated previously, this goal corresponds to finding a sequence of vectors that align with directions of maximum retained variance given the other components, while ensuring all principal components are mutually orthogonal.

In practice, the algorithm solves the problem either by computing the eigenvectors of the covariance matrix or by using the **singular value decomposition (SVD)**.

We illustrate the computation using a randomly generated three-dimensional ellipse with 100 data points, as shown in the left panel of *Figure 13.5*, including the two-dimensional hyperplane defined by the first two principal components. (Refer to the notebook `the_math_behind_pca` for the code samples in the following three sections.)

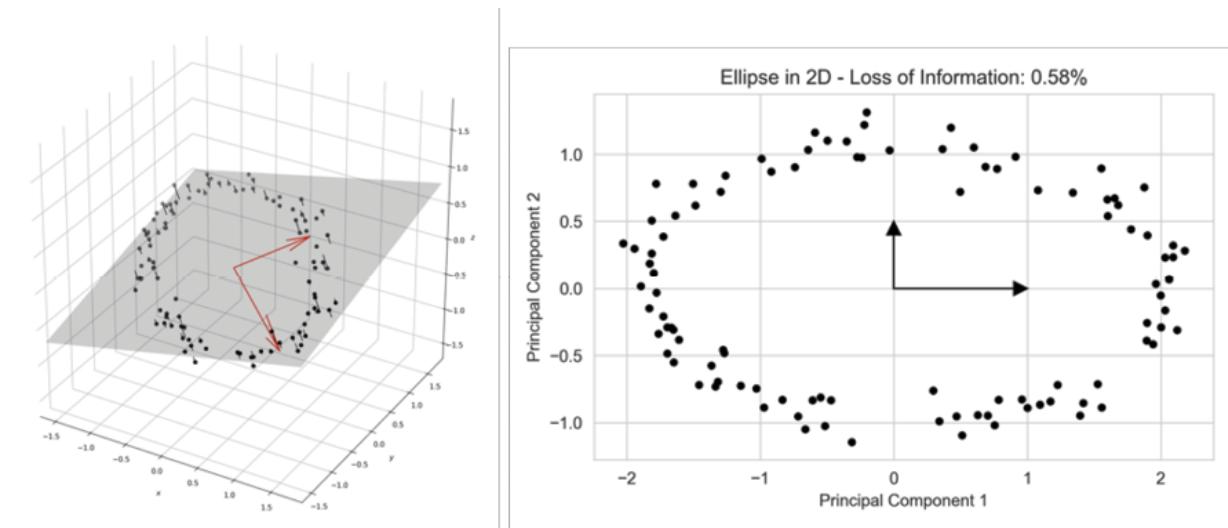


Figure 13.5: Visual representation of dimensionality reduction from 3D to 2D

PCA based on the covariance matrix

We first compute the principal components using the square covariance matrix with the pairwise sample covariances for the features x_i, x_j , $i, j = 1, \dots, n$ as entries in row i and column j :

$$cov_{i,j} = \frac{\sum_{k=1}^N (x_{ik} - \bar{x}_i)(x_{jk} - \bar{x}_j)}{N - 1}$$

For a square matrix M of n dimension, we define the eigenvectors ω_i and eigenvalues λ_i , $i=1, \dots, n$ as follows:

$$M\omega_i = \lambda_i\omega_i$$

Therefore, we can represent the matrix M using eigenvectors and eigenvalues, where W is a matrix that contains the eigenvectors as column vectors, and L is a matrix that contains λ_i as diagonal entries (and 0s otherwise). We define the **eigendecomposition** as:

$$M = WLW^{-1}$$

Using NumPy, we implement this as follows, where the pandas DataFrame data contains the 100 data points of the ellipse:

```
# compute covariance matrix:
cov = np.cov(data.T) # expects variables in rows by default
cov.shape
(3, 3)
```

Next, we calculate the eigenvectors and eigenvalues of the covariance matrix. The eigenvectors contain the principal components (where the sign is arbitrary):

```

eigen_values, eigen_vectors = eig(cov)
eigen_vectors
array([[ 0.71409739, -0.66929454, -0.20520656],
       [-0.70000234, -0.68597301, -0.1985894 ],
       [ 0.00785136, -0.28545725,  0.95835928]])

```

We can compare the result with the result obtained from sklearn and find that they match in absolute terms:

```

pca = PCA()
pca.fit(data)
C = pca.components_.T # columns = principal components
C
array([[ 0.71409739,  0.66929454,  0.20520656],
       [-0.70000234,  0.68597301,  0.1985894 ],
       [ 0.00785136,  0.28545725, -0.95835928]])
np.allclose(np.abs(C), np.abs(eigen_vectors))
True

```

We can also **verify the eigendecomposition**, starting with the diagonal matrix L that contains the eigenvalues:

```

# eigenvalue matrix
ev = np.zeros((3, 3))
np.fill_diagonal(ev, eigen_values)
ev # diagonal matrix
array([[1.92923132, 0.          , 0.          ],
       [0.          , 0.55811089, 0.          ],
       [0.          , 0.          , 0.00581353]])

```

We find that the result does indeed hold:

```

decomposition = eigen_vectors.dot(ev).dot(inv(eigen_vectors))
np.allclose(cov, decomposition)

```

PCA using the singular value decomposition

Next, we'll take a look at the alternative computation using the SVD. This algorithm is slower when the number of observations is greater than the number of features (which is the typical case) but yields better **numerical stability**, especially when some of the features are strongly correlated (which is often the reason to use PCA in the first place).

SVD generalizes the eigendecomposition that we just applied to the square and symmetric covariance matrix to the more general case of $m \times n$ rectangular matrices. It has the form shown at the center of the following figure. The diagonal values of Σ are the singular values, and the transpose of V^* contains the principal components as column vectors.

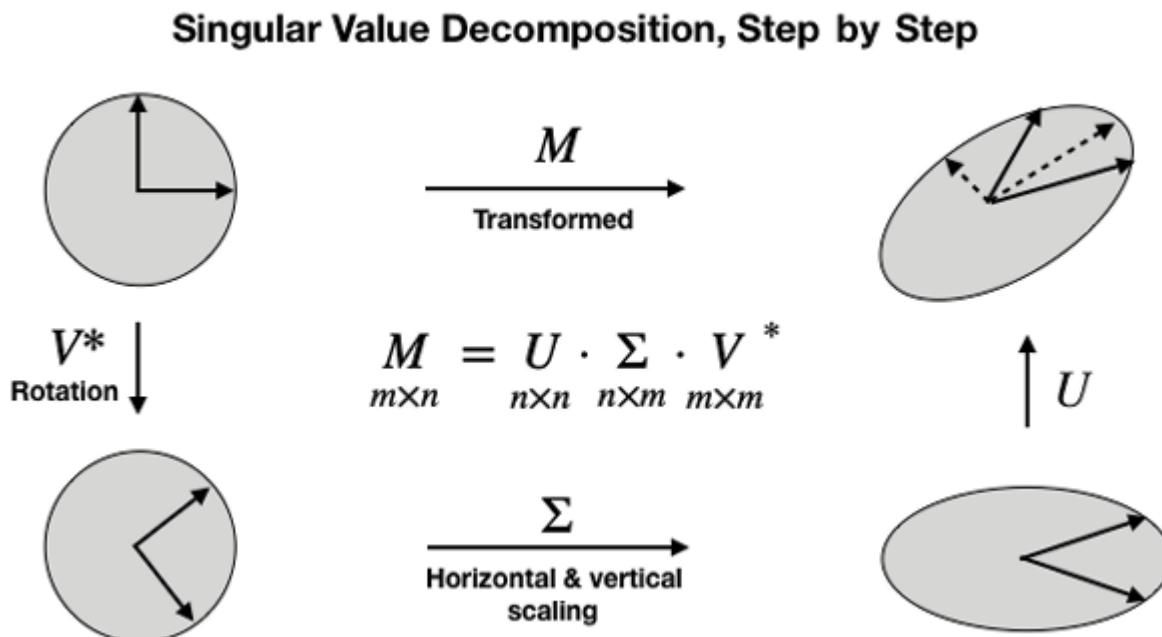


Figure 13.6: The SVD decomposed

In this case, we need to make sure our data is centered with mean zero (the computation of the covariance earlier took care of this):

```
n_features = data.shape[1]
data_ = data - data.mean(axis=0)
```

Using the centered data, we compute the SVD:

```
U, s, Vt = svd(data_)
U.shape, s.shape, Vt.shape
((100, 100), (3,), (3, 3))
```

We can convert the vector `s`, which contains only singular values, into an $n \times m$ matrix and show that the decomposition works:

```
S = np.zeros_like(data_)
S[:n_features, :n_features] = np.diag(s)
S.shape
(100, 3)
```

We find that the decomposition does indeed reproduce the standardized data:

```
np.allclose(data_, U.dot(S).dot(Vt))
True
```

Lastly, we confirm that the columns of the transpose of V^* contain the principal components:

```
np.allclose(np.abs(C), np.abs(Vt.T))
```

In the next section, we will demonstrate how sklearn implements PCA.

PCA with sklearn

The `sklearn.decomposition.PCA` implementation follows the standard API based on the `fit()` and `transform()` methods that compute the desired number of principal components and project the data into the component space, respectively. The convenience method `fit_transform()` accomplishes this in a single step.

PCA offers three different algorithms that can be specified using the `svd_solver` parameter:

- **full** computes the exact SVD using the LAPACK solver provided by `scipy`.
- **arpack** runs a truncated version suitable for computing less than the full number of components.
- **randomized** uses a sampling-based algorithm that is more efficient when the dataset has more than 500 observations and features, and the goal is to compute less than 80 percent of the components.
- **auto** also randomizes where it is most efficient; otherwise, it uses the full SVD.

Please view the references on GitHub for algorithmic implementation details.

Other key configuration parameters of the PCA object are:

- **n_components**: Compute all principal components by passing `None` (the default), or limit the number to `int`. For `svd_solver=full`, there are two additional options: a `float` in the interval $[0, 1]$ computes the number of components required to retain the corresponding share of the variance in the data, and the option `mle` estimates the number of dimensions using the maximum likelihood.

- **whiten**: If `True`, it standardizes the component vectors to unit variance, which, in some cases, can be useful in a predictive model (the default is `False`).

To compute the first two principal components of the three-dimensional ellipsis and project the data into the new space, use `fit_transform()`:

```
pca2 = PCA(n_components=2)
projected_data = pca2.fit_transform(data)
projected_data.shape
(100, 2)
```

The explained variance of the first two components is very close to 100 percent:

```
pca2.explained_variance_ratio_
array([0.77381099, 0.22385721])
```

Figure 13.5 shows the projection of the data into the new two-dimensional space.

Independent component analysis

ICA is another linear algorithm that identifies a new basis to represent the original data but pursues a different objective than PCA. Refer to Hyvärinen and Oja (2000) for a detailed introduction.

ICA emerged in signal processing, and the problem it aims to solve is called **blind source separation**. It is typically framed as the cocktail party problem, where a given number of guests are speaking at the same time so that a single microphone records overlapping signals. ICA assumes there are as many different

microphones as there are speakers, each placed at different locations so that they record a different mix of signals. ICA then aims to recover the individual signals from these different recordings.

In other words, there are n original signals and an unknown square mixing matrix A that produces an n -dimensional set of m observations so that

$$\begin{matrix} X \\ n \times m \end{matrix} = \begin{matrix} A \\ n \times n \end{matrix} \begin{matrix} s \\ n \times m \end{matrix}$$

The goal is to find the matrix $W = A^{-1}$ that untangles the mixed signals to recover the sources.

The ability to uniquely determine the matrix W hinges on the non-Gaussian distribution of the data. Otherwise, W could be rotated arbitrarily given the multivariate normal distribution's symmetry under rotation. Furthermore, ICA assumes the mixed signal is the sum of its components and is, therefore, unable to identify Gaussian components because their sum is also normally distributed.

ICA assumptions

ICA makes the following critical assumptions:

- The sources of the signals are statistically independent
- Linear transformations are sufficient to capture the relevant information
- The independent components do not have a normal distribution
- The mixing matrix A can be inverted

ICA also requires the data to be centered and whitened, that is, to be mutually uncorrelated with unit variance. Preprocessing the data

using PCA, as outlined earlier, achieves the required transformations.

The ICA algorithm

`FastICA`, used by `sklearn`, is a fixed-point algorithm that uses higher-order statistics to recover the independent sources. In particular, it maximizes the distance to a normal distribution for each component as a proxy for independence.

An alternative algorithm called `InfoMax` minimizes the mutual information between components as a measure of statistical independence.

ICA with `sklearn`

The ICA implementation by `sklearn` uses the same interface as PCA, so there is little to add. Note that there is no measure of explained variance because ICA does not compute components successively. Instead, each component aims to capture the independent aspects of the data.

Manifold learning – nonlinear dimensionality reduction

Linear dimensionality reduction projects the original data onto a lower-dimensional hyperplane that aligns with informative directions in the data. The focus on linear transformations simplifies the computation and echoes common financial metrics, such as PCA's goal to capture the maximum variance.

However, linear approaches will naturally ignore signals reflected in nonlinear relationships in the data. Such relationships are very important in alternative datasets containing, for example, image or text data. Detecting such relationships during exploratory analysis can provide important clues about the data's potential signal content.

In contrast, the **manifold hypothesis** emphasizes that high-dimensional data often lies on or near a lower-dimensional nonlinear manifold that is embedded in the higher-dimensional space. The two-dimensional Swiss roll displayed in *Figure 13.1* (at the beginning of this chapter) illustrates such a topological structure. Manifold learning aims to find the manifold of intrinsic dimensionality and then represent the data in this subspace. A simplified example uses a road as a one-dimensional manifold in a three-dimensional space and identifies data points using house numbers as local coordinates.

Several techniques approximate a lower-dimensional manifold. One example is **locally linear embedding (LLE)**, which was invented by Lawrence Saul and Sam Roweis (2000) and used to "unroll" the Swiss roll shown in *Figure 13.1* (view the examples in the `manifold_learning_lle` notebook).

For each data point, LLE identifies a given number of nearest neighbors and computes weights that represent each point as a linear combination of its neighbors. It finds a lower-dimensional embedding by linearly projecting each neighborhood on global internal coordinates on the lower-dimensional manifold and can be thought of as a sequence of PCA applications.

Visualization requires that the reduction is at least three dimensions, possibly below the intrinsic dimensionality, and poses the **challenge of faithfully representing both the local and global structure**. This

challenge relates to the curse of dimensionality; that is, while the volume of a sphere expands exponentially with the number of dimensions, the lower-dimensional space available to represent high-dimensional data is much more limited. For instance, in 12 dimensions, there can be 13 equidistant points; however, in two dimensions, there can only be 3 that form a triangle with sides of equal length. Therefore, accurately reflecting the distance of one point to its high-dimensional neighbors in lower dimensions risks distorting the relationships among all other points. The result is the **crowding problem**: to maintain global distances, local points may need to be placed too closely together.

The next two sections cover techniques that have allowed us to make progress in addressing the crowding problem for the visualization of complex datasets. We will use the fashion MNIST dataset, which is a more sophisticated alternative to the classic handwritten digit MNIST benchmark data used for computer vision. It contains 60,000 training and 10,000 test images of fashion objects in 10 classes (take a look at the sample images in the notebook `manifold_learning_intro`). The goal of a manifold learning algorithm for this data is to detect whether the classes lie on distinct manifolds to facilitate their recognition and differentiation.

t-distributed Stochastic Neighbor Embedding

t-SNE is an award-winning algorithm, developed by Laurens van der Maaten and Geoff Hinton in 2008, to detect patterns in high-dimensional data. It takes a probabilistic, nonlinear approach to locate data on several different but related low-dimensional manifolds. The algorithm emphasizes keeping similar points together in low dimensions as opposed to maintaining the distance

between points that are apart in high dimensions, which results from algorithms like PCA that minimize squared distances.

The algorithm proceeds by **converting high-dimensional distances into (conditional) probabilities**, where high probabilities imply low distance and reflect the likelihood of sampling two points based on similarity. It accomplishes this by, first, positioning a normal distribution over each point and computing the density for a point and each neighbor, where the `perplexity` parameter controls the effective number of neighbors. In the second step, it arranges points in low dimensions and uses similarly computed low-dimensional probabilities to match the high-dimensional distribution. It measures the difference between the distributions using the Kullback-Leibler divergence, which puts a high penalty on misplacing similar points in low dimensions.

The low-dimensional probabilities use a Student's t-distribution with one degree of freedom because it has fatter tails that reduce the penalty of misplacing points that are more distant in high dimensions to manage the crowding problem.

The upper panels in *Figure 13.7* show how t-SNE is able to differentiate between the FashionMNIST image classes. A higher perplexity value increases the number of neighbors used to compute the local structure and gradually results in more emphasis on global relationships. (Refer to the repository for a high-resolution color version of this figure.)

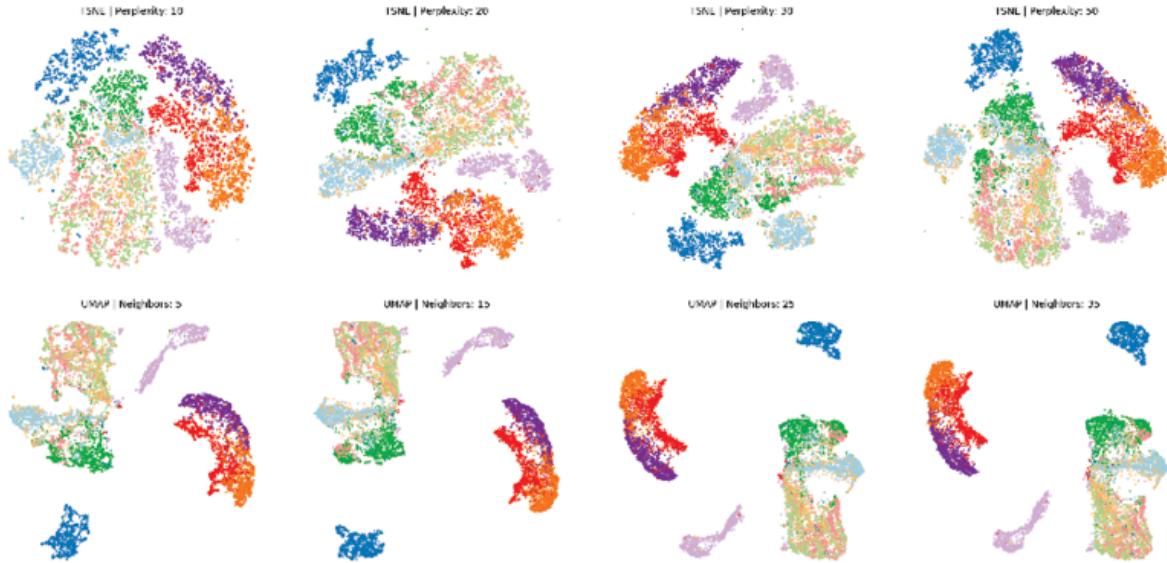


Figure 13.7: t-SNE and UMAP visualization of Fashion MNIST image data for different hyperparameters

t-SNE is the current state of the art in high-dimensional data visualization. Weaknesses include the computational complexity that scales quadratically in the number n of points because it evaluates all pairwise distances, but a subsequent tree-based implementation has reduced the cost to $n \log n$.

Unfortunately, t-SNE does not facilitate the projection of new data points into the low-dimensional space. The compressed output is not a very useful input for distance- or density-based cluster algorithms because t-SNE treats small and large distances differently.

Uniform Manifold Approximation and Projection

UMAP is a more recent algorithm for visualization and general dimensionality reduction. It assumes the data is uniformly distributed on a locally connected manifold and looks for the closest low-dimensional equivalent using fuzzy topology. It uses a

`neighbors` parameter, which impacts the result in a similar way to `perplexity` in the preceding section.

It is faster and hence scales better to large datasets than t-SNE and sometimes preserves the global structure better than t-SNE. It can also work with different distance functions, including cosine similarity, which is used to measure the distance between word count vectors.

The preceding figure illustrates how UMAP does indeed move the different clusters further apart, whereas t-SNE provides more granular insight into the local structure.

The notebook also contains interactive Plotly visualizations for each of the algorithms that permit the exploration of the labels and identify which objects are placed close to each other.

PCA for trading

PCA is useful for algorithmic trading in several respects, including:

- The data-driven derivation of risk factors by applying PCA to asset returns
- The construction of uncorrelated portfolios based on the principal components of the correlation matrix of asset returns

We will illustrate both of these applications in this section.

Data-driven risk factors

In *Chapter 7, Linear Models – From Risk Factors to Return Forecasts*, we explored **risk factor models** used in quantitative finance to capture the main drivers of returns. These models explain differences in returns on assets based on their exposure to systematic risk factors and the rewards associated with these factors. In particular, we explored the **Fama-French approach**, which specifies factors based on prior knowledge about the empirical behavior of average returns, treats these factors as observable, and then estimates risk model coefficients using linear regression.

An alternative approach treats risk factors as **latent variables** and uses factor analytic techniques like PCA to simultaneously learn the factors from data and estimate how they drive returns. In this section, we will demonstrate how this method derives factors in a purely statistical or data-driven way with the advantage of not requiring *ex ante* knowledge of the behavior of asset returns (see the notebook `pca_and_risk_factor_models` for more details).

Preparing the data – top 350 US stocks

We will use the Quandl stock price data and select the daily adjusted close prices of the 500 stocks with the largest market capitalization and data for the 2010-2018 period. We will then compute the daily returns as follows:

```
idx = pd.IndexSlice
with pd.HDFStore('../data/assets.h5') as store:
    stocks = store['us_equities/stocks'].marketcap.nlargest(500)
    returns = (store['quandl/wiki/prices']
               .loc[idx['2010': '2018', stocks.index], 'adj_close']
               .unstack('ticker')
               .pct_change())
```

We obtain 351 stocks and returns for over 2,000 trading days:

```
returns.info()  
DatetimeIndex: 2072 entries, 2010-01-04 to 2018-03-27  
Columns: 351 entries, A to ZTS
```

PCA is sensitive to outliers, so we winsorize the data at the 2.5 percent and 97.5 percent quantiles, respectively:

PCA does not permit missing data, so we will remove any stocks that do not have data for at least 95 percent of the time period. Then, in a second step, we will remove trading days that do not have observations on at least 95 percent of the remaining stocks:

```
returns = returns.dropna(thresh=int(returns.shape[0] * .95), axis=1)  
returns = returns.dropna(thresh=int(returns.shape[1] * .95))
```

We are left with 315 equity return series covering a similar period:

```
returns.info()  
DatetimeIndex: 2071 entries, 2010-01-05 to 2018-03-27  
Columns: 315 entries, A to LYB
```

We impute any remaining missing values using the average return for any given trading day:

```
daily_avg = returns.mean(1)  
returns = returns.apply(lambda x: x.fillna(daily_avg))
```

Running PCA to identify the key return drivers

Now we are ready to fit the principal components model to the asset returns using default parameters to compute all of the components using the full SVD algorithm:

```
pca = PCA(n_components='mle')
pca.fit(returns)
```

We find that the most important factor explains around 55 percent of the daily return variation. The dominant factor is usually interpreted as "the market," whereas the remaining factors can be interpreted as industry or style factors in line with our discussions in *Chapter 5, Portfolio Optimization and Performance Evaluation*, and *Chapter 7, Linear Models – From Risk Factors to Return Forecasts*, depending on the results of a closer inspection (please refer to the next example).

The plot on the right of *Figure 13.8* shows the cumulative explained variance and indicates that around 10 factors explain 60 percent of the returns of this cross-section of stocks.

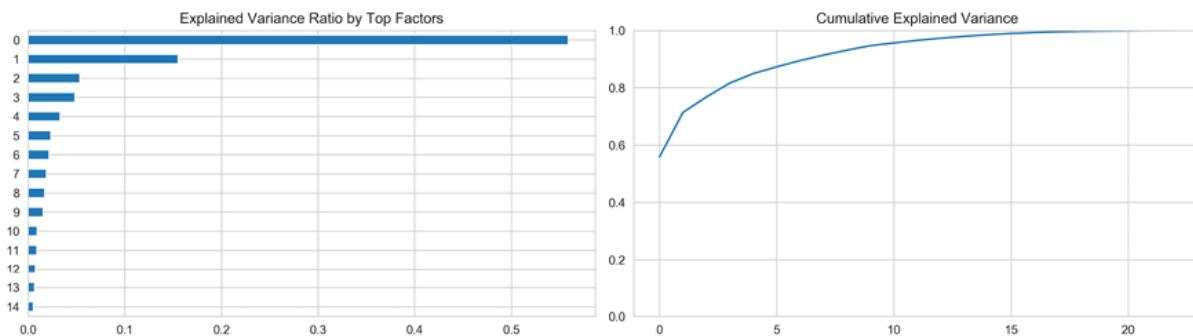


Figure 13.8: (Cumulative) explained return variance by PCA-based risk factors

The notebook contains a **simulation** for a broader cross-section of stocks and the longer 2000-2018 time period. It finds that, on average, the first three components explained 40 percent, 10 percent, and 5 percent of 500 randomly selected stocks, as shown in *Figure 13.9*:

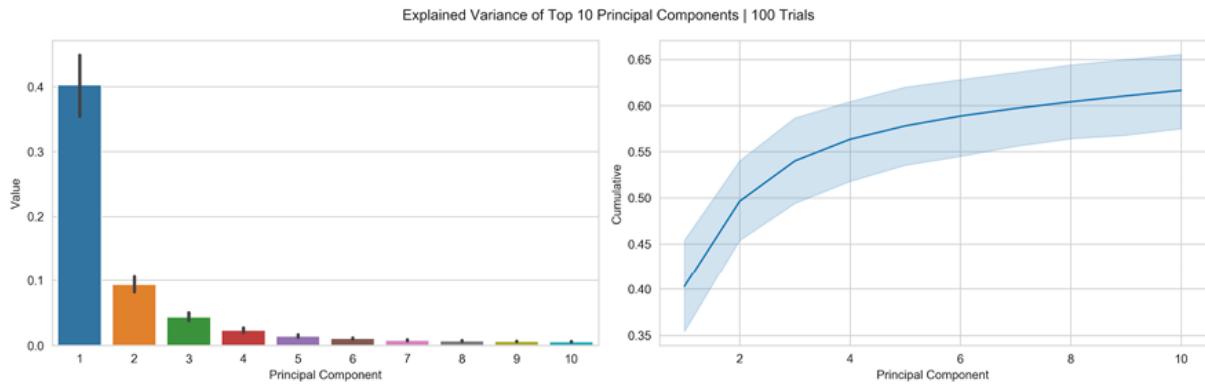


Figure 13.9: Explained variance of the top 10 principal components—100 trials

The cumulative plot shows a typical "elbow" pattern that can help to identify a suitable target dimensionality as the number of components beyond which additional components add less incremental value.

We can select the top two principal components to verify that they are indeed uncorrelated:

```
risk_factors = pd.DataFrame(pca.transform(returns)[:, :2],
                           columns=['Principal Component 1',
                                     'Principal Component 2'],
                           index=returns.index)
(risk_factors['Principal Component 1']
 .corr(risk_factors['Principal Component 2']))
7.773256996252084e-15
```

Moreover, we can plot the time series to highlight how each factor captures different volatility patterns, as shown in the following figure:

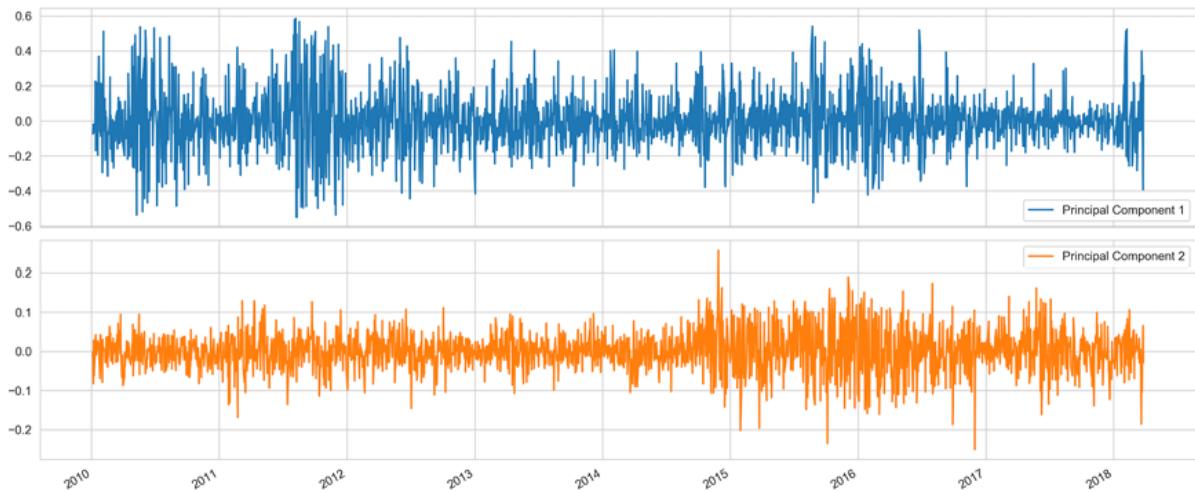


Figure 13.10: Return volatility patterns captured by the first two principal components

A risk factor model would employ a subset of the principal components as features to predict future returns, similar to our approach in *Chapter 7, Linear Models – From Risk Factors to Return Forecasts*.

Eigenportfolios

Another application of PCA involves the covariance matrix of the normalized returns. The principal components of the correlation matrix capture most of the covariation among assets in descending order and are mutually uncorrelated. Moreover, we can use standardized principal components as portfolio weights. You can find the code example for this section in the notebook

`pca_and_eigen_portfolios`.

Let's use the 30 largest stocks with data for the 2010-2018 period to facilitate the exposition:

```
idx = pd.IndexSlice
with pd.HDFStore('../data/assets.h5') as store:
```

```
stocks = store['us_equities/stocks'].marketcap.nlargest(30)
returns = (store['quandl/wiki/prices']
           .loc[idx['2010': '2018', stocks.index], 'adj_close']
           .unstack('ticker')
           .pct_change())
```

We again winsorize and also normalize the returns:

```
normed_returns = scale(returns
                       .clip(lower=returns.quantile(q=.025),
                             upper=returns.quantile(q=.975),
                             axis=1)
                       .apply(lambda x: x.sub(x.mean()).div(x.std())))
```

After dropping assets and trading days like in the previous example, we are left with 23 assets and over 2,000 trading days. We compute the return covariance and estimate all of the principal components to find that the two largest explain 55.9 percent and 15.5 percent of the covariation, respectively:

```
cov = returns.cov()
pca = PCA()
pca.fit(cov)
pd.Series(pca.explained_variance_ratio_).head()
0      55.91%
1      15.52%
2       5.36%
3       4.85%
4       3.32%
```

Next, we select and normalize the four largest components so that they sum to 1, and we can use them as weights for portfolios that we can compare to an EW portfolio formed from all of the stocks:

```
top4 = pd.DataFrame(pca.components_[:4], columns=cov.columns)
eigen_portfolios = top4.div(top4.sum(1), axis=0)
eigen_portfolios.index = [f'Portfolio {i}' for i in range(1, 5)]
```

The weights show distinct emphasis, as you can see in *Figure 13.11*. For example, Portfolio 3 puts large weights on Mastercard and Visa, the two payment processors in the sample, whereas Portfolio 2 has more exposure to technology companies:

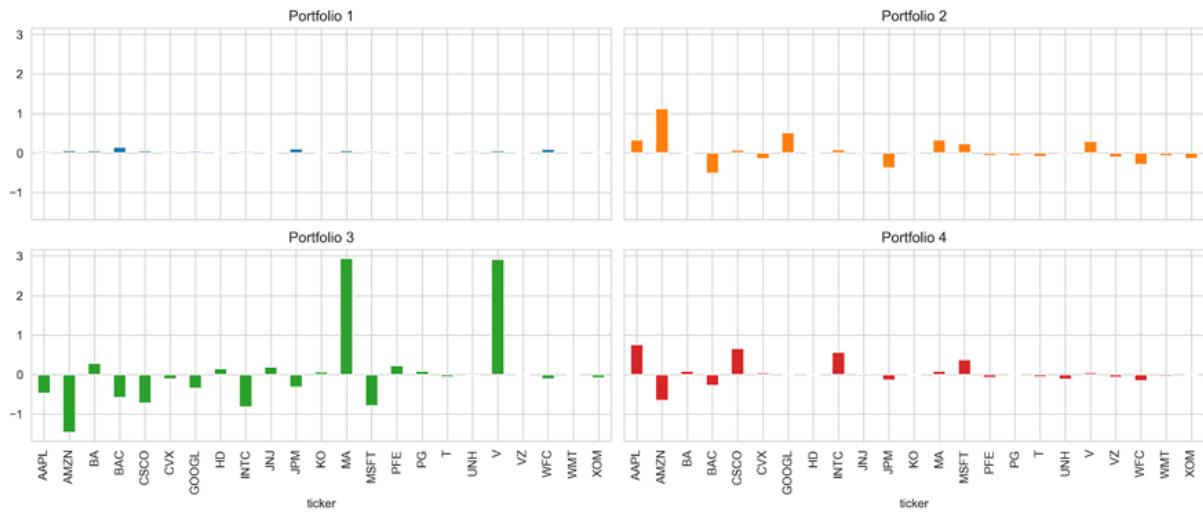


Figure 13.11: Eigenportfolio weights

When comparing the performance of each portfolio over the sample period to "the market" consisting of our small sample, we find that Portfolio 1 performs very similarly, whereas the other portfolios capture different return patterns (see *Figure 13.12*).

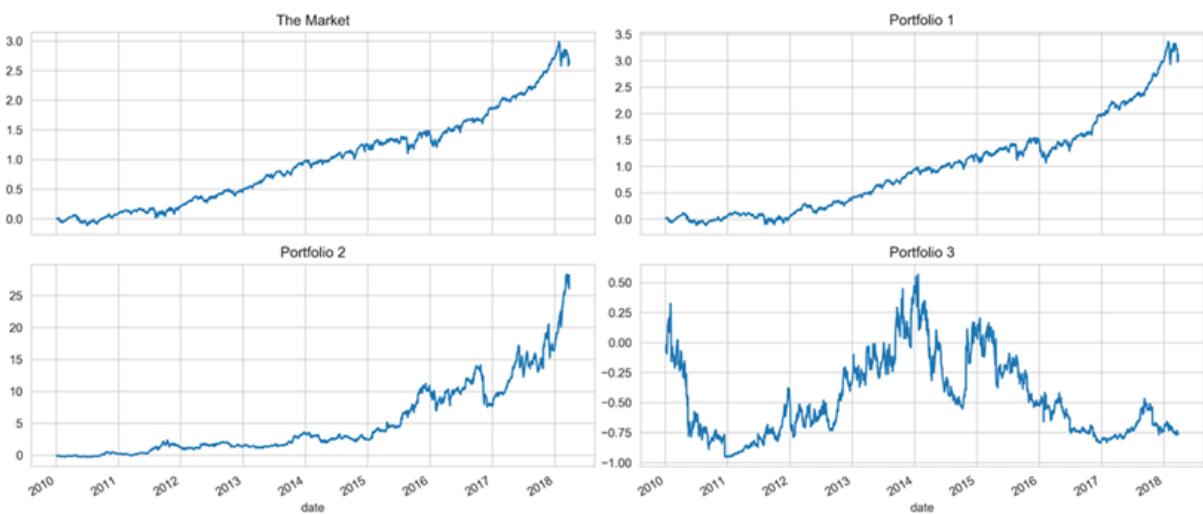


Figure 13.12: Cumulative eigenportfolio returns

Clustering

Both clustering and dimensionality reduction summarize the data. As we have just discussed, dimensionality reduction compresses the data by representing it using new, fewer features that capture the most relevant information. Clustering algorithms, in contrast, assign existing observations to subgroups that consist of similar data points.

Clustering can serve to better understand the data through the lens of categories learned from continuous variables. It also permits you to automatically categorize new objects according to the learned criteria. Examples of related applications include hierarchical taxonomies, medical diagnostics, and customer segmentation. Alternatively, clusters can be used to represent groups as prototypes, using, for example, the midpoint of a cluster as the best representatives of learned grouping. An example application includes image compression.

Clustering algorithms differ with respect to their strategy of identifying groupings:

- **Combinatorial** algorithms select the most coherent of different groupings of observations.
- **Probabilistic** modeling estimates distributions that most likely generated the clusters.
- **Hierarchical clustering** finds a sequence of nested clusters that optimizes coherence at any given stage.

Algorithms also differ by the notion of what constitutes a useful collection of objects that needs to match the data characteristics, domain, and goal of the applications. Types of groupings include:

- Clearly separated groups of various shapes
- Prototype- or center-based, compact clusters
- Density-based clusters of arbitrary shape
- Connectivity- or graph-based clusters

Important additional aspects of a clustering algorithm include whether it:

- Requires exclusive cluster membership
- Makes hard, that is, binary, or soft, probabilistic assignments
- Is complete and assigns all data points to clusters

The following sections introduce key algorithms, including **k-means**, **hierarchical**, and **density-based clustering**, as well as **Gaussian mixture models (GMMs)**. The notebook `clustering_algos` compares the performance of these algorithms on different, labeled datasets to highlight strengths and weaknesses. It uses mutual information (refer to *Chapter 6, The Machine Learning Process*) to measure the congruence of cluster assignments and labels.

k-means clustering

k-means is the most well-known clustering algorithm, and it was first proposed by Stuart Lloyd at Bell Labs in 1957. It finds k centroids and assigns each data point to exactly one cluster with the goal of minimizing the within-cluster variance (called *inertia*). It typically uses the Euclidean distance, but other metrics can also be

used. k-means assumes that clusters are spherical and of equal size and ignores the covariance among features.

Assigning observations to clusters

The problem is computationally difficult (NP-hard) because there are k^N ways to partition the N observations into k clusters. The standard iterative algorithm delivers a local optimum for a given k and proceeds as follows:

1. Randomly define k cluster centers and assign points to the nearest centroid
2. Repeat:
 1. For each cluster, compute the centroid as the average of the features
 2. Assign each observation to the closest centroid
3. Convergence: assignments (or within-cluster variation) don't change

The notebook `kmeans_implementation` shows you how to code the algorithm using Python. It visualizes the algorithm's iterative optimization and demonstrates how the resulting centroids partition the feature space into areas called Voronoi that delineate the clusters. The result is optimal for the given initialization, but alternative starting positions will produce different results. Therefore, we compute multiple clusterings from different initial values and select the solution that minimizes within-cluster variance.

k-means requires continuous or one-hot encoded categorical variables. Distance metrics are typically sensitive to scale, making it necessary to standardize features to ensure they have equal weight.

The **strengths** of k-means include its wide range of applicability, fast convergence, and linear scalability to large data while producing clusters of even size. The **weaknesses** include the need to tune the hyperparameter k , no guarantee of finding a global optimum, the restrictive assumption that clusters are spheres, and features not being correlated. It is also sensitive to outliers.

Evaluating cluster quality

Cluster quality metrics help select from among alternative clustering results. The notebook `kmeans_evaluation` illustrates the following options.

The **k-means objective** function suggests we compare the evolution of the inertia or within-cluster variance. Initially, additional centroids decrease the inertia sharply because new clusters improve the overall fit. Once an appropriate number of clusters has been found (assuming it exists), new centroids reduce the **within-cluster variance** by much less, as they tend to split natural groupings.

Therefore, when k-means finds a good cluster representation of the data, the **inertia** tends to follow an elbow-shaped path similar to the explained variance ratio for PCA (take a look at the notebook for implementation details).

The **silhouette coefficient** provides a more detailed picture of cluster quality. It answers the question: how far are the points in the nearest cluster relative to the points in the assigned cluster? To this end, it compares the mean intra-cluster distance a to the mean distance of the nearest cluster b and computes the following score s :

$$s = \frac{b - a}{\max(a, b)} \in [-1, 1]$$

The score can vary between -1 and 1, but negative values are unlikely in practice because they imply that the majority of points are assigned to the wrong cluster. A useful visualization of the silhouette score compares the values for each data point to the global average because it highlights the coherence of each cluster relative to the global configuration. The rule of thumb is to avoid clusters with mean scores below the average for all samples.

Figure 13.13 shows an excerpt from the silhouette plot for three and four clusters, where the former highlights the poor fit of cluster 1 by subpar contributions to the global silhouette score, whereas all of the four clusters have some values that exhibit above-average scores.

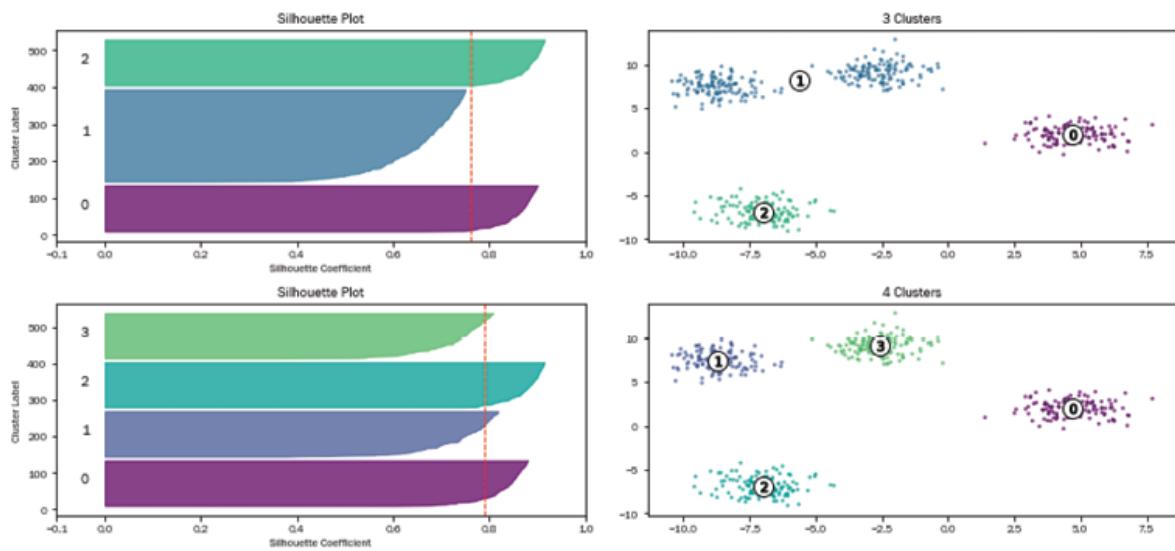


Figure 13.13: Silhouette plots for three and four clusters

In sum, given the usually unsupervised nature, it is necessary to vary the hyperparameters of the cluster algorithms and evaluate the different results. It is also important to calibrate the scale of the features, particularly when some should be given a higher weight and thus be measured on a larger scale. Finally, to validate the robustness of the results, use subsets of data to identify whether particular cluster patterns emerge consistently.

Hierarchical clustering

Hierarchical clustering avoids the need to specify a target number of clusters because it assumes that data can successively be merged into increasingly dissimilar clusters. It does not pursue a global objective but decides incrementally how to produce a sequence of nested clusters that range from a single cluster to clusters consisting of the individual data points.

Different strategies and dissimilarity measures

There are two approaches to hierarchical clustering:

1. **Agglomerative clustering** proceeds bottom-up, sequentially merging two of the remaining groups based on similarity.
2. **Divisive clustering** works top-down and sequentially splits the remaining clusters to produce the most distinct subgroups.

Both groups produce $N-1$ hierarchical levels and facilitate the selection of clustering at the level that best partitions data into homogenous groups. We will focus on the more common agglomerative clustering approach.

The agglomerative clustering algorithm departs from the individual data points and computes a similarity matrix containing all mutual distances. It then takes $N-1$ steps until there are no more distinct clusters and, each time, updates the similarity matrix to substitute elements that have been merged by the new cluster so that the matrix progressively shrinks.

While hierarchical clustering does not have hyperparameters like k-means, the **measure of dissimilarity** between clusters (as opposed to

individual data points) has an important impact on the clustering result. The options differ as follows:

- **Single-link**: Distance between the nearest neighbors of two clusters
- **Complete link**: Maximum distance between the respective cluster members
- **Ward's method**: Minimize within-cluster variance
- **Group average**: Uses the cluster midpoint as a reference distance

Visualization – dendograms

Hierarchical clustering provides insight into degrees of similarity among observations as it continues to merge data. A significant change in the similarity metric from one merge to the next suggests that a natural clustering existed prior to this point.

The **dendrogram** visualizes the successive merges as a binary tree, displaying the individual data points as leaves and the final merge as the root of the tree. It also shows how the similarity monotonically decreases from the bottom to the top. Therefore, it is natural to select a clustering by cutting the dendrogram. Refer to the notebook [hierarchical_clustering](#) for implementation details.

Figure 13.14 illustrates the dendrogram for the classic Iris dataset with four classes and three features using the four different distance metrics introduced in the preceding section. It evaluates the fit of the hierarchical clustering using the **cophenetic correlation** coefficient that compares the pairwise distances among points and the cluster similarity metric at which a pairwise merge occurred. A coefficient of 1 implies that closer points always merge earlier.

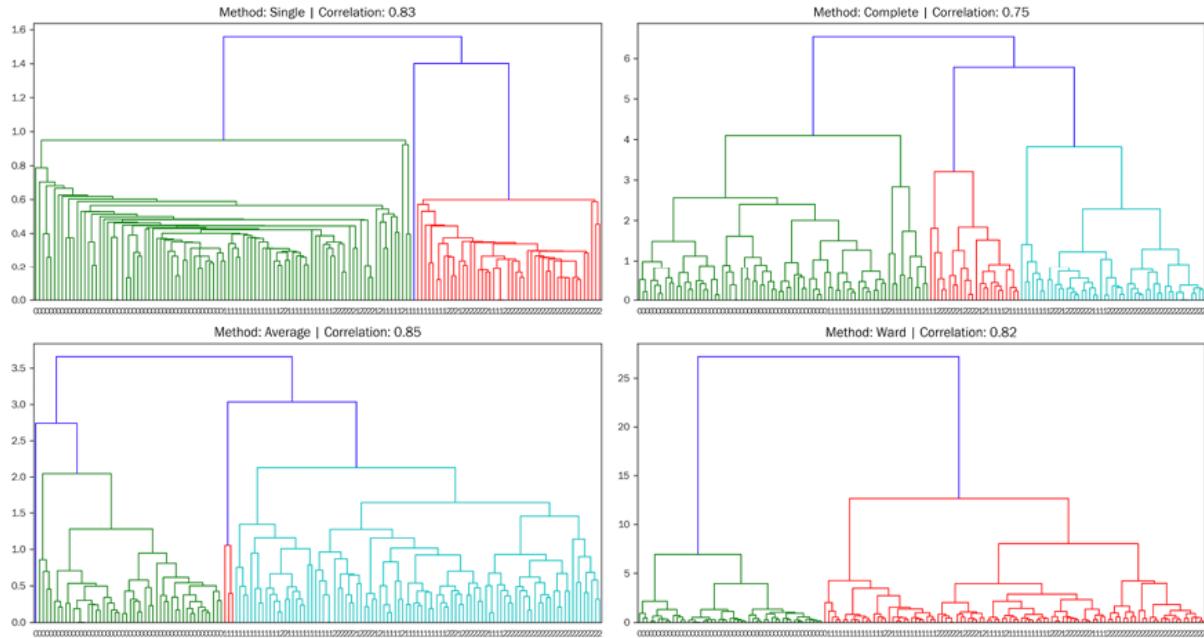


Figure 13.14: Dendrograms and cophenetic correlation for different dissimilarity measures

Different linkage methods produce different dendrogram "looks" so that we cannot use this visualization to compare results across methods. In addition, the Ward method, which minimizes the within-cluster variance, may not properly reflect the change in variance from one level to the next. Instead, the dendrogram can reflect the total within-cluster variance at different levels, which may be misleading. Alternative quality metrics are more appropriate, such as the **cophenetic correlation** or measures like **inertia** if aligned with the overall goal.

The **strengths** of hierarchical clustering include:

- The algorithm does not need the specific number of clusters but, instead, provides insight about potential clustering by means of an intuitive visualization.
- It produces a hierarchy of clusters that can serve as a taxonomy.

- It can be combined with k-means to reduce the number of items at the start of the agglomerative process.

On the other hand, its **weaknesses** include:

- The high cost in terms of computation and memory due to the numerous similarity matrix updates.
- All merges are final so that it does not achieve the global optimum.
- The curse of dimensionality leads to difficulties with noisy, high-dimensional data.

Density-based clustering

Density-based clustering algorithms assign cluster membership based on proximity to other cluster members. They pursue the goal of identifying dense regions of arbitrary shapes and sizes. They do not require the specification of a certain number of clusters but instead rely on parameters that define the size of a neighborhood and a density threshold.

We'll outline the two popular algorithms: DBSCAN and its newer hierarchical refinement. Refer to the notebook

`density_based_clustering` for the relevant code samples and the link in this chapter's `README` on GitHub to a Quantopian example by Jonathan Larking that uses DBSCAN for a pairs trading strategy.

DBSCAN

Density-based spatial clustering of applications with noise (DBSCAN) was developed in 1996 and awarded the KDD Test of

Time award at the 2014 KDD conference because of the attention it has received in theory and practice.

It aims to identify core and non-core samples, where the former extend a cluster and the latter are part of a cluster but do not have sufficient nearby neighbors to further grow the cluster. Other samples are outliers and are not assigned to any cluster.

It uses a parameter `eps` for the radius of the neighborhood and `min_samples` for the number of members required for core samples. It is deterministic and exclusive and has difficulties with clusters of different density and high-dimensional data. It can be challenging to tune the parameters to the requisite density, especially as it is often not constant.

Hierarchical DBSCAN

Hierarchical DBSCAN (HDBSCAN) is a more recent development that assumes clusters are islands of potentially differing density to overcome the DBSCAN challenges just mentioned. It also aims to identify the core and non-core samples. It uses the parameters `min_cluster_size` and `min_samples` to select a neighborhood and extend a cluster. The algorithm iterates over multiple `eps` values and chooses the most stable clustering. In addition to identifying clusters of varying density, it provides insight into the density and hierarchical structure of the data.

Figure 13.15 shows how DBSCAN and HDBSCAN, respectively, are able to identify clusters that differ in shape significantly from those discovered by k-means, for example. The selection of the clustering algorithm is a function of the structure of your data; refer to the pairs

trading strategy that was referenced earlier in this section for a practical example.

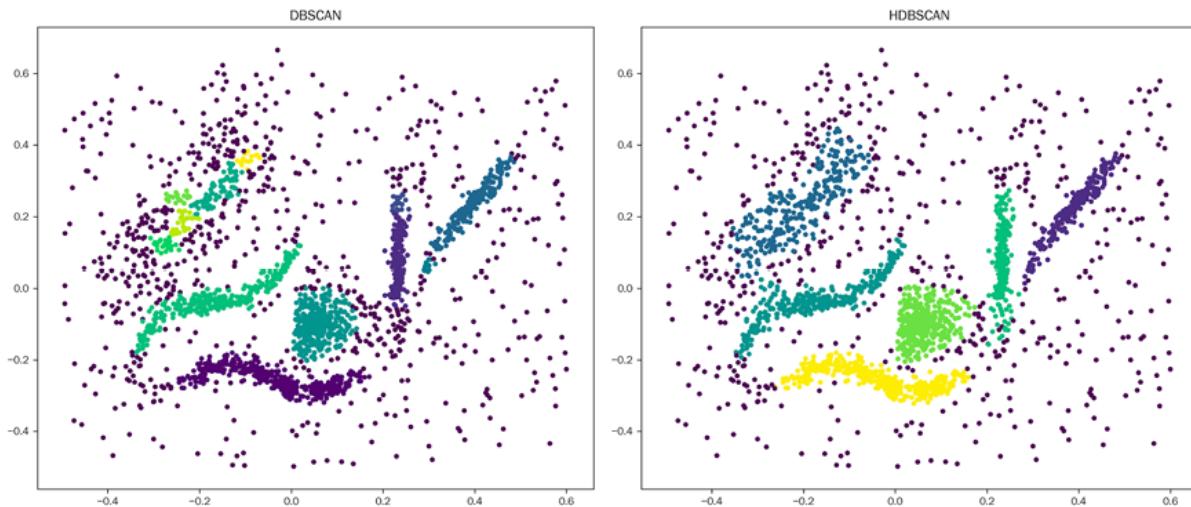


Figure 13.15: Comparing the DBSCAN and HDBSCAN clustering algorithms

Gaussian mixture models

GMMs are generative models that assume the data has been generated by a mix of various multivariate normal distributions. The algorithm aims to estimate the mean and covariance matrices of these distributions.

A GMM generalizes the k-means algorithm: it adds covariance among features so that clusters can be ellipsoids rather than spheres, while the centroids are represented by the means of each distribution. The GMM algorithm performs soft assignments because each point has a probability of being a member of any cluster.

The notebook `gaussian_mixture_models` demonstrates the implementation and visualizes the resulting cluster. You are likely to

prefer GMM over other clustering algorithms when the k-means assumption of spherical clusters is too constraining; GMM often needs fewer clusters to produce a good fit given its greater flexibility. The GMM algorithm is also preferable when you need a generative model; because GMM estimates the probability distributions that generated the samples, it is easy to generate new samples based on the result.

Hierarchical clustering for optimal portfolios

In *Chapter 5, Portfolio Optimization and Performance Evaluation*, we discussed several methods that aim to choose portfolio weights for a given set of assets to optimize the risk and return profile of the resulting portfolio. These included the mean-variance optimization of Markowitz's modern portfolio theory, the Kelly criterion, and risk parity. In this section, we cover **hierarchical risk parity (HRP)**, a more recent innovation (Prado 2016) that leverages hierarchical clustering to assign position sizes to assets based on the risk characteristics of subgroups.

We will first present how HRP works and then compare its performance against alternatives using a long-only strategy driven by the gradient boosting models we developed in the last chapter.

How hierarchical risk parity works

The key ideas of hierarchical risk parity are to do the following:

- Use hierarchical clustering of the covariance matrix to group assets with a similar correlation structure together
- Reduce the number of degrees of freedom by only considering similar assets as substitutes when constructing the portfolio

Refer to the notebook and Python files in the subfolder `hierarchical_risk_parity` for implementation details.

The first step is to compute a distance matrix that represents proximity for correlated assets and meets distance metric requirements. The resulting matrix becomes an input to the SciPy hierarchical clustering function that computes the successive clusters using one of several available methods, as discussed previously in this chapter.

```
def get_distance_matrix(corr):
    """Compute distance matrix from correlation;
    0 <= d[i,j] <= 1"""
    return np.sqrt((1 - corr) / 2)
distance_matrix = get_distance_matrix(corr)
linkage_matrix = linkage(squareform(distance_matrix), 'single')
```

The `linkage_matrix` can be used as input to the `sns.clustermap` function to visualize the resulting hierarchical clustering. The dendrogram displayed by seaborn shows how individual assets and clusters of assets merged based on their relative distances (see the left panel of *Figure 13.16*).

```
clustergrid = sns.clustermap(distance_matrix,
                             method='single',
                             row_linkage=linkage_matrix,
                             col_linkage=linkage_matrix,
                             cmap=cmap, center=0)
sorted_idx = clustergrid.dendrogram_row.reordered_ind
sorted_tickers = corr.index[sorted_idx].tolist()
```

Compared to a `seaborn.heatmap` of the original correlation matrix, there is now significantly more structure in the sorted data (the right panel) compared to the original correlation matrix displayed in the central panel.

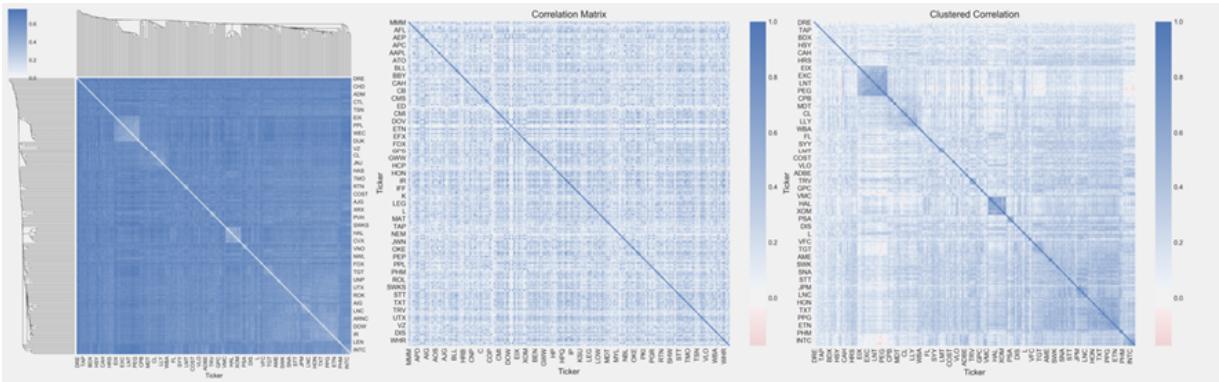


Figure 13.16: Original and clustered correlation matrix

Using the tickers sorted according to the hierarchy induced by the clustering algorithm, HRP now proceeds to compute a top-down inverse-variance allocation that successively adjusts weights depending on the variance of the subclusters further down the tree.

```
def get_inverse_var_pf(cov):
    """Compute the inverse-variance portfolio"""
    ivp = 1 / np.diag(cov)
    return ivp / ivp.sum()
def get_cluster_var(cov, cluster_items):
    """Compute variance per cluster"""
    cov_ = cov.loc[cluster_items, cluster_items] # matrix slice
    w_ = get_inverse_var_pf(cov_)
    return (w_ @ cov_ @ w_).item()
```

To this end, the algorithm uses a bisectional search to allocate the variance of a cluster to its elements based on their relative riskiness.

```
def get_hrp_allocation(cov, tickers):
    """Compute top-down HRP weights"""
```

```

weights = pd.Series(1, index=tickers)
clusters = [tickers] # initialize one cluster with all assets
while len(clusters) > 0:
    # run bisectional search:
    clusters = [c[start:stop] for c in clusters
                for start, stop in ((0, int(len(c) / 2)),
                                     (int(len(c) / 2), len(c)))
                if len(c) > 1]
    for i in range(0, len(clusters), 2): # parse in pairs
        cluster0 = clusters[i]
        cluster1 = clusters[i + 1]
        cluster0_var = get_cluster_var(cov, cluster0)
        cluster1_var = get_cluster_var(cov, cluster1)
        weight_scaler = 1 - cluster0_var / (cluster0_var + cluster1_var)
        weights[cluster0] *= weight_scaler
        weights[cluster1] *= 1 - weight_scaler
return weights

```

The resulting portfolio allocation produces weights that sum to 1 and reflect the structure present in the correlation matrix (refer to the notebook for details).

Backtesting HRP using an ML trading strategy

Now that we know how HRP works, we would like to test how it performs in practice compared to some alternatives, namely a simple equal-weighted portfolio and a mean-variance optimized portfolio. You can find the code samples for this section and additional details and analyses in the notebook

`pf_optimization_with_hrp_zipline_benchmark`.

To this end, we'll build on the gradient boosting models developed in the last chapter. We will backtest a strategy for 2015-2017 with a universe of the 1,000 most liquid US stocks. The strategy relies on the model predictions to enter long positions in the 25 stocks with the

highest positive return prediction for the next day. On a daily basis, we rebalance our holdings so that the weights for our target positions match the values suggested by HRP.

Ensembling the gradient boosting model predictions

We begin by averaging the predictions of the 10 models that performed best during the 2015-16 cross-validation period (refer to *Chapter 12, Boosting Your Trading Strategy*, for details), as shown in the following code excerpt:

```
def load_predictions(bundle):
    path = Path('..../12_gradient_boosting_machines/data')
    predictions = (pd.read_hdf(path / 'predictions.h5', 'lgb/train/01'
                               .append(pd.read_hdf(path / 'predictions.h5', 'lgb/t
    predictions = (predictions.loc[~predictions.index.duplicated()])
                   .iloc[:, :10]
                   .mean(1)
                   .sort_index()
                   .dropna()
                   .to_frame('prediction'))
```

On a daily basis, we obtain the model predictions and select the top 25 tickers. If there are at least 20 tickers with positive forecasts, we enter the long positions and close all of the other holdings:

```
def before_trading_start(context, data):
    """
    Called every day before market open.
    """
    output = pipeline_output('signals')['longs'].astype(int)
    context.longs = output[output!=0].index
    if len(context.longs) < MIN_POSITIONS:
        context.divest = set(context.portfolio.positions.keys())
    else:
        context.divest = context.portfolio.positions.keys() - context.
```

Using PyPortfolioOpt to compute HRP weights

PyPortfolioOpt, which we used in *Chapter 5, Portfolio Optimization and Performance Evaluation*, to compute mean-variance optimized weights, also implements HRP. We'll run it as part of the scheduled rebalancing that takes place every morning. It needs the return history for the target assets and returns a dictionary of ticker-weight pairs that we use to place orders:

```
def rebalance_hierarchical_risk_parity(context, data):
    """Execute orders according to schedule_function()"""
    for symbol, open_orders in get_open_orders().items():
        for open_order in open_orders:
            cancel_order(open_order)
    for asset in context.divest:
        order_target(asset, target=0)

    if len(context.longs) > context.min_positions:
        returns = (data.history(context.longs, fields='price',
                               bar_count=252+1, # for 1 year of returns
                               frequency='1d'))
        .pct_change()
        .dropna(how='all')
    hrp_weights = HRP0pt(returns=returns).hrp_portfolio()
    for asset, target in hrp_weights.items():
        order_target_percent(asset=asset, target=target)
```

Markowitz rebalancing follows a similar process, as outlined in *Chapter 5, Portfolio Optimization and Performance Evaluation*, and is included in the notebook.

Performance comparison with pyfolio

The following charts show the cumulative returns for the in- and out-of-sample (with respect to the ML model selection process) of the **equal-weighted (EW)**, the HRP, and the **mean-variance (MV)** optimized portfolios.

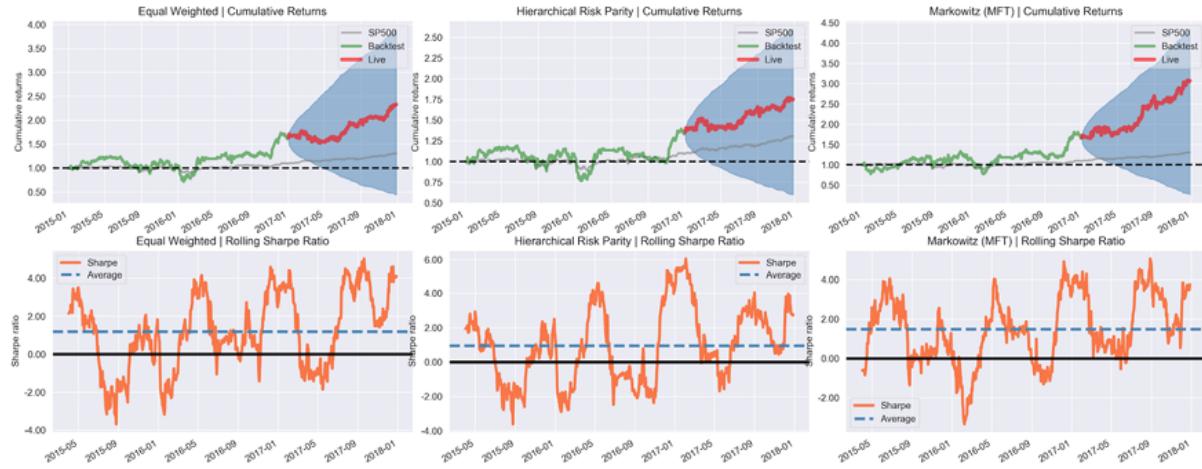


Figure 13.17: Cumulative returns for the different portfolios

The cumulative returns are 207.3 percent for MV, 133 percent for EW, and 75.1 percent for HRP. The Sharpe ratios are 1.16, 1.01, and 0.83, respectively. Alpha returns are 0.28 for MV, 0.16 for EW, and 0.16 for HRP, with betas of 1.77, 1.87, and 1.67, respectively.

Therefore, it turns out that, in this particular context, the often-criticized MV approach does best, while HRP comes up last. However, be aware that the results are quite sensitive to the number of stocks traded, the time period, and other factors.

Try it out for yourself, and learn which technique performs best under the circumstances most relevant for you!

Summary

In this chapter, we explored unsupervised learning methods that allow us to extract valuable signals from our data without relying on the help of outcome information provided by labels.

We learned how to use linear dimensionality reduction methods like PCA and ICA to extract uncorrelated or independent components from data that can serve as risk factors or portfolio weights. We also covered advanced nonlinear manifold learning techniques that produce state-of-the-art visualizations of complex, alternative datasets. In the second part of the chapter, we covered several clustering methods that produce data-driven groupings under various assumptions. These groupings can be useful, for example, to construct portfolios that apply risk-parity principles to assets that have been clustered hierarchically.

In the next three chapters, we will learn about various machine learning techniques for a key source of alternative data, namely natural language processing for text documents.

Text Data for Trading – Sentiment Analysis

This is the first of three chapters dedicated to extracting signals for algorithmic trading strategies from text data using **natural language processing (NLP)** and **machine learning (ML)**.

Text data is very rich in content but highly unstructured, so it requires more preprocessing to enable an ML algorithm to extract relevant information. A key challenge consists of converting text into a numerical format without losing its meaning. We will cover several techniques capable of capturing the nuances of language so that they can be used as input for ML algorithms.

In this chapter, we will introduce fundamental **feature extraction** techniques that focus on individual semantic units, that is, words or short groups of words called **tokens**. We will show how to represent documents as vectors of token counts by creating a document-term matrix and then proceed to use it as input for **news classification** and **sentiment analysis**. We will also introduce the naive Bayes algorithm, which is popular for this purpose.

In the following two chapters, we build on these techniques and use ML algorithms such as topic modeling and word-vector embeddings to capture the information contained in a broader context.

In particular in this chapter, we will cover the following:

- What the fundamental NLP workflow looks like
- How to build a multilingual feature extraction pipeline using spaCy and TextBlob
- Performing NLP tasks such as **part-of-speech (POS)** tagging or named entity recognition
- Converting tokens to numbers using the document-term matrix
- Classifying text using the naive Bayes model
- How to perform sentiment analysis



You can find the code samples for this chapter and links to additional resources in the corresponding directory of the GitHub repository. The notebooks include color versions of the images.

ML with text data – from language to features

Text data can be extremely valuable given how much information humans communicate and store using natural language. The diverse set of data sources relevant to financial investments range from formal documents like company statements, contracts, and patents, to news, opinion, and analyst research or commentary, to various types of social media postings or messages.

Numerous and diverse text data samples are available online to explore the use of NLP algorithms, many of which are listed among the resources included in this chapter's `README` file on GitHub. For a comprehensive introduction, see Jurafsky and Martin (2008).

To realize the potential value of text data, we'll introduce the specialized NLP techniques and the most effective Python libraries, outline key challenges particular to working with language data, introduce critical elements of the NLP workflow, and highlight NLP applications relevant for algorithmic trading.

Key challenges of working with text data

The conversion of unstructured text into a machine-readable format requires careful preprocessing to preserve the valuable semantic aspects of the data. How humans comprehend the content of language is not fully understood and improving machines' ability to understand language remains an area of very active research.

NLP is particularly challenging because the effective use of text data for ML requires an understanding of the inner workings of language as well as knowledge about the world to which it refers. Key challenges include the following:

- Ambiguity due to **polysemy**, that is, a word or phrase having different meanings depending on context ("Local High School Dropouts Cut in Half")
- The nonstandard and **evolving use** of language, especially on social media
- The use of **idioms** like "throw in the towel"
- Tricky **entity names** like "Where is A Bug's Life playing?"
- Knowledge of the world: "Mary and Sue are sisters" versus "Mary and Sue are mothers"

The NLP workflow

A key goal for using ML from text data for algorithmic trading is to extract signals from documents. A document is an individual sample from a relevant text data source, for example, a company report, a headline, a news article, or a tweet. A corpus, in turn, is a collection of documents.

Figure 14.1 lays out the **key steps** to convert documents into a dataset that can be used to train a supervised ML algorithm capable of making actionable predictions:

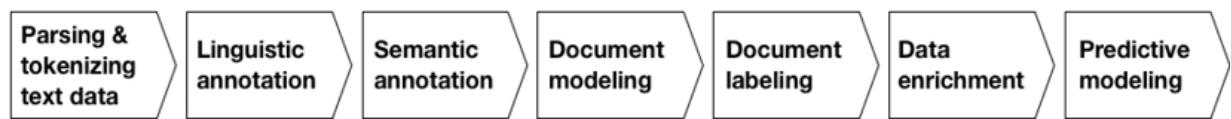


Figure 14.1: The NLP workflow

Fundamental techniques extract text features as isolated semantic units called tokens and use rules and dictionaries to annotate them with linguistic and semantic information. The bag-of-words model uses token frequency to model documents as token vectors, which leads to the document-term matrix that is frequently used for text classification, retrieval, or summarization.

Advanced approaches rely on ML to refine basic features such as tokens and produce richer document models. These include topic models that reflect the joint usage of tokens across documents and word-vector models that aim to capture the context of token usage.

We will review key decisions at each step of the workflow and the related tradeoffs in more detail before illustrating their implementation using the spaCy library in the next section. The following table summarizes the key tasks of an NLP pipeline:

Feature	Description

Tokenization	Segment text into words, punctuation marks, and so on.
Part-of-speech tagging	Assign word types to tokens, such as a verb or noun.
Dependency parsing	Label syntactic token dependencies, like subject <=> object.
Stemming and lemmatization	Assign the base forms of words: "was" => "be", "rats" => "rat".
Sentence boundary detection	Find and segment individual sentences.
Named entity recognition	Label "real-world" objects, such as people, companies, or locations.
Similarity	Evaluate the similarity of words, text spans, and documents.

Parsing and tokenizing text data – selecting the vocabulary

A token is an instance of a sequence of characters in a given document and is considered a semantic unit. The vocabulary is the set of tokens contained in a corpus deemed relevant for further processing; tokens not in the vocabulary will be ignored.

The **goal**, of course, is to extract tokens that most accurately reflect the document's meaning. The **key tradeoff** at this step is the choice of a larger vocabulary to better reflect the text source at the expense of more features and higher model complexity (discussed as the

curse of dimensionality in Chapter 13, *Data-Driven Risk Factors and Asset Allocation with Unsupervised Learning*).

Basic choices in this regard concern the treatment of punctuation and capitalization, the use of spelling correction, and whether to exclude very frequent so-called "stop words" (such as "and" or "the") as meaningless noise.

In addition, we need to decide whether to include groupings of n individual tokens called **n -grams** as semantic units (an individual token is also called *unigram*). An example of a two-gram (or *bigram*) is "New York", whereas "New York City" is a three-gram (or *trigram*). The decision can rely on dictionaries or a comparison of the relative frequencies of the individual and joint usage. There are more unique combinations of tokens than unigrams, hence adding n -grams will drive up the number of features and risks adding noise unless filtered for by frequency.

Linguistic annotation – relationships among tokens

Linguistic annotations include the application of **syntactic and grammatical rules** to identify the boundary of a sentence despite ambiguous punctuation, and a token's role and relationships in a sentence for POS tagging and dependency parsing. It also permits the identification of common root forms for stemming and lemmatization to group together related words.

The following are some key concepts related to annotations:

- **Stemming** uses simple rules to remove common endings such as *s*, *ly*, *ing*, or *ed* from a token and reduce it to its stem or root form.

- **Lemmatization** uses more sophisticated rules to derive the canonical root (**lemma**) of a word. It can detect irregular common roots such as "better" and "best" and more effectively condenses the vocabulary but is slower than stemming. Both approaches simplify the vocabulary at the expense of semantic nuances.
- **POS** annotations help disambiguate tokens based on their function (for example, when a verb and noun have the same form), which increases the vocabulary but may capture meaningful distinctions.
- **Dependency parsing** identifies hierarchical relationships among tokens and is commonly used for translation. It is important for interactive applications that require more advanced language understanding, such as chatbots.

Semantic annotation – from entities to knowledge graphs

Named-entity recognition (NER) aims at identifying tokens that represent objects of interest, such as persons, countries, or companies. It can be further developed into a **knowledge graph** that captures semantic and hierarchical relationships among such entities. It is a critical ingredient for applications that, for example, aim at predicting the impact of news events on sentiment.

Labeling – assigning outcomes for predictive modeling

Many NLP applications learn to predict outcomes based on meaningful information extracted from the text. Supervised learning requires labels to teach the algorithm the true input-output

relationship. With text data, establishing this relationship may be challenging and require explicit data modeling and collection.

Examples include decisions on how to quantify the sentiment implicit in a text document such as an email, transcribed interview, or tweet with respect to a new domain, or which aspects of a research document or news report should be assigned a specific outcome.

Applications

The use of ML with text data for trading relies on extracting meaningful information in the form of features that help predict future price movements. Applications range from the exploitation of the short-term market impact of news to the longer-term fundamental analysis of the drivers of asset valuation. Examples include the following:

- The evaluation of product review sentiment to assess a company's competitive position or industry trends
- The detection of anomalies in credit contracts to predict the probability or impact of a default
- The prediction of news impact in terms of direction, magnitude, and affected entities

JP Morgan, for instance, developed a predictive model based on 250,000 analyst reports that outperformed several benchmark indices and produced uncorrelated signals relative to sentiment factors formed from consensus EPS and recommendation changes.

From text to tokens – the NLP pipeline

In this section, we will demonstrate how to construct an NLP pipeline using the open-source Python library spaCy. The textacy library builds on spaCy and provides easy access to spaCy attributes and additional functionality.

Refer to the notebook `nlp_pipeline_with_spacy` for the following code samples, installation instruction, and additional details.

NLP pipeline with spaCy and textacy

spaCy is a widely used Python library with a comprehensive feature set for fast text processing in multiple languages. The usage of the tokenization and annotation engines requires the installation of language models. The features we will use in this chapter only require the small models; the larger models also include word vectors that we will cover in *Chapter 16, Word Embeddings for Earnings Calls and SEC Filings*.

With the library installed and linked, we can instantiate a spaCy language model and then apply it to the document. The result is a `Doc` object that tokenizes the text and processes it according to configurable pipeline components that by default consist of a tagger, a parser, and a named-entity recognizer:

```
nlp = spacy.load('en')
nlp.pipe_names
['tagger', 'parser', 'ner']
```

Let's illustrate the pipeline using a simple sentence:

```
sample_text = 'Apple is looking at buying U.K. startup for $1 billion'
doc = nlp(sample_text)
```

Parsing, tokenizing, and annotating a sentence

The parsed document content is iterable, and each element has numerous attributes produced by the processing pipeline. The next sample illustrates how to access the following attributes:

- `.text` : The original word text
- `.lemma_` : The word root
- `.pos_` : A basic POS tag
- `.tag_` : The detailed POS tag
- `.dep_` : The syntactic relationship or dependency between tokens
- `.shape_` : The shape of the word, in terms of capitalization, punctuation, and digits
- `.is_alpha` : Checks whether the token is alphanumeric
- `.is_stop` : Checks whether the token is on a list of common words for the given language

We iterate over each token and assign its attributes to a `pd.DataFrame`:

```
pd.DataFrame([[t.text, t.lemma_, t.pos_, t.tag_, t.dep_, t.shape_,
               t.is_alpha, t.is_stop]
              for t in doc],
              columns=['text', 'lemma', 'pos', 'tag', 'dep', 'shape',
                       'is_alpha', 'is_stop'])
```

This produces the following result:

text	lemma	pos	tag	dep	shape	is_alpha	is_stop
Apple	apple	PROPN	NNP	nsubj	Xxxxx	TRUE	FALSE
is	be	VERB	VBZ	aux	xx	TRUE	TRUE
looking	look	VERB	VBG	ROOT	xxxx	TRUE	FALSE
at	at	ADP	IN	prep	xx	TRUE	TRUE
buying	buy	VERB	VBG	pcomp	xxxx	TRUE	FALSE
U.K.	u.k.	PROPN	NNP	compound	X.X.	FALSE	FALSE
startup	startup	NOUN	NN	dobj	xxxx	TRUE	FALSE
for	for	ADP	IN	prep	xxx	TRUE	TRUE
\$	\$	SYM	\$	quantmod	\$	FALSE	FALSE
1	1	NUM	CD	compound	d	FALSE	FALSE
billion	billion	NUM	CD	pobj	xxxx	TRUE	FALSE

We can visualize the syntactic dependency in a browser or notebook using the following:

```
displacy.render(doc, style='dep', options=options, jupyter=True)
```

The preceding code allows us to obtain a dependency tree like the following one:

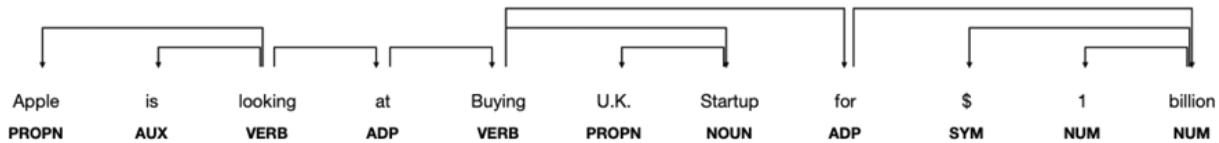


Figure 14.2: spaCy dependency tree

We can get additional insight into the meaning of attributes using `spacy.explain()`, such as the following, for example:

```
spacy.explain("VBZ")
verb, 3rd person singular present
```

Batch-processing documents

We will now read a larger set of 2,225 BBC News articles (see GitHub for the data source details) that belong to five categories and are stored in individual text files. We do the following:

1. Call the `.glob()` method of the `pathlib` module's `Path` object.
2. Iterate over the resulting list of paths.
3. Read all lines of the news article excluding the heading in the first line.
4. Append the cleaned result to a list:

```
files = Path('..', 'data', 'bbc').glob('**/*.txt')
bbc_articles = []
for i, file in enumerate(sorted(list(files))):
    with file.open(encoding='latin1') as f:
        lines = f.readlines()
        body = ' '.join([l.strip() for l in lines[1:]]).strip()
        bbc_articles.append(body)
```

```
len(bbc_articles)
2225
```

Sentence boundary detection

We will illustrate sentence detection by calling the NLP object on the first of the articles:

```
doc = nlp(bbc_articles[0])
type(doc)
spacy.tokens.doc.Doc
```

spaCy computes sentence boundaries from the syntactic parse tree so that punctuation and capitalization play an important but not decisive role. As a result, boundaries will coincide with clause boundaries, even for poorly punctuated text.

We can access the parsed sentences using the `.sents` attribute:

```
sentences = [s for s in doc.sents]
sentences[:3]
[Quarterly profits at US media giant TimeWarner jumped 76% to $1.13bn
 The firm, which is now one of the biggest investors in Google, benefi
TimeWarner said fourth quarter sales rose 2% to $11.1bn from $10.9bn.
```

Named entity recognition

spaCy enables named entity recognition using the `.ent_type_` attribute:

```
for t in sentences[0]:
    if t.ent_type_:
        print('{} | {} | {}'.format(t.text, t.ent_type_, spacy.explain
Quarterly | DATE | Absolute or relative dates or periods
```

```
US | GPE | Countries, cities, states
TimeWarner | ORG | Companies, agencies, institutions, etc.
```

Textacy makes access to the named entities that appear in the first article easy:

```
entities = [e.text for e in entities(doc)]
pd.Series(entities).value_counts().head()
TimeWarner      7
AOL             5
fourth quarter  3
year-earlier    2
one             2
```

N-grams

N-grams combine n consecutive tokens. This can be useful for the bag-of-words model because, depending on the textual context, treating (for example) "data scientist" as a single token may be more meaningful than the two distinct tokens "data" and "scientist".

Textacy makes it easy to view the `ngrams` of a given length `n` occurring at least `min_freq` times:

```
pd.Series([n.text for n in ngrams(doc, n=2, min_freq=2)]).value_counts()
fourth quarter      3
quarter profits    2
Time Warner        2
company said      2
AOL Europe        2
```

spaCy's streaming API

To pass a larger number of documents through the processing pipeline, we can use spaCy's streaming API as follows:

```
iter_texts = (bbc_articles[i] for i in range(len(bbc_articles)))
for i, doc in enumerate(nlp.pipe(iter_texts, batch_size=50, n_threads=4)):
    assert doc.is_parsed
```

Multi-language NLP

spaCy includes trained language models for English, German, Spanish, Portuguese, French, Italian, and Dutch, as well as a multi-language model for named-entity recognition. Cross-language usage is straightforward since the API does not change.

We will illustrate the Spanish language model using a parallel corpus of TED talk subtitles (see the GitHub repo for data source references). For this purpose, we instantiate both language models:

```
model = {}
for language in ['en', 'es']:
    model[language] = spacy.load(language)
```

We read small corresponding text samples in each model:

```
text = {}
path = Path('../data/TED')
for language in ['en', 'es']:
    file_name = path / 'TED2013_sample.{}.format(language)
    text[language] = file_name.read_text()
```

Sentence boundary detection uses the same logic but finds a different breakdown:

```
parsed, sentences = {}, {}
for language in ['en', 'es']:
    parsed[language] = model[language](text[language])
    sentences[language] = list(parsed[language].sents)
    print('Sentences:', language, len(sentences[language]))
```

```
Sentences: en 22
Sentences: es 22
```

POS tagging also works in the same way:

```
pos = {}
for language in ['en', 'es']:
    pos[language] = pd.DataFrame([[t.text, t.pos_, spacy.explain(t.pos_)]
                                  for t in sentences[language][0]],
                                  columns=['Token', 'POS Tag', 'Meaning'])
pd.concat([pos['en'], pos['es']], axis=1).head()
```

This produces the following table:

Token	POS Tag	Meaning	Token	POS Tag	Meaning
There	ADV	adverb	Existe	VERB	verb
s	VERB	verb	una	DET	determiner
a	DET	determiner	estrecha	ADJ	adjective
tight	ADJ	adjective	y	CONJ	conjunction
and	CCONJ	coordinating conjunction	sorprendente	ADJ	adjective

The next section illustrates how to use the parsed and annotated tokens to build a document-term matrix that can be used for text classification.

NLP with TextBlob

TextBlob is a Python library that provides a simple API for common NLP tasks and builds on the **Natural Language Toolkit (NLTK)** and the Pattern web mining libraries. TextBlob facilitates POS tagging, noun phrase extraction, sentiment analysis, classification, and translation, among others.

To illustrate the use of TextBlob, we sample a BBC Sport article with the headline "Robinson ready for difficult task". Similar to spaCy and other libraries, the first step is to pass the document through a pipeline represented by the `TextBlob` object to assign the annotations required for various tasks (see the notebook `nlp_with_textblob`):

```
from textblob import TextBlob
article = docs.sample(1).squeeze()
parsed_body = TextBlob(article.body)
```

Stemming

To perform stemming, we instantiate the `SnowballStemmer` from the NLTK library, call its `.stem()` method on each token, and display tokens that were modified as a result:

```
from nltk.stem.snowball import SnowballStemmer
stemmer = SnowballStemmer('english')
[(word, stemmer.stem(word)) for i, word in enumerate(parsed_body.words
  if word.lower() != stemmer.stem(parsed_body.words[i])]
[('Manchester', 'manchest'),
 ('United', 'unit'),
 ('reduced', 'reduc'),
 ('points', 'point'),
 ('scrappy', 'scrappi')]
```

Sentiment polarity and subjectivity

TextBlob provides polarity and subjectivity estimates for parsed documents using dictionaries provided by the Pattern library. These dictionaries lexicon-map adjectives frequently found in product reviews to sentiment polarity scores, ranging from -1 to +1 (negative ↔ positive) and a similar subjectivity score (objective ↔ subjective).

The `.sentiment` attribute provides the average for each score over the relevant tokens, whereas the `.sentiment_assessments` attribute lists the underlying values for each token (see the notebook):

```
parsed_body.sentiment
Sentiment(polarity=0.088031914893617, subjectivity=0.46456433637284694
```

Counting tokens – the document-term matrix

In this section, we first introduce how the bag-of-words model converts text data into a numeric vector space representations. The goal is to approximate document similarity by their distance in that space. We then proceed to illustrate how to create a document-term matrix using the sklearn library.

The bag-of-words model

The bag-of-words model represents a document based on the frequency of the terms or tokens it contains. Each document becomes a vector with one entry for each token in the vocabulary that reflects the token's relevance to the document.

Creating the document-term matrix

The document-term matrix is straightforward to compute given the vocabulary. However, it is also a crude simplification because it abstracts from word order and grammatical relationships. Nonetheless, it often achieves good results in text classification quickly and, thus, provides a very useful starting point.

The left panel of *Figure 14.3* illustrates how this document model converts text data into a matrix with numerical entries where each row corresponds to a document and each column to a token in the vocabulary. The resulting matrix is usually both very high-dimensional and sparse, that is, it contains many zero entries because most documents only contain a small fraction of the overall vocabulary.

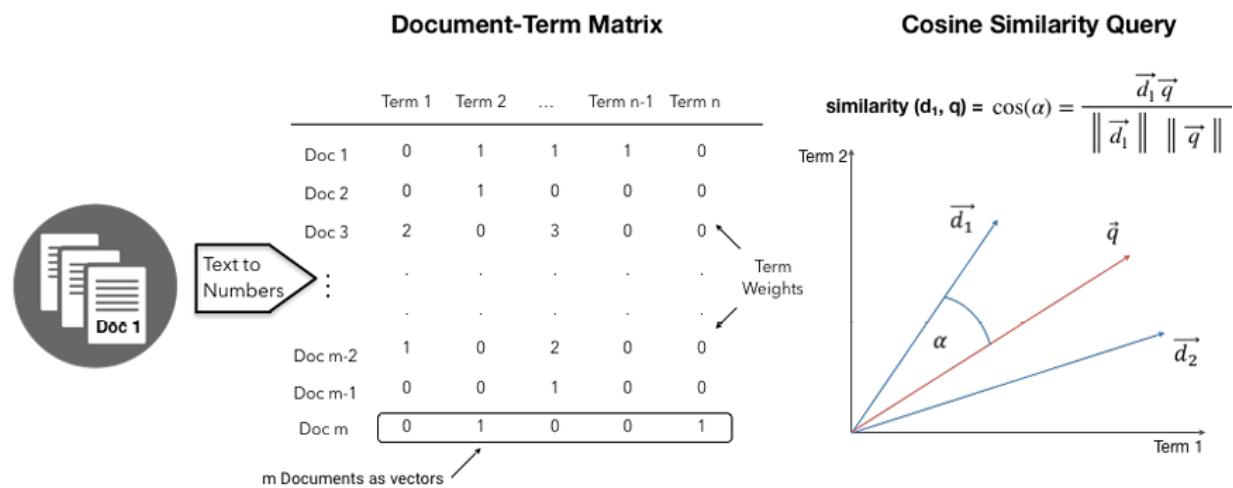


Figure 14.3: Document-term matrix and cosine similarity

There are several ways to weigh a token's vector entry to capture its relevance to the document. We will illustrate how to use `sklearn` to use binary flags that indicate presence or absence, counts, and weighted counts that account for differences in term frequencies across all documents in the corpus.

Measuring the similarity of documents

The representation of documents as word vectors assigns to each document a location in the vector space created by the vocabulary. Interpreting the vector entries as Cartesian coordinates in this space, we can use the angle between two vectors to measure their similarity because vectors that point in the same direction contain the same terms with the same frequency weights.

The right panel of the preceding figure illustrates—simplified in two dimensions—the calculation of the distance between a document represented by a vector d_1 and a query vector (either a set of search terms or another document) represented by the vector q .

The **cosine similarity** equals the cosine of the angle between the two vectors. It translates the size of the angle into a number in the range $[0, 1]$ since all vector entries are non-negative token weights. A value of 1 implies that both documents are identical with respect to their token weights, whereas a value of 0 implies that the two documents only contain distinct tokens.

As shown in the figure, the cosine of the angle is equal to the dot product of the vectors, that is, the sum product of their coordinates, divided by the product of the lengths, measured by the Euclidean norms, of each vector.

Document-term matrix with scikit-learn

The scikit-learn preprocessing module offers two tools to create a document-term matrix. `CountVectorizer` uses binary or absolute counts to measure the **term frequency (TF)** $tf(d, t)$ for each document d and token t .

`TfidfVectorizer`, by contrast, weighs the (absolute) term frequency by the **inverse document frequency (IDF)**. As a result, a term that appears in more documents will receive a lower weight than a token with the same frequency for a given document but with a lower frequency across all documents. More specifically, using the default settings, the $tf\text{-}idf(d, t)$ entries for the document-term matrix are computed as $tf\text{-}idf(d, t) = tf(d, t) \times idf(t)$ with:

$$idf(t) = \log \frac{1 + n_d}{1 + df(d, t)} + 1$$

where n_d is the number of documents and $df(d, t)$ the document frequency of term t . The resulting TF-IDF vectors for each document are normalized with respect to their absolute or squared totals (see the `sklearn` documentation for details). The TF-IDF measure was originally used in information retrieval to rank search engine results and has subsequently proven useful for text classification and clustering.

Both tools use the same interface and perform tokenization and further optional preprocessing of a list of documents before vectorizing the text by generating token counts to populate the document-term matrix.

Key parameters that affect the size of the vocabulary include the following:

- `stop_words` : Uses a built-in or user-provided list of (frequent) words to exclude
- `ngram_range` : Includes n -grams in a range of n defined by a tuple of (n_{\min}, n_{\max})
- `lowercase` : Converts characters accordingly (the default is `True`)

- `min_df / max_df` : Ignores words that appear in less/more (`int`) or are present in a smaller/larger share of documents (if `float` [0.0,1.0])
- `max_features` : Limits the number of tokens in vocabulary accordingly
- `binary` : Sets non-zero counts to 1 (`True`)

See the notebook `document_term_matrix` for the following code samples and additional details. We are again using the 2,225 BBC news articles for illustration.

Using CountVectorizer

The notebook contains an interactive visualization that explores the impact of the `min_df` and `max_df` settings on the size of the vocabulary. We read the articles into a DataFrame, set `CountVectorizer` to produce binary flags and use all tokens, and call its `.fit_transform()` method to produce a document-term matrix:

```
binary_vectorizer = CountVectorizer(max_df=1.0,
                                    min_df=1,
                                    binary=True)
binary_dtm = binary_vectorizer.fit_transform(docs.body)
<2225x29275 sparse matrix of type '<class 'numpy.int64'>'  

  with 445870 stored elements in Compressed Sparse Row format>
```

The output is a `scipy.sparse` matrix in row format that efficiently stores a small share (<0.7 percent) of the 445,870 non-zero entries in the 2,225 (document) rows and 29,275 (token) columns.

Visualizing the vocabulary distribution

The visualization in *Figure 14.4* shows that requiring tokens to appear in at least 1 percent and less than 50 percent of documents

restricts the vocabulary to around 10 percent of the almost 30,000 tokens.

This leaves a mode of slightly over 100 unique tokens per document, as shown in the left panel of the following plot. The right panel shows the document frequency histogram for the remaining tokens:

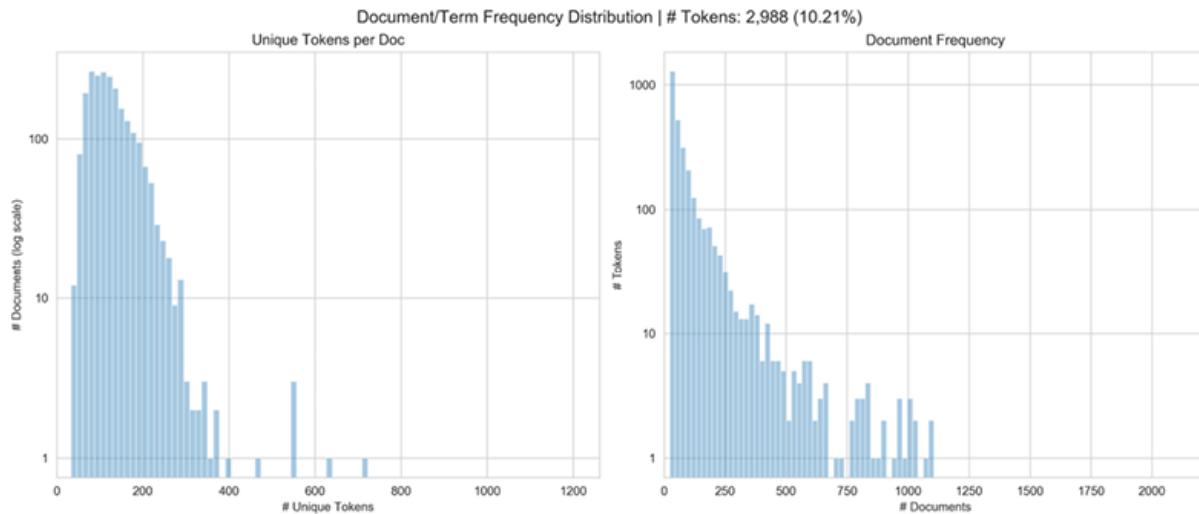


Figure 14.4: The distributions of unique tokens and number of tokens per document

Finding the most similar documents

`CountVectorizer` result lets us find the most similar documents using the `pdist()` functions for pairwise distances provided by the `scipy.spatial.distance` module. It returns a condensed distance matrix with entries corresponding to the upper triangle of a square matrix. We use `np.triu_indices()` to translate the index that minimizes the distance to the row and column indices that in turn correspond to the closest token vectors:

```
m = binary_dtm.todense()          # pdist does not accept sparse format
pairwise_distances = pdist(m, metric='cosine')
closest = np.argmin(pairwise_distances) # index that minimizes dist
rows, cols = np.triu_indices(n_docs)    # get row-col indices
```

```
rows[closest], cols[closest]
(6, 245)
```

Articles 6 and 245 are closest by cosine similarity, due to the fact that they share 38 tokens out of a combined vocabulary of 303 (see notebook). The following table summarizes these two articles and demonstrates the limited ability of similarity measures based on word counts to identify deeper semantic similarity:

	Article 6	Article 245
Topic	Business	Business
Heading	Jobs growth still slow in the US	Ebbers 'aware' of WorldCom fraud
Body	The US created fewer jobs than expected in January, but a fall in jobseekers pushed the unemployment rate to its lowest level in three years. According to Labor Department figures, US firms added only 146,000 jobs in January.	Former WorldCom boss Bernie Ebbers was directly involved in the \$11bn financial fraud at the firm, his closest associate has told a US court. Giving evidence in the criminal trial of Mr Ebbers, ex-finance chief Scott Sullivan implicated his colleague in the accounting scandal at the firm.

Both `CountVectorizer` and `TfidfVectorizer` can be used with spaCy, for example, to perform lemmatization and exclude certain characters during tokenization:

```
nlp = spacy.load('en')
def tokenizer(doc):
    return [w.lemma_ for w in nlp(doc)
           if not w.is_punct | w.is_space]
vectorizer = CountVectorizer(tokenizer=tokenizer, binary=True)
doc_term_matrix = vectorizer.fit_transform(docs.body)
```

See the notebook for additional detail and more examples.

TfidfTransformer and TfidfVectorizer

`TfidfTransformer` computes the TF-IDF weights from a document-term matrix of token counts like the one produced by `CountVectorizer`.

`TfidfVectorizer` performs both computations in a single step. It adds a few parameters to the `CountVectorizer` API that controls the smoothing behavior.

The TFIDF computation works as follows for a small text sample:

```
sample_docs = ['call you tomorrow',
               'Call me a taxi',
               'please call me... PLEASE!']
```

We compute the term frequency as before:

```
vectorizer = CountVectorizer()
tf_dtm = vectorizer.fit_transform(sample_docs).todense()
tokens = vectorizer.get_feature_names()
term_frequency = pd.DataFrame(data=tf_dtm,
                                columns=tokens)
call  me  please  taxi  tomorrow  you
0      1    0        0    0        1    1
1      1    1        0    1        0    0
2      1    1        2    0        0    0
```

The document frequency is the number of documents containing the token:

```
vectorizer = CountVectorizer(binary=True)
df_dtm = vectorizer.fit_transform(sample_docs).todense().sum(axis=0)
document_frequency = pd.DataFrame(data=df_dtm,
                                    columns=tokens)
```

```
call  me  please  taxi  tomorrow  you
0      3    2        1      1        1    1
```

The TF-IDF weights are the ratio of these values:

```
tfidf = pd.DataFrame(data=tf_dtm/df_dtm, columns=tokens)
call  me  please  taxi  tomorrow  you
0  0.33 0.00    0.00  0.00      1.00 1.00
1  0.33 0.50    0.00  1.00      0.00 0.00
2  0.33 0.50    2.00  0.00      0.00 0.00
```

The effect of smoothing

To avoid zero division, `TfidfVectorizer` uses smoothing for document and term frequencies:

- `smooth_idf`: Adds one to document frequency, as if an extra document contained every token in the vocabulary, to prevent zero divisions
- `sublinear_tf`: Applies sublinear tf scaling, that is, replaces tf with $1 + \log(tf)$

In combination with normed weights, the results differ slightly:

```
vect = TfidfVectorizer(smooth_idf=True,
                      norm='l2',  # squared weights sum to 1 by document
                      sublinear_tf=False, # if True, use 1+log(tf)
                      binary=False)
pd.DataFrame(vect.fit_transform(sample_docs).todense(),
              columns=vect.get_feature_names())
call  me  please  taxi  tomorrow  you
0  0.39 0.00    0.00  0.00      0.65 0.65
1  0.43 0.55    0.00  0.72      0.00 0.00
2  0.27 0.34    0.90  0.00      0.00 0.00
```

Summarizing news articles using `TfidfVectorizer`

Due to their ability to assign meaningful token weights, TF-IDF vectors are also used to summarize text data. For example, Reddit's `autotldr` function is based on a similar algorithm. See the notebook for an example using the BBC articles.

Key lessons instead of lessons learned

The large number of techniques and options to process natural language for use in ML models corresponds to the complex nature of this highly unstructured data source. The engineering of good language features is both challenging and rewarding, and arguably the most important step in unlocking the semantic value hidden in text data.

In practice, experience helps to select transformations that remove the noise rather than the signal, but it will likely remain necessary to cross-validate and compare the performance of different combinations of preprocessing choices.

NLP for trading

Once text data has been converted into numerical features using the NLP techniques discussed in the previous sections, text classification works just like any other classification task.

In this section, we will apply these preprocessing techniques to news articles, product reviews, and Twitter data and teach various classifiers to predict discrete news categories, review scores, and sentiment polarity.

First, we will introduce the naive Bayes model, a probabilistic classification algorithm that works well with the text features produced by a bag-of-words model.

The code samples for this section are in the notebook [news_text_classification](#).

The naive Bayes classifier

The naive Bayes algorithm is very popular for text classification because its low computational cost and memory requirements facilitate training on very large, high-dimensional datasets. Its predictive performance can compete with more complex models, provides a good baseline, and is best known for successful spam detection.

The model relies on Bayes' theorem and the assumption that the various features are independent of each other given the outcome class. In other words, for a given outcome, knowing the value of one feature (for example, the presence of a token in a document) does not provide any information about the value of another feature.

Bayes' theorem refresher

Bayes' theorem expresses the conditional probability of one event (for example, that an email is spam as opposed to benign "ham") given another event (for example, that the email contains certain words) as follows:

$$\underbrace{P(\text{is spam} \mid \text{has words})}_{\text{Posterior}} = \frac{\underbrace{P(\text{has words} \mid \text{is spam})}_{\text{Likelihood}} \underbrace{P(\text{is spam})}_{\text{Prior}}}{\underbrace{P(\text{has words})}_{\text{Evidence}}} \%$$

The **posterior** probability that an email is in fact spam given that it contains certain words depends on the interplay of three factors:

- The **prior** probability that an email is spam
- The **likelihood** of encountering these words in a spam email
- The **evidence**, that is, the probability of seeing these words in an email

To compute the posterior, we can ignore the evidence because it is the same for all outcomes (spam versus ham), and the unconditional prior may be easy to compute.

However, the likelihood poses insurmountable challenges for a reasonably sized vocabulary and a real-world corpus of emails. The reason is the combinatorial explosion of the words that did or did not appear jointly in different documents that prevent the evaluation required to compute a probability table and assign a value to the likelihood.

The conditional independence assumption

The key assumption to make the model both tractable and earn it the name *naive* is that the features are independent conditional on the outcome. To illustrate, let's classify an email with the three words "Send money now" so that Bayes' theorem becomes the following:

$$P(\text{spam} \mid \text{send money now}) = \frac{P(\text{send money now} \mid \text{spam}) \times P(\text{spam})}{P(\text{send money now})}$$

Formally, the assumption that the three words are conditionally independent means that the probability of observing "send" is not affected by the presence of the other terms given that the mail is spam, that is, $P(\text{send} \mid \text{money, now, spam}) = P(\text{send} \mid \text{spam})$. As a result, we can simplify the likelihood function:

$$P(\text{spam} \mid \text{send money now}) = \frac{P(\text{send} \mid \text{spam}) \times P(\text{money} \mid \text{spam}) \times P(\text{now} \mid \text{spam}) \times P(\text{spam})}{P(\text{send money now})}$$

Using the "naive" conditional independence assumption, each term in the numerator is straightforward to compute as relative frequencies from the training data. The denominator is constant across classes and can be ignored when posterior probabilities need to be compared rather than calibrated. The prior probability becomes less relevant as the number of factors, that is, features, increases.

In sum, the advantages of the naive Bayes model are fast training and prediction because the number of parameters is proportional to the number of features, and their estimation has a closed-form solution (based on training data frequencies) rather than expensive iterative optimization. It is also intuitive and somewhat interpretable, does not require hyperparameter tuning and is relatively robust to irrelevant features given sufficient signal.

However, when the independence assumption does not hold and text classification depends on combinations of features, or features are correlated, the model will perform poorly.

Classifying news articles

We start with an illustration of the naive Bayes model for news article classification using the BBC articles that we read before to

obtain a `DataFrame` with 2,225 articles from five categories:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2225 entries, 0 to 2224
Data columns (total 3 columns):
topic      2225 non-null object
heading    2225 non-null object
body       2225 non-null object
```

To train and evaluate a multinomial naive Bayes classifier, we split the data into the default 75:25 train-test set ratio, ensuring that the test set classes closely mirror the train set:

```
y = pd.factorize(docs.topic)[0] # create integer class values
X = docs.body
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42,
                                                    stratify=y)
```

We proceed to learn the vocabulary from the training set and transform both datasets using `CountVectorizer` with default settings to obtain almost 26,000 features:

```
vectorizer = CountVectorizer()
X_train_dtm = vectorizer.fit_transform(X_train)
X_test_dtm = vectorizer.transform(X_test)
X_train_dtm.shape, X_test_dtm.shape
((1668, 25919), (557, 25919))
```

Training and prediction follow the standard sklearn fit/predict interface:

```
nb = MultinomialNB()
nb.fit(X_train_dtm, y_train)
y_pred_class = nb.predict(X_test_dtm)
```

We evaluate the multiclass predictions using `accuracy` to find the default classifier achieved an accuracy of almost 98 percent:

```
accuracy_score(y_test, y_pred_class)
0.97666068222621
```

Sentiment analysis with Twitter and Yelp data

Sentiment analysis is one of the most popular uses of NLP and ML for trading because positive or negative perspectives on assets or other price drivers are likely to impact returns.

Generally, modeling approaches to sentiment analysis rely on dictionaries (as does the TextBlob library) or models trained on outcomes for a specific domain. The latter is often preferable because it permits more targeted labeling, for example, by tying text features to subsequent price changes rather than indirect sentiment scores.

We will illustrate ML for sentiment analysis using a Twitter dataset with binary polarity labels and a large Yelp business review dataset with a five-point outcome scale.

Binary sentiment classification with Twitter data

We use a dataset that contains 1.6 million training and 350 test tweets from 2009 with algorithmically assigned binary positive and negative sentiment scores that are fairly evenly split (see the notebook for more detailed data exploration).

Multinomial naive Bayes

We create a document-term matrix with 934 tokens as follows:

```
vectorizer = CountVectorizer(min_df=.001, max_df=.8, stop_words='english')
train_dtm = vectorizer.fit_transform(train.text)
<1566668x934 sparse matrix of type '<class 'numpy.int64'>'  
with 6332930 stored elements in Compressed Sparse Row format>
```

We then train the `MultinomialNB` classifier as before and predict the test set:

```
nb = MultinomialNB()
nb.fit(train_dtm, train.polarity)
predicted_polarity = nb.predict(test_dtm)
```

The result has over **77.5** percent accuracy:

```
accuracy_score(test.polarity, predicted_polarity)
0.7768361581920904
```

Comparison with TextBlob sentiment scores

We also obtain TextBlob sentiment scores for the tweets and note (see the left panel in *Figure 14.5*) that positive test tweets receive a significantly higher sentiment estimate. We then use the

`MultinomialNB` model's `.predict_proba()` method to compute predicted probabilities and compare both models using the respective area **under the curve**, or **AUC**, that we introduced in *Chapter 6, The Machine Learning Process* (see the right panel in *Figure 14.5*).

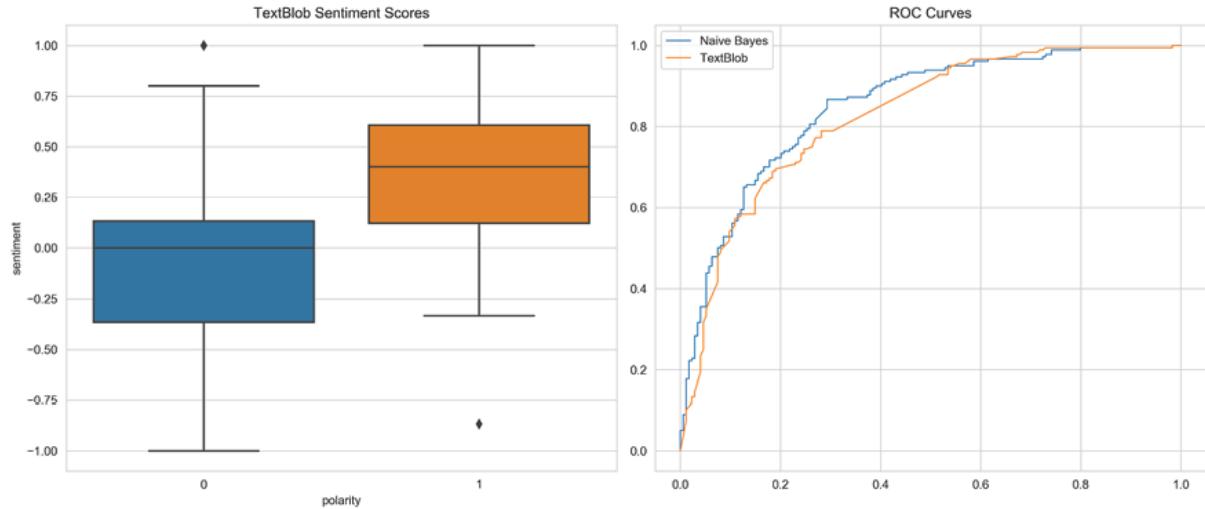


Figure 14.5: Accuracy of custom versus generic sentiment scores

The custom naive Bayes model outperforms TextBlob in this case, achieving a test AUC of 0.848 compared to 0.825 for TextBlob.

Multiclass sentiment analysis with Yelp business reviews

Finally, we apply sentiment analysis to the significantly larger Yelp business review dataset with five outcome classes (see the notebook `sentiment_analysis_yelp` for code and additional details).

The data consists of several files with information on the business, the user, the review, and other aspects that Yelp provides to encourage data science innovation.

We will use around six million reviews produced over the 2010-2018 period (see the notebook for details). The following figure shows the number of reviews and the average number of stars per year, as well as the star distribution across all reviews.

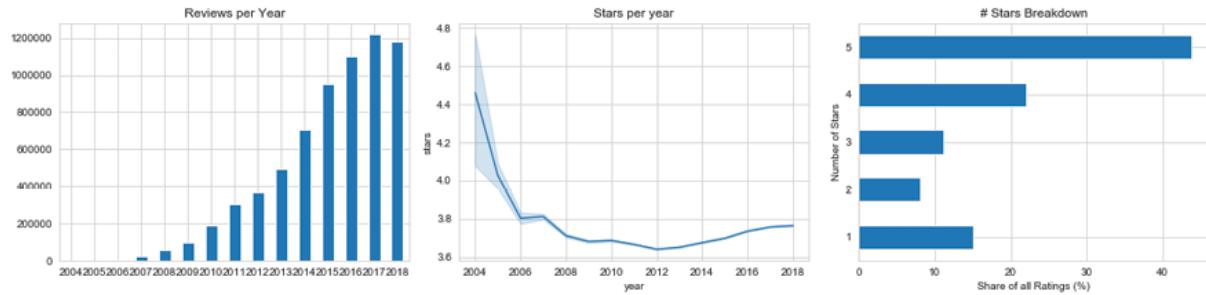


Figure 14.6: Basic exploratory analysis of Yelp reviews

We will train various models on a 10 percent sample of the data through 2017 and use the 2018 reviews as the test set. In addition to the text features resulting from the review texts, we will also use other information submitted with the review about the given user.

Combining text and numerical features

The dataset contains various numerical features (see notebook for implementation details).

The vectorizers produce `scipy.sparse` matrices. To combine the vectorized text data with other features, we need to first convert these to sparse matrices as well; many sklearn objects and other libraries such as LightGBM can handle these very memory-efficient data structures. Converting the sparse matrix to a dense NumPy array risks memory overflow.

Most variables are categorical, so we use one-hot encoding since we have a fairly large dataset to accommodate the increase in features.

We convert the encoded numerical features and combine them with the document-term matrix:

```
train_numeric = sparse.csr_matrix(train_dummies.astype(np.uint))
train_dtm_numeric = sparse.hstack((train_dtm, train_numeric))
```

Benchmark accuracy

Using the most frequent number of stars (=5) to predict the test set achieves an accuracy close to 52 percent:

```
test['predicted'] = train.stars.mode().iloc[0]
accuracy_score(test.stars, test.predicted)
0.5196950594793454
```

Multinomial naive Bayes model

Next, we train a naive Bayes classifier using a document-term matrix produced by `CountVectorizer` with default settings.

```
nb = MultinomialNB()
nb.fit(train_dtm,train.stars)
predicted_stars = nb.predict(test_dtm)
```

The prediction produces 64.7 percent accuracy on the test set, a 24.4 percent improvement over the benchmark:

```
accuracy_score(test.stars, predicted_stars)
0.6465164206691094
```

Training with the combination of text and other features improves the test accuracy to 0.671.

Logistic regression

In *Chapter 7, Linear Models – From Risk Factors to Return Forecasts*, we introduced binary logistic regression. sklearn also implements a multiclass model with a multinomial and a one-versus-all training option, where the latter trains a binary model for each class while considering all other classes as the negative class. The multinomial

option is much faster and more accurate than the one-versus-all implementation.

We evaluate a range of values for the regularization parameter `c` to identify the best performing model, using the `lbgfs` solver as follows (see the `sklearn` documentation for details):

```
def evaluate_model(model, X_train, X_test, name, store=False):
    start = time()
    model.fit(X_train, train.stars)
    runtime[name] = time() - start
    predictions[name] = model.predict(X_test)
    accuracy[result] = accuracy_score(test.stars, predictions[result])
    if store:
        joblib.dump(model, f'results/{result}.joblib')
Cs = np.logspace(-5, 5, 11)
for C in Cs:
    model = LogisticRegression(C=C, multi_class='multinomial', solver=
evaluate_model(model, train_dtm, test_dtm, result, store=True)
```

Figure 14.7 shows the plots of the validation results.

Multiclass gradient boosting with LightGBM

For comparison, we also train a LightGBM gradient boosting tree ensemble with default settings and a `multiclass` objective:

```
param = {'objective':'multiclass', 'num_class': 5}
booster = lgb.train(params=param,
                     train_set=lgb_train,
                     num_boost_round=500,
                     early_stopping_rounds=20,
                     valid_sets=[lgb_train, lgb_test])
```

Predictive performance

Figure 14.7 displays the accuracy of each model for the combined data. The right panel plots the validation performance for the logistic regression models for both datasets and different levels of regularization.

Multinomial logistic regression performs best with a test accuracy slightly above 74 percent. Naive Bayes performs significantly worse. The default LightGBM settings did not improve over the linear model with an accuracy of 0.736. However, we could tune the hyperparameters of the gradient boosting model and may well see performance improvements that put it at least on par with logistic regression. Either way, the result serves as a reminder not to discount simple, regularized models as they may deliver not only good results, but also do so quickly.

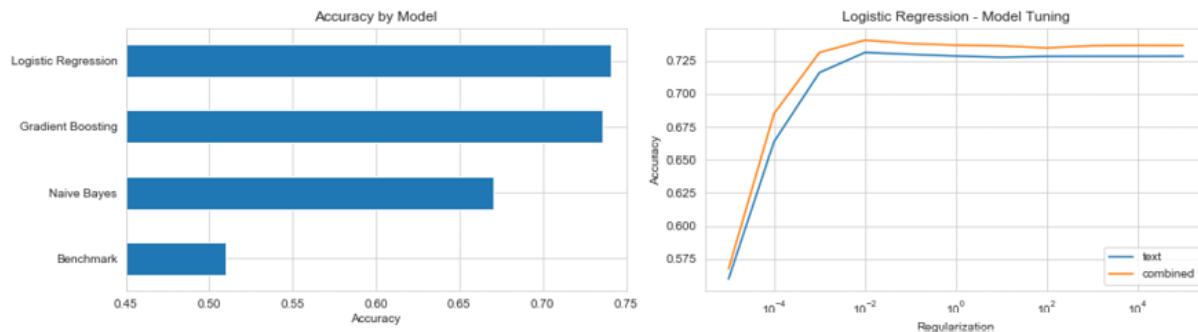


Figure 14.7: Test performance on combined data (all models, left) and for logistic regression with varying regularization

Summary

In this chapter, we explored numerous techniques and options to process unstructured data with the goal of extracting semantically meaningful numerical features for use in ML models.

We covered the basic tokenization and annotation pipeline and illustrated its implementation for multiple languages using spaCy and TextBlob. We built on these results to build a document model based on the bag-of-words model to represent documents as numerical vectors. We learned how to refine the preprocessing pipeline and then used the vectorized text data for classification and sentiment analysis.

We have two more chapters on alternative text data. In the next chapter, we will learn how to summarize texts using unsupervised learning to identify latent topics. Then, in *Chapter 16, Word Embeddings for Earnings Calls and SEC Filings*, we will learn how to represent words as vectors that reflect the context of word usage, a technique that has been used very successfully to provide richer text features for various classification tasks.

Topic Modeling – Summarizing Financial News

In the last chapter, we used the **bag-of-words (BOW)** model to convert unstructured text data into a numerical format. This model abstracts from word order and represents documents as word vectors, where each entry represents the relevance of a token to the document. The resulting **document-term matrix (DTM)**—or transposed as the term-document matrix—is useful for comparing documents to each other or a query vector for similarity based on their token content and, therefore, finding the proverbial needle in a haystack. It provides informative features to classify documents, such as in our sentiment analysis examples.

However, this document model produces both high-dimensional data and very sparse data, yet it does little to summarize the content or get closer to understanding what it is about. In this chapter, we will use **unsupervised machine learning** to extract hidden themes from documents using **topic modeling**. These themes can produce detailed insights into a large body of documents in an automated way. They are very useful in order to understand the haystack itself and allow us to tag documents based on their affinity with the various topics.

Topic models generate sophisticated, interpretable text features that can be a first step toward extracting trading signals from large collections of documents. They speed up the review of documents, help identify and cluster similar documents, and support predictive modeling.

Applications include the unsupervised discovery of potentially insightful themes in company disclosures or earnings call transcripts, customer reviews, or contracts. Furthermore, the document-topic associations facilitate the labeling by assigning, for example, sentiment metrics or, more directly, subsequent relevant asset returns.

More specifically, after reading this chapter, you'll understand:

- How topic modeling has evolved, what it achieves, and why it matters
- Reducing the dimensionality of the DTM using **latent semantic indexing (LSI)**
- Extracting topics with **probabilistic latent semantic analysis (pLSA)**

- How **latent Dirichlet allocation (LDA)** improves pLSA to become the most popular topic model
- Visualizing and evaluating topic modeling results
- Running LDA using sklearn and Gensim
- How to apply topic modeling to collections of earnings calls and financial news articles



You can find the code samples for this chapter and links to additional resources in the corresponding directory of the GitHub repository. The notebooks include color versions of the images.

Learning latent topics – Goals and approaches

Topic modeling discovers hidden themes that capture semantic information beyond individual words in a body of documents. It aims to address a key challenge for a machine learning algorithm that learns from text data by transcending the lexical level of "what actually has been written" to the semantic level of "what was intended." The resulting topics can be used to annotate documents based on their association with various topics.

In practical terms, topic models automatically **summarize large collections of documents** to facilitate organization and management as well as search and recommendations. At the same time, it enables the understanding of documents to the extent that humans can interpret the descriptions of topics.

Topic models also mitigate the **curse of dimensionality** that often plagues the BOW model; representing documents with high-dimensional, sparse vectors can make similarity measures noisy, lead to inaccurate distance measurements, and result in the overfitting of text classification models.

Moreover, the BOW model loses context as well as semantic information since it ignores word order. It is also unable to capture synonymy (where several words have the same meaning) or polysemy (where one word has several meanings). As a result of the latter, document retrieval or similarity search may miss the point when the documents are not indexed by the terms used to search or compare.

These shortcomings of the BOW model prompt the question: how can we learn meaningful topics from data that facilitate a more productive interaction with documentary data?

Initial attempts by topic models to improve on the vector space model (developed in the mid-1970s) applied linear algebra to reduce the dimensionality of the DTM. This approach is similar to the algorithm that we discussed as principal component analysis in *Chapter 13, Data-Driven Risk Factors and Asset Allocation with Unsupervised Learning*. While effective, it is difficult to evaluate the results of these models without a benchmark model. In response, probabilistic models have emerged that assume an explicit document generation process and provide algorithms to reverse engineer this process and recover the underlying topics.

The following table highlights key milestones in the model evolution, which we will address in more detail in the following sections:

Model	Year	Description
Latent semantic indexing (LSI)	1988	Captures the semantic document-term relationship by reducing the dimensionality of the word space
Probabilistic latent semantic analysis (pLSA)	1999	Reverse engineers a generative process that assumes words generate a topic and documents as a mix of topics
Latent Dirichlet allocation (LDA)	2003	Adds a generative process for documents: a three-level hierarchical Bayesian model

Latent semantic indexing

Latent semantic indexing (LSI)—also called **latent semantic analysis (LSA)**—set out to improve the results of queries that omitted relevant documents containing synonyms of query terms (Dumais et al. 1988). Its goal was to model the relationships between documents and terms so that it could predict that a term should be associated with a document, even though, because of the variability in word use, no such association was observed.

LSI uses linear algebra to find a given number k of latent topics by decomposing the DTM. More specifically, it uses the **singular value decomposition (SVD)** to find the best lower-rank DTM approximation using k singular values and vectors. In other words, LSI builds on some of the dimensionality reduction techniques we encountered in *Chapter 13, Data-Driven Risk Factors and Asset Allocation with Unsupervised Learning*. The authors also experimented with hierarchical clustering but found it too restrictive for this purpose.

In this context, SVD identifies a set of uncorrelated indexing variables or factors that represent each term and document by its vector of factor values. *Figure 15.1* illustrates how SVD decomposes the DTM into three matrices: two matrices that contain orthogonal singular vectors and a diagonal matrix with singular values that serve as scaling factors.

Assuming some correlation in the input DTM, singular values decay in value. Therefore, selecting the T -largest singular values yields a lower-dimensional approximation of the original DTM that loses relatively little information. In the compressed version, the rows or columns that had N items only have $T < N$ entries.

The LSI decomposition of the DTM can be interpreted as shown in *Figure 15.1*:

- The first $M \times T$ matrix represents the relationships between documents and topics.
- The diagonal matrix scales the topics by their corpus strength.
- The third matrix models the term-topic relationship.

$$\begin{array}{c}
 \begin{array}{c} \text{N Terms} \\ \vdots \quad \cdots \quad \vdots \\ \dots \end{array} & \begin{array}{c} \text{Document-Topic} \\ \text{Similarity} \\ \vdots \quad \cdots \quad \vdots \\ \dots \end{array} & \begin{array}{c} \text{Topic} \\ \text{Strength} \\ \ddots \quad \Sigma \quad \ddots \\ \dots \end{array} & \begin{array}{c} \text{Term-Topic} \\ \text{Similarity} \\ \vdots \quad \cdots \quad \vdots \\ \dots \end{array} \\
 \text{M Docs} & \left(\begin{array}{c} \vdots \quad \cdots \quad \vdots \\ \vdots \quad \ddots \quad \vdots \\ \dots \end{array} \right) = \left(\begin{array}{c} \vdots \quad \cdots \quad \vdots \\ \vdots \quad \ddots \quad \vdots \\ \dots \end{array} \right) \underbrace{\left[\begin{array}{c} \ddots \quad \Sigma \quad \ddots \\ \vdots \quad \vdots \quad \vdots \end{array} \right]}_{\substack{\text{Singular Vectors} \\ (M \times N)}} \left(\begin{array}{c} \vdots \quad \cdots \quad \vdots \\ \vdots \quad \ddots \quad \vdots \\ \dots \end{array} \right) & \text{Singular Values} & \text{Singular Vectors} \\
 \text{Document-Term} & & (N \times N) & (N \times M) \\
 \text{Matrix} & & & \\
 \end{array} \\
 \begin{array}{c} \text{1. Reduce dimensionality using } T < N \text{ singular values} \\ \text{2. Estimate document-topic matrix using } U_T \Sigma_T \end{array}$$

Figure 15.1: LSI and the SVD

The rows of the matrix produced by multiplying the first two matrices $U_T \Sigma_T$ correspond to the locations of the original documents projected into the latent topic space.

How to implement LSI using sklearn

We will illustrate LSI using the BBC articles data that we introduced in the last chapter because they are small enough for quick training and allow us to compare topic assignments with category labels. Refer to the notebook `latent_semantic_indexing` for additional implementation details.

We begin by loading the documents and creating a train and (stratified) test set with 50 articles. Then, we vectorize the data using `TfidfVectorizer` to obtain weighted DTM counts and filter out words that appear in less than 1 percent or more than 25 percent of the documents, as well as generic stopwords, to obtain a vocabulary of around 2,900 words:

```

vectorizer = TfidfVectorizer(max_df=.25, min_df=.01,
                            stop_words='english',
                            binary=False)
train_dtm = vectorizer.fit_transform(train_docs.article)
test_dtm = vectorizer.transform(test_docs.article)

```

We use scikit-learn's `TruncatedSVD` class, which only computes the k -largest singular values, to reduce the dimensionality of the DTM. The deterministic `arpack` algorithm delivers an exact solution, but the default "randomized" implementation is more efficient for large matrices.

We compute five topics to match the five categories, which explain only 5.4 percent of the total DTM variance, so a larger number of topics would be reasonable:

```

svd = TruncatedSVD(n_components=5, n_iter=5, random_state=42)
svd.fit(train_dtm)
svd.explained_variance_ratio_.sum()
0.05382357286057269

```

LSI identifies a new orthogonal basis for the DTM that reduces the rank to the number of desired topics. The `.transform()` method of the trained `svd` object projects the documents into the new topic space. This space results from reducing the dimensionality of the document vectors and corresponds to the $U_T \Sigma_T$ transformation illustrated earlier in this section:

```

train_doc_topics = svd.transform(train_dtm)
train_doc_topics.shape
(2175, 5)

```

We can sample an article to view its location in the topic space. We draw a "Politics" article that is most (positively) associated with topics 1 and 2:

```

i = randint(0, len(train_docs))
train_docs.iloc[i, :2].append(pd.Series(doc_topics[i], index=topic_labels))
Category          Politics
Heading    What the election should really be about?
Topic 1            0.33
Topic 2            0.18
Topic 3            0.12
Topic 4            0.02
Topic 5            0.06

```

The topic assignments for this sample align with the average topic weights for each category illustrated in *Figure 15.2* ("Politics" is the rightmost bar). They illustrate how LSI expresses the k topics as directions in a k -dimensional space (the notebook includes a projection of the average topic assignments per category into two-dimensional space).

Each category is clearly defined, and the test assignments match with train assignments. However, the weights are both positive and negative, making it more difficult to interpret the topics.

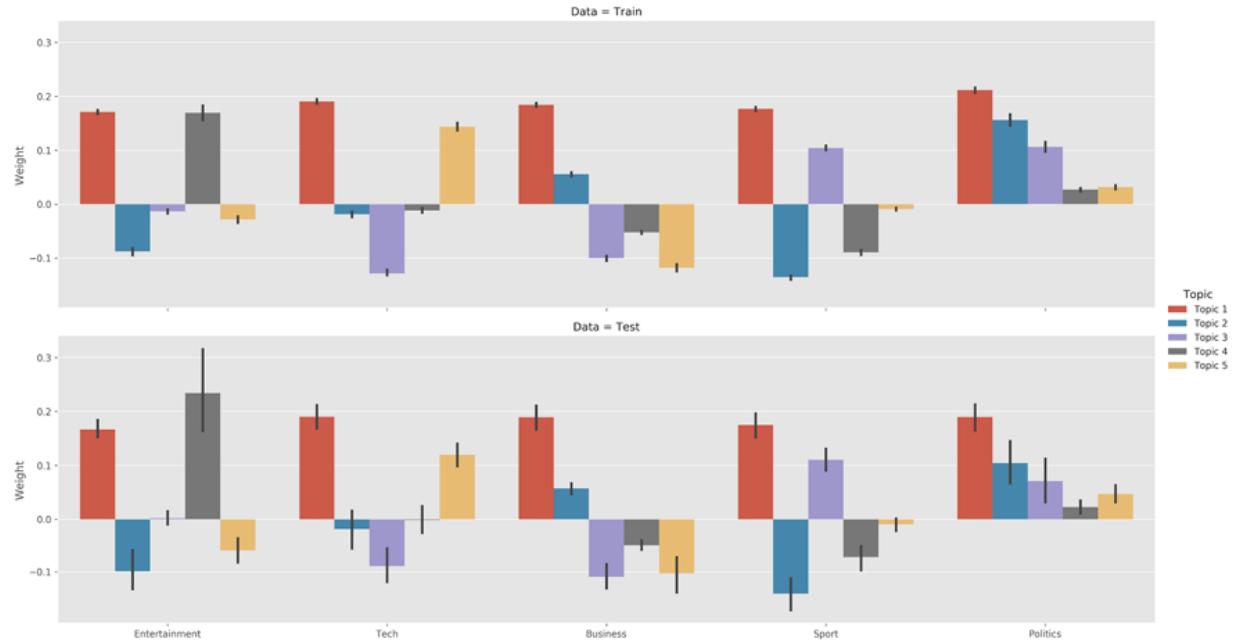


Figure 15.2: LSI topic weights for train and test data

We can also display the words that are most closely associated with each topic (in absolute terms). The topics appear to capture some semantic information but are not clearly differentiated (refer to *Figure 15.3*).



Figure 15.3: Top 10 words per LSI topic

Strengths and limitations

The strengths of LSI include the removal of noise and the mitigation of the curse of dimensionality. It also captures some semantic aspects, like synonymy, and clusters both documents and terms via their topic associations. Furthermore, it does not require knowledge of the document language, and both information retrieval queries and document comparisons are easy to do.

However, the results of LSI are difficult to interpret because topics are word vectors with both positive and negative entries. In addition, there is no underlying model that would permit the evaluation of fit or provide guidance when selecting the number of dimensions or topics to use.

Probabilistic latent semantic analysis

Probabilistic latent semantic analysis (pLSA) takes a **statistical perspective** on LSI/LSA and creates a generative model to address the lack of theoretical underpinnings of LSA (Hofmann 2001).

pLSA explicitly models the probability word w appearing in document d , as described by the DTM as a mixture of conditionally independent multinomial distributions that involve topics t .

There are both **symmetric and asymmetric formulations** of how word-document co-occurrences come about. The former assumes that both words and documents are generated by the latent topic class. In contrast, the asymmetric model assumes that topics are selected given the document, and words result in a second step given the topic.

$$P(w, d) = \underbrace{\sum_t P(d|t)P(w|t)}_{\text{symmetric}} = \underbrace{P(d) \sum_t P(t|d)P(w|t)}_{\text{asymmetric}}$$

The number of topics is a **hyperparameter** chosen prior to training and is not learned from the data.

The **plate notation** in *Figure 15.4* describes the statistical dependencies in a probabilistic model. More specifically, it encodes the relationship just described for the asymmetric model. Each rectangle represents multiple items: the outer block stands for M documents, while the inner shaded rectangle symbolizes N words for each document. We only observe the documents and their content; the model infers the hidden or latent topic distribution:

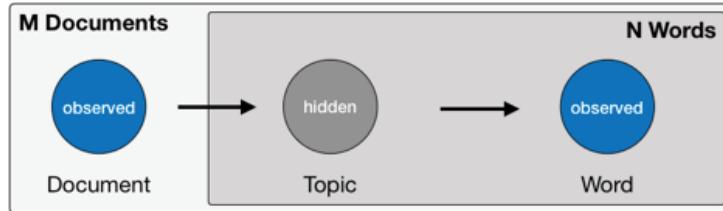


Figure 15.4: The statistical dependencies modeled by pLSA in plate notation

Let's now take a look at how we can implement this model in practice.

How to implement pLSA using sklearn

pLSA is equivalent to **non-negative matrix factorization (NMF)** using a Kullback-Leibler divergence objective (view the references on GitHub). Therefore, we can use the `sklearn.decomposition.NMF` class to implement this model following the LSI example.

Using the same train-test split of the DTM produced by `TfidfVectorizer`, we fit pLSA like so:

```
nmf = NMF(n_components=n_components,
           random_state=42,
           solver='mu',
           beta_loss='kullback-leibler',
           max_iter=1000)
nmf.fit(train_dtm)
```

We get a measure of the reconstruction error that is a substitute for the explained variance measure from earlier:

```
nmf.reconstruction_err_
316.2609400385988
```

Due to its probabilistic nature, pLSA produces only positive topic weights that result in more straightforward topic-category relationships for the test and training sets, as shown in *Figure 15.5*:

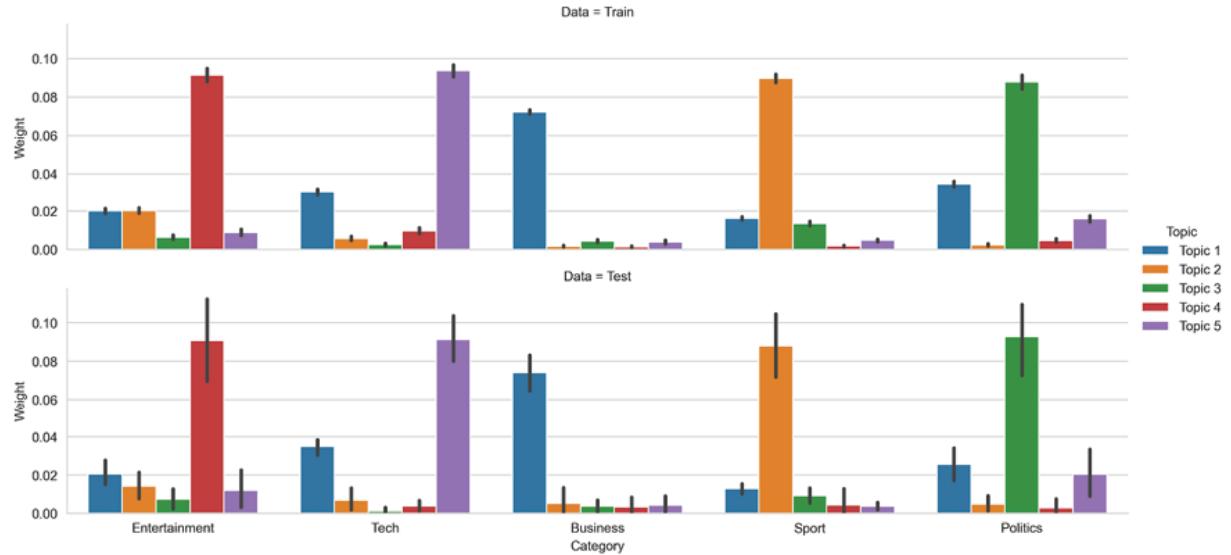


Figure 15.5: pLSA weights by topic for train and test data

We also note that the word lists that describe each topic begin to make more sense; for example, the "Entertainment" category is most directly associated with Topic 4, which includes the words "film," "star," and so forth, as you can see in *Figure 15.6*:



Figure 15.6: Top words per topic for pLSA

Strengths and limitations

The benefit of using a probability model is that we can now compare the performance of different models by evaluating the probability they assign to new documents given the parameters learned during training. It also means that the results have a clear probabilistic interpretation. In addition, pLSA captures more semantic information, including polysemy.

On the other hand, pLSA increases the computational complexity compared to LSI, and the algorithm may only yield a local as opposed to a global maximum. Finally, it does not yield a generative model for new documents because it takes them as given.

Latent Dirichlet allocation

Latent Dirichlet allocation (LDA) extends pLSA by adding a generative process for topics (Blei, Ng, and Jordan 2003). It is the most popular topic model because it tends to produce meaningful topics that humans can relate to, can assign topics to new documents, and is extensible. Variants of LDA models can include metadata, like authors or image data, or learn hierarchical topics.

How LDA works

LDA is a **hierarchical Bayesian model** that assumes topics are probability distributions over words, and documents are distributions over topics. More specifically, the model assumes that topics follow a sparse Dirichlet distribution, which implies that documents reflect only a small set of topics, and topics use only a limited number of terms frequently.

The Dirichlet distribution

The Dirichlet distribution produces probability vectors that can be used as a discrete probability distribution. That is, it randomly generates a given number of values that are positive and sum to one. It has a parameter α of positive real value that controls the concentration of the probabilities. Values closer to zero mean that only a few values will be positive and receive most of the probability mass. *Figure 15.7* illustrates three draws of size 10 for $\alpha = 0.1$:

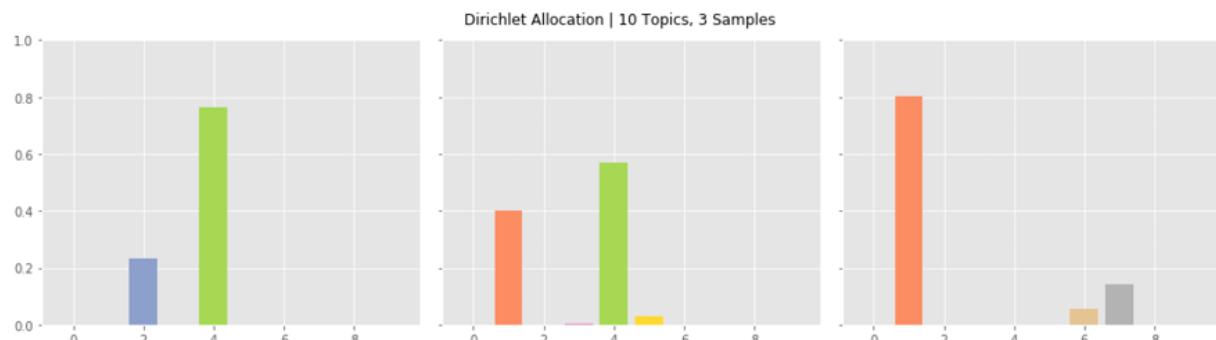


Figure 15.7: Three draws from the Dirichlet distribution

The notebook `dirichlet_distribution` contains a simulation that lets you experiment with different parameter values.

The generative model

The LDA topic model assumes the following generative process when an author adds an article to a body of documents:

1. Randomly mix a small subset of topics with proportions defined by the Dirichlet probabilities.
2. For each word in the text, select one of the topics according to the document-topic probabilities.
3. Select a word from the topic's word list according to the topic-word probabilities.

As a result, the article content depends on the weight of each topic and the terms that make up each topic. The Dirichlet distribution governs the selection of topics for documents and words for topics. It encodes the idea that a document only covers a few topics, while each topic uses only a small number of words frequently.

The **plate notation** for the LDA model in *Figure 15.8* summarizes these relationships and highlights the key model parameters:

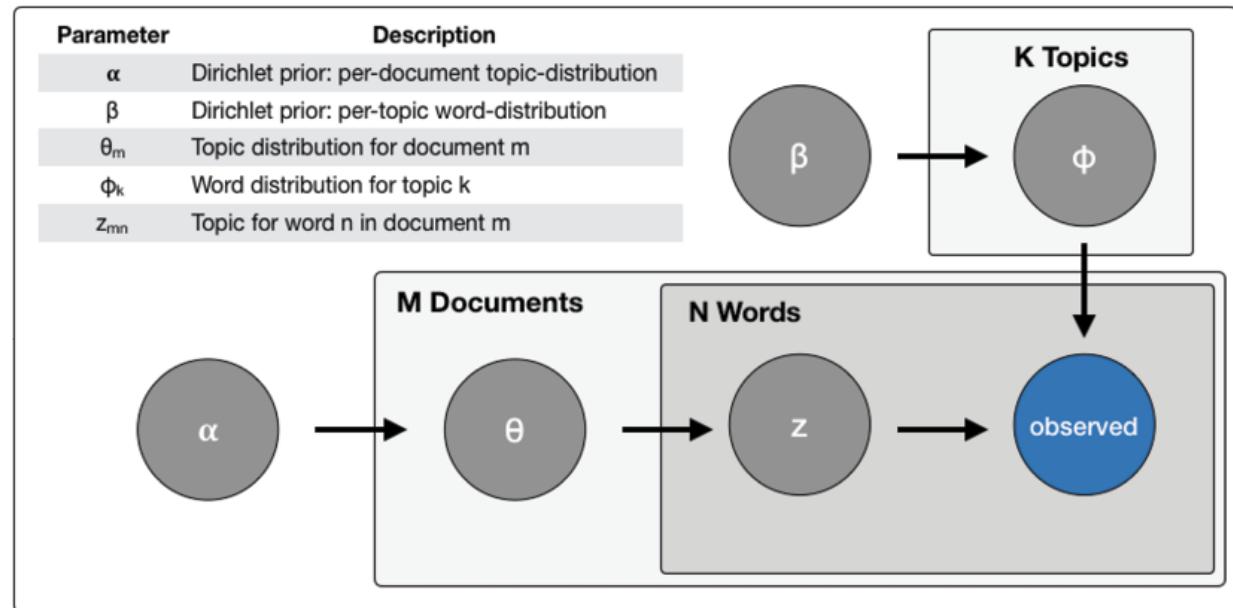


Figure 15.8: The statistical dependencies of the LDA model in plate notation

Reverse engineering the process

The generative process is clearly fictional but turns out to be useful because it permits the recovery of the various distributions. The LDA algorithm reverse engineers the work of the imaginary author and arrives at a summary of the document-topic-word relationships that concisely describes:

- The percentage contribution of each topic to a document
- The probabilistic association of each word with a topic

LDA solves the **Bayesian inference** problem of recovering the distributions from the body of documents and the words they contain by reverse engineering the assumed content generation process. The original paper by Blei et al. (2003) uses **variational Bayes (VB)** to approximate the posterior distribution. Alternatives include Gibbs sampling and expectation propagation. We will illustrate, shortly, the implementations by the `sklearn` and `Gensim` libraries.

How to evaluate LDA topics

Unsupervised topic models do not guarantee that the result will be meaningful or interpretable, and there is no objective metric to assess the quality of the result as in supervised learning. Human topic evaluation is considered the gold standard, but it is potentially expensive and not readily available at scale.

Two options to evaluate results more objectively include **perplexity**, which evaluates the model on unseen documents, and **topic coherence** metrics, which aim to evaluate the semantic quality of the uncovered patterns.

Perplexity

Perplexity, when applied to LDA, measures how well the topic-word probability distribution recovered by the model predicts a sample of unseen text documents. It is based on the entropy $H(p)$ of this distribution p and is computed with respect to the set of tokens w :

$$2^{h(p)} = 2^{-\sum_w p(w) \log_2 p(w)}$$

Measures closer to zero imply the distribution is better at predicting the sample.

Topic coherence

Topic coherence measures the semantic consistency of the topic model results, that is, whether humans would perceive the words and their probabilities associated with topics as meaningful.

To this end, it scores each topic by measuring the degree of semantic similarity between the words most relevant to the topic. More specifically, coherence measures are based on the probability of observing the set of words W that defines a topic together.

There are two measures of coherence that have been designed for LDA and are shown to align with human judgments of topic quality, namely the UMass and the UCI metrics.

The UCI metric (Stevens et al. 2012) defines a word pair's score to be the sum of the **pointwise mutual information (PMI)** between two distinct pairs of (top) topic words w_i , $w_j \in w$ and a smoothing factor ϵ :

$$\text{coherence}_{\text{UCI}} = \sum_{(w_i, w_j) \in w} \log \frac{p(w_i, w_j) + \epsilon}{p(w_i)p(w_j)}$$

The probabilities are computed from word co-occurrence frequencies in a sliding window over an external corpus like Wikipedia so that this metric can be thought of as an external comparison to semantic ground truth.

In contrast, the UMass metric (Mimno et al. 2011) uses the co-occurrences in a number of documents D from the training corpus to compute a coherence score:

$$\text{coherence}_{\text{UMass}} = \sum_{(w_i, w_j) \in w} \log \frac{D(w_i, w_j) + \epsilon}{D(w_j)}$$

Rather than comparing the model result to extrinsic ground truth, this measure reflects intrinsic coherence. Both measures have been evaluated to align well with human judgment (Röder, Both, and Hinneburg 2015). In both cases, values closer to zero imply that a topic is more coherent.

How to implement LDA using sklearn

We will use the BBC data as before and train an LDA model using sklearn's `decomposition.LatentDirichletAllocation` class with five topics (refer to the sklearn documentation for details on the parameters and the notebook `lda_with_sklearn` for implementation details):

```
lda_opt = LatentDirichletAllocation(n_components=5,
                                    n_jobs=-1,
                                    max_iter=500,
                                    learning_method='batch',
                                    evaluate_every=5,
                                    verbose=1,
```

```

        random_state=42)
ldat.fit(train_dtm)
LatentDirichletAllocation(batch_size=128, doc_topic_prior=None,
                        evaluate_every=5, learning_decay=0.7, learning_method='batch',
                        learning_offset=10.0, max_doc_update_iter=100, max_iter=500,
                        mean_change_tol=0.001, n_components=5, n_jobs=-1,
                        n_topics=None, perp_tol=0.1, random_state=42,
                        topic_word_prior=None, total_samples=1000000.0, verbose=1)

```

The model tracks the in-sample perplexity during training and stops iterating once this measure stops improving. We can persist and load the result as usual with sklearn objects:

```

joblib.dump(lda, model_path / 'lda_opt.pkl')
lda_opt = joblib.load(model_path / 'lda_opt.pkl')

```

How to visualize LDA results using pyLDAvis

Topic visualization facilitates the evaluation of topic quality using human judgment. pyLDAvis is a Python port of LDAvis, developed in R and `d3.js` (Sievert and Shirley 2014). We will introduce the key concepts; each LDA application notebook contains examples.

pyLDAvis displays the global relationships among topics while also facilitating their semantic evaluation by inspecting the terms most closely associated with each individual topic and, inversely, the topics associated with each term. It also addresses the challenge that terms that are frequent in a corpus tend to dominate the distribution over words that define a topic.

To this end, LDAvis introduces the **relevance** r of term w to topic t . The relevance produces a flexible ranking of terms by topic, by computing a weighted average of two metrics:

- The degree of association of topic t with term w , expressed as the conditional probability $p(w | t)$
- The saliency, or lift, which measures how the frequency of term w for the topic t , $p(w | t)$, compares to its overall frequency across all documents, $p(w)$

More specifically, we can compute the relevance r for a term w and a topic t given a user-defined weight $0 \leq \lambda \leq 1$, like the following:

$$r(w, t | \lambda) = \lambda \log(p(w|t)) + (1 - \lambda) \log \frac{p(w|t)}{p(w)}$$

The tool allows the user to interactively change λ to adjust the relevance, which updates the ranking of terms. User studies have found $\lambda = 0.6$ to produce the most plausible results.

How to implement LDA using Gensim

Gensim is a specialized **natural language processing (NLP)** library with a fast LDA implementation and many additional features. We will also use it in the next chapter on word vectors (refer to the notebook `lda_with_gensim` for details and the installation directory for related instructions).

We convert the DTM produced by sklearn's `CountVectorizer` or `TfidfVectorizer` into Gensim data structures as follows:

```
train_corpus = Sparse2Corpus(train_dtm, documents_columns=False)
test_corpus = Sparse2Corpus(test_dtm, documents_columns=False)
id2word = pd.Series(vectorizer.get_feature_names()).to_dict()
```

Gensim's LDA algorithm includes numerous settings:

```
LdaModel(corpus=None,
          num_topics=100,
          id2word=None,
          distributed=False,
          chunksize=2000, # No of doc per training chunk.
          passes=1, # No of passes through corpus during training
          update_every=1, # No of docs to be iterated through per update
          alpha='symmetric',
          eta=None, # a-priori belief on word probability
          decay=0.5, # % of Lambda forgotten when new doc is examined
          offset=1.0, # controls slow down of first few iterations.
          eval_every=10, # how often estimate log perplexity (costly)
          iterations=50, # Max. of iterations through the corpus
          gamma_threshold=0.001, # Min. change in gamma to continue
          minimum_probability=0.01, # Filter topics with lower probability
          random_state=None,
          ns_conf=None,
          minimum_phi_value=0.01, # Lower bound on term probabilities
          per_word_topics=False, # Compute most word-topic probabilities
          callbacks=None,
          dtype=<class 'numpy.float32'>)
```

Gensim also provides an `LdaMulticore` model for parallel training that may speed up training using Python's multiprocessing features for parallel computation.

Model training just requires instantiating `LdaModel`, as follows:

```
lda_gensim = LdaModel(corpus=train_corpus,
                      num_topics=5,
                      id2word=id2word)
```

Gensim evaluates topic coherence, as introduced in the previous section, and shows the most important words per topic:

```
coherence = lda_gensim.top_topics(corpus=train_corpus, coherence='u_mass')
```

We can display the results as follows:

```
topic_coherence = []
topic_words = pd.DataFrame()
for t in range(len(coherence)):
    label = topic_labels[t]
    topic_coherence.append(coherence[t][1])
    df = pd.DataFrame(coherence[t][0], columns=[(label, 'prob'),
                                                (label, 'term')])
    df[(label, 'prob')] = df[(label, 'prob')].apply(
        lambda x: '{:.2%}'.format(x))
    topic_words = pd.concat([topic_words, df], axis=1)

topic_words.columns = pd.MultiIndex.from_tuples(topic_words.columns)
pd.set_option('expand_frame_repr', False)
print(topic_words.head())
```

This shows the following top words for each topic:

Topic 1		Topic 2		Topic 3		Topic 4		Topic 5	
Probability	Term	Probability	Term	Probability	Term	Probability	Term	Probability	Term
0.55%	online	0.90%	best	1.04%	mobile	0.64%	market	0.94%	labour
0.51%	site	0.87%	game	0.98%	phone	0.53%	growth	0.72%	blair
0.46%	game	0.62%	play	0.51%	music	0.52%	sales	0.72%	brown
0.45%	net	0.61%	won	0.48%	film	0.49%	economy	0.65%	election
0.44%	used	0.56%	win	0.48%	use	0.45%	prices	0.57%	united

The left panel of *Figure 15.9* displays the topic coherence scores, which highlight the decay of topic quality (at least, in part, due to the relatively small dataset):

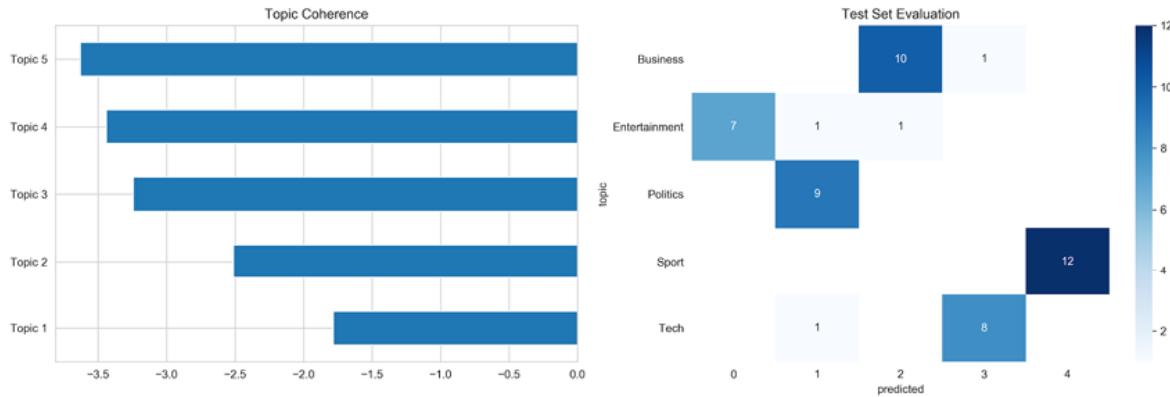


Figure 15.9: Topic coherence and test set assignments

The right panel displays the evaluation of our test set of 50 articles with our trained model. The model makes four mistakes for an accuracy of 92 percent.

Modeling topics discussed in earnings calls

In *Chapter 3, Alternative Data for Finance – Categories and Use Cases*, we learned how to scrape earnings call data from the SeekingAlpha site. In this section, we will illustrate topic modeling using this source. I'm using a sample of some 700 earnings call transcripts between 2018 and 2019. This is a fairly small dataset; for a practical application, we would need a larger dataset.

The directory `earnings_calls` contains several files with the code examples used in this section. Refer to the notebook `1da_earnings_calls` for details on loading, exploring, and preprocessing the data, as well as training and evaluating individual models, and the `run_experiments.py` file for the experiments described next.

Data preprocessing

The transcripts consist of individual statements by company representatives, an operator, and a Q&A session with analysts. We will treat each of these statements as separate documents, ignoring operator statements, to obtain 32,047 items with mean and median word counts of 137 and 62, respectively:

```
documents = []
for transcript in earnings_path.iterdir():
    content = pd.read_csv(transcript / 'content.csv')
```

```
documents.extend(content.loc[(content.speaker!='Operator') & (content.content.str.len() > len(documents))]
32047
```

We use spaCy to preprocess these documents, as illustrated in *Chapter 13, Data-Driven Risk Factors and Asset Allocation with Unsupervised Learning*, (refer to the notebook), and store the cleaned and lemmatized text as a new text file.

Exploration of the most common tokens, as shown in *Figure 15.10*, reveals domain-specific stopwords like "year" and "quarter" that we remove in a second step, where we also filter out statements with fewer than 10 words so that some 22,582 remain.

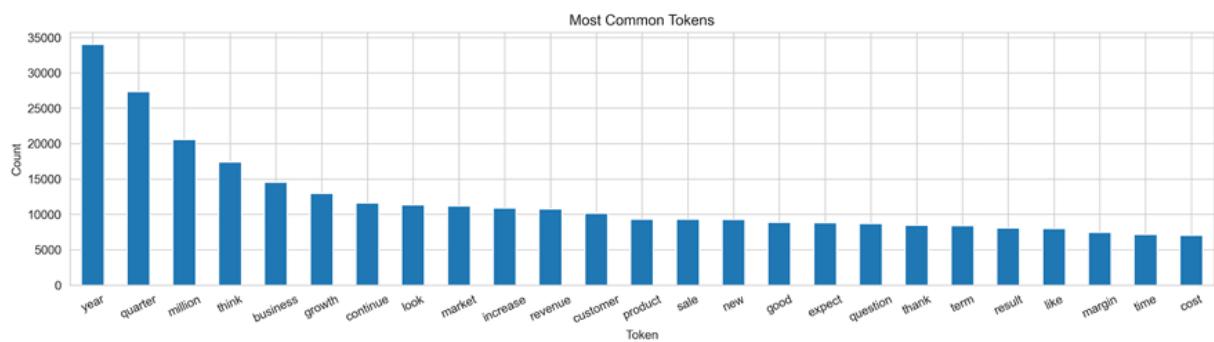


Figure 15.10: Most common earnings call tokens

Model training and evaluation

For illustration, we create a DTM containing terms appearing in between 0.5 and 25 percent of documents that results in 1,529 features. Now we proceed to train a 15-topic model using 25 passes over the corpus. This takes a bit over two minutes on a 4-core i7.

The top 10 words per topic, as shown in *Figure 15.11*, identify several distinct themes that range from obvious financial information to clinical trials (Topic 5), China and tariff issues (Topic 9), and technology issues (Topic 11).

0	statement	expense	service	brand	capital	patient	lot	technology	project	price	yes	cloud	store	maybe	chief
1	today	compare	platform	retail	billion	datum	thing	client	slide	china	guidance	service	comp	little	officer
2	financial	approximately	provide	channel	performance	study	way	need	production	pricing	say	deal	traffic	bit	today
3	release	gross	financial	digit	flow	program	people	process	asset	tariff	actually	enterprise	category	kind	president
4	risk	total	user	category	return	clinical	need	area	debt	thing	balance	security	team	sort	investor
5	gaap	income	value	consumer	improve	trial	different	team	month	inventory	basis	large	online	guess	financial
6	measure	basis	solution	launch	loan	phase	value	change	low	lot	mean	subscription	open	okay	join
7	information	prior	focus	performance	basis	month	yes	fuel	portfolio	half	change	datum	marketing	guy	bank
8	non	tax	deliver	segment	organic	fda	build	power	loan	yes	line	software	great	follow	executive
9	earning	period	technology	focus	low	process	focus	tool	average	demand	contract	platform	experience	wonder	capital
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	

Figure 15.11: Most important words for earnings call topics

Using pyLDAVis' relevance metric with a 0.6 weighting of unconditional frequency relative to lift, topic definitions become more intuitive, as illustrated in *Figure 15.12* for Topic 7 about China and the trade wars:

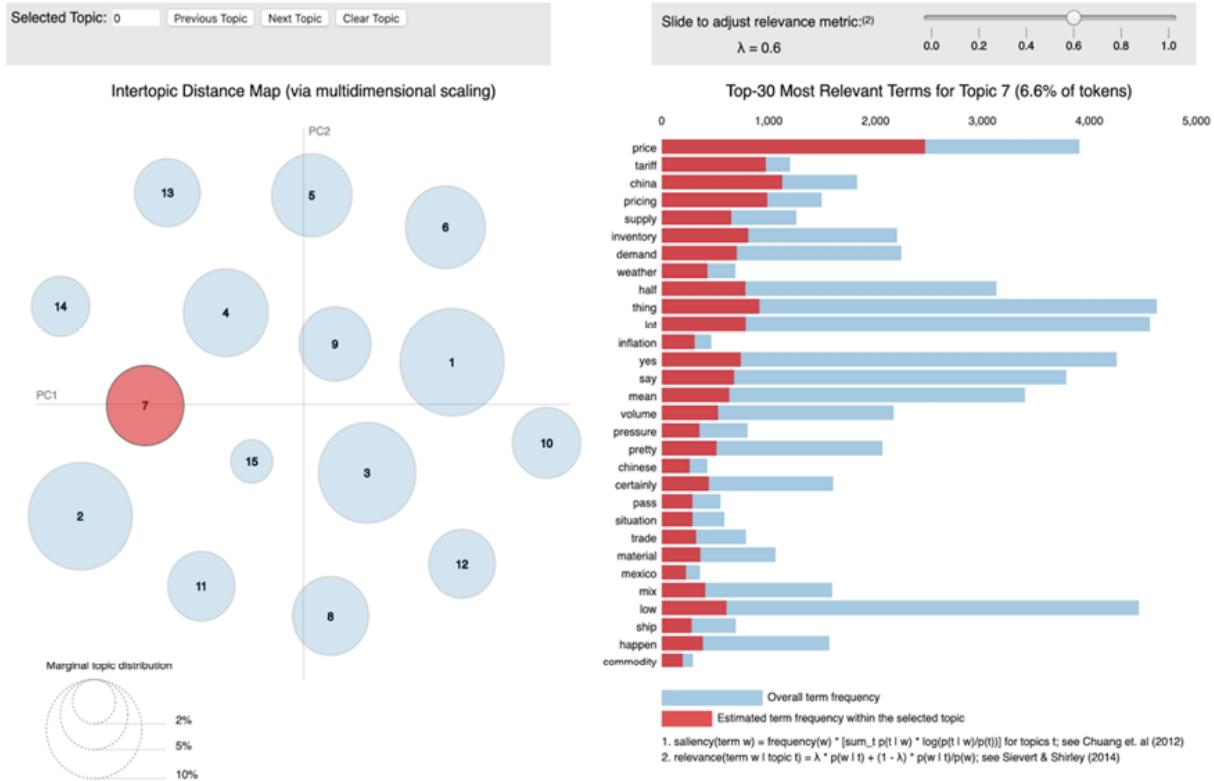


Figure 15.12: pyLDAVis' interactive topic explorer

The notebook also illustrates how you can look up documents by their topic association. In this case, an analyst can review relevant statements for nuances, use sentiment analysis to further process the topic-specific text data, or assign labels derived from market prices.

Running experiments

To illustrate the impact of different parameter settings, we run a few hundred experiments for different DTM constraints and model parameters. More specifically, we let the `min_df` and `max_df` parameters range from 50-500 words and 10 to 100 percent of documents, respectively, using alternatively binary and absolute counts. We then train LDA models with 3 to 50 topics, using 1 and 25 passes over the corpus.

The chart in *Figure 15.13* illustrates the results in terms of topic coherence (higher is better) and perplexity (lower is better). Coherence drops after 25-30 topics, and perplexity similarly increases.

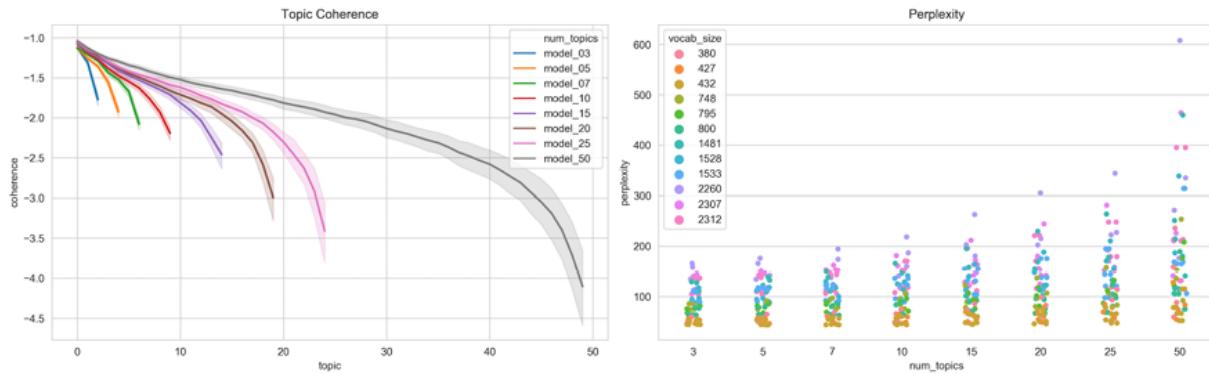


Figure 15.13: Impact of LDA hyperparameter settings on topic quality

The notebook includes regression results that quantify the relationships between parameters and outcomes. We generally get better results using absolute counts and a smaller vocabulary.

Topic modeling for with financial news

The notebook `lda_financial_news` contains an example of LDA applied to a subset of over 306,000 financial news articles from the first five months of 2018. The datasets have been posted on Kaggle, and the articles have been sourced from CNBC, Reuters, the Wall Street Journal, and more. The notebook contains download instructions.

We select the most relevant 120,000 articles based on their section titles with a total of 54 million tokens for an average word count of 429 words per article. To prepare the data for the LDA model, we rely on spaCy to remove numbers and punctuation and lemmatize the results.

Figure 15.14 highlights the remaining most frequent tokens and the article length distribution with a median length of 231 tokens; the 90th percentile is 642 words.

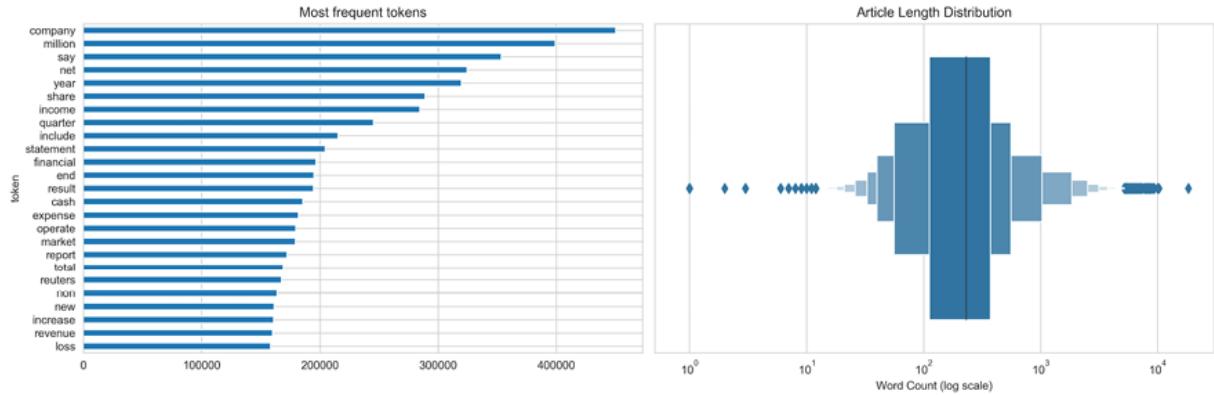


Figure 15.14: Corpus statistics for financial news data

In *Figure 15.15*, we show results for one model using a vocabulary of 3,570 tokens based on `min_df=0.005` and `max_df=0.1`, with a single pass to avoid the length training time for 15 topics. We can use the `top_topics` attribute of the trained `LdaModel` to obtain the most likely words for each topic (refer to the notebook for more details).

0	gaap	webcast	mr	clinical	korea	index	syria	police	britain	euro	facebook	trump	eikon	oil	vehicle
1	adjust	replay	client	patient	trump	inflation	iran	election	eu	stake	amazon	israel	dividend	qtrly	class
2	ebida	dial	leadership	pharmaceutical	russian	bond	syrian	court	brexit	loan	apple	house	min	energy	car
3	dilute	corporation	role	drug	korean	yield	turkey	kill	london	deutsche	cnbc	court	holding	gas	tesla
4	loan	eastern	brand	therapeutics	russia	euro	macron	opposition	union	bid	user	washington	sea	saudi	motor
5	liability	et	university	treatment	south	currency	force	arrest	italy	pound	store	israeli	bancorp	crude	esq
6	fiscal	host	health	trial	kim	central	merkel	vote	british	ipo	google	white	versus	production	attorney
7	distribution	audio	organization	cancer	sanction	feed	military	protest	prime	goldman	online	republican	fy	barrel	index
8	dividend	listen	digital	disease	moscow	forecast	germany	prime	uk	lender	game	donald	corporation	eikon	lp
9	margin	caller	software	phase	nuclear	hit	attack	attack	vote	regulator	app	senate	declare	boeing	electric
10	flow	section	corporation	study	tariff	drop	ai	corruption	school	london	story	investigation	appoint	uber	kong
11	consolidate	archive	healthcare	ida	chinese	benchmark	france	authority	pound	morgan	think	palestinian	thomson	airline	hong
12	gross	passcode	network	therapy	washington	economist	french	parliament	ireland	takeover	social	democrat	compensation	arabia	plaintiff
13	decrease	toll	expertise	medical	beijing	gold	turkish	myanmar	league	bengaluru	ad	jerusalem	qtrly	thomson	lawsuit
14	sec	presentation	excite	bitcoin	putin	tariff	rebel	political	gun	matt	brand	lawyer	trust	airbus	stake

Figure 15.15: Top 15 words for financial news topics

The topics outline several issues relevant to the time period, including Brexit (Topic 8), North Korea (Topic 4), and Tesla (Topic 14).

Gensim provides a `LdaMultiCore` implementation that allows for parallel training using Python's multiprocessing module and improves performance by 50 percent when using four workers. More workers do not further reduce training time, though, due to I/O bottlenecks.

Summary

In this chapter, we explored the use of topic modeling to gain insights into the content of a large collection of documents. We covered latent semantic indexing that uses

dimensionality reduction of the DTM to project documents into a latent topic space. While effective in addressing the curse of dimensionality caused by high-dimensional word vectors, it does not capture much semantic information. Probabilistic models make explicit assumptions about the interplay of documents, topics, and words that allow algorithms to reverse engineer the document generation process and evaluate the model fit on new documents. We learned that LDA is capable of extracting plausible topics that allow us to gain a high-level understanding of large amounts of text in an automated way, while also identifying relevant documents in a targeted way.

In the next chapter, we will learn how to train neural networks that embed individual words in a high-dimensional vector space that captures important semantic information and allows us to use the resulting word vectors as high-quality text features.

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.

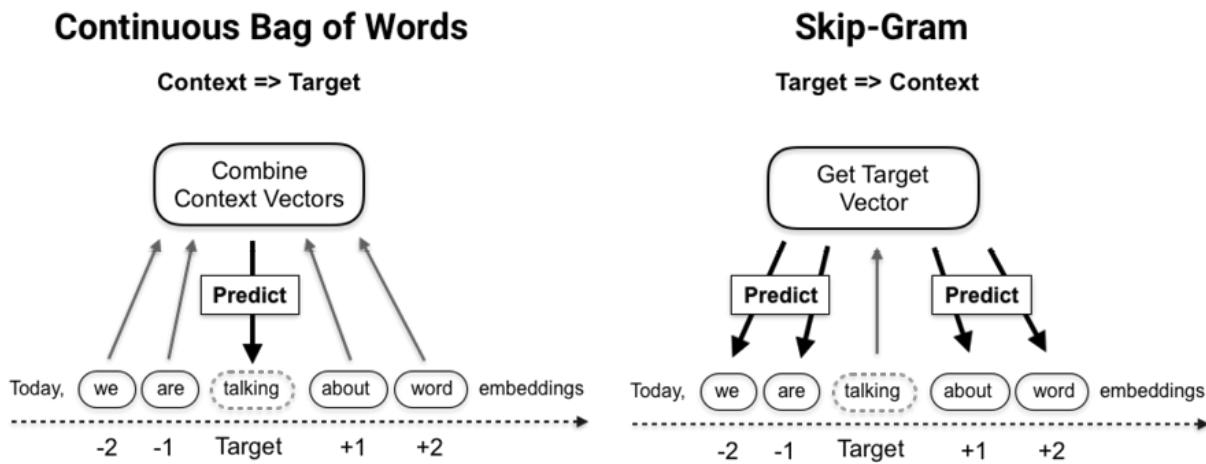


Figure 16.1: Continuous-bag-of-words versus skip-gram processing logic

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 :

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

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 single 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:

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

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":

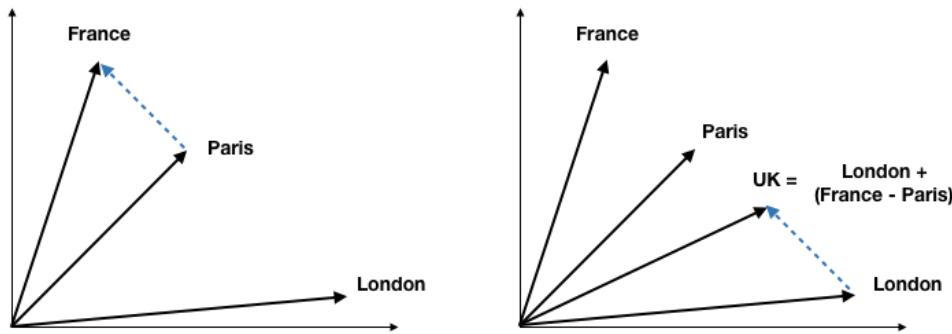


Figure 16.2: Embedding vector arithmetic

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	decrease	decreases	describe	describes
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 of 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:

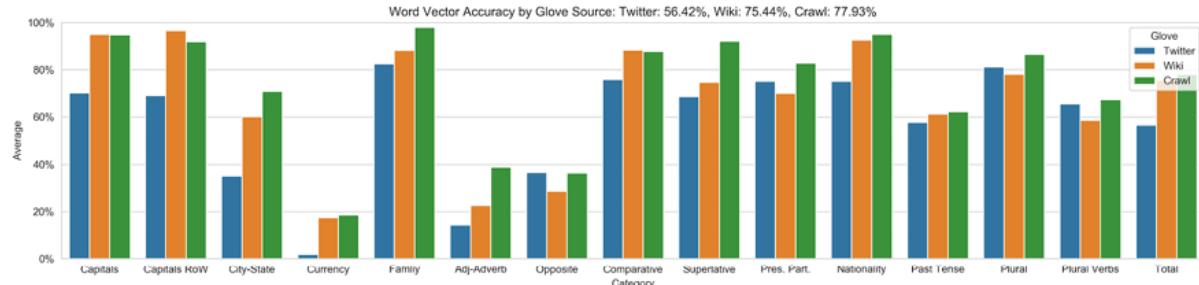


Figure 16.3: GloVe accuracy on word2vec analogies

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:

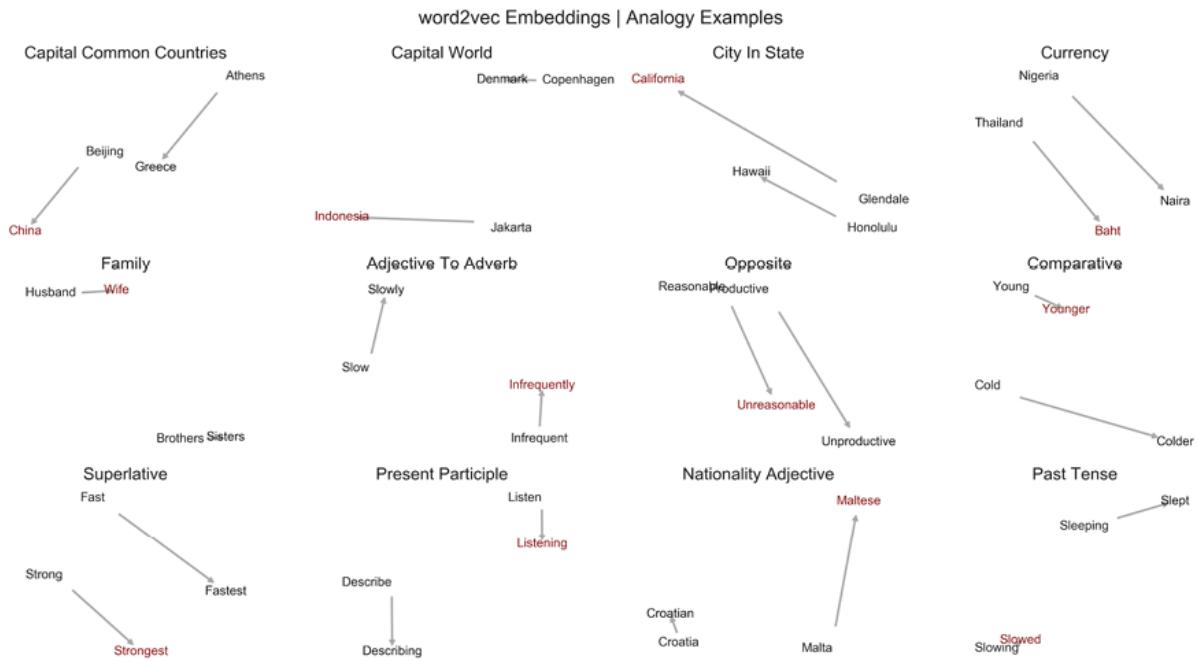


Figure 16.4: 2D visualization of selected analogy embeddings

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 `1da_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(SAMPLE_SIZE* len(sentences)), replace=False)).split()

```

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 $59,617 \times 200$ 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:

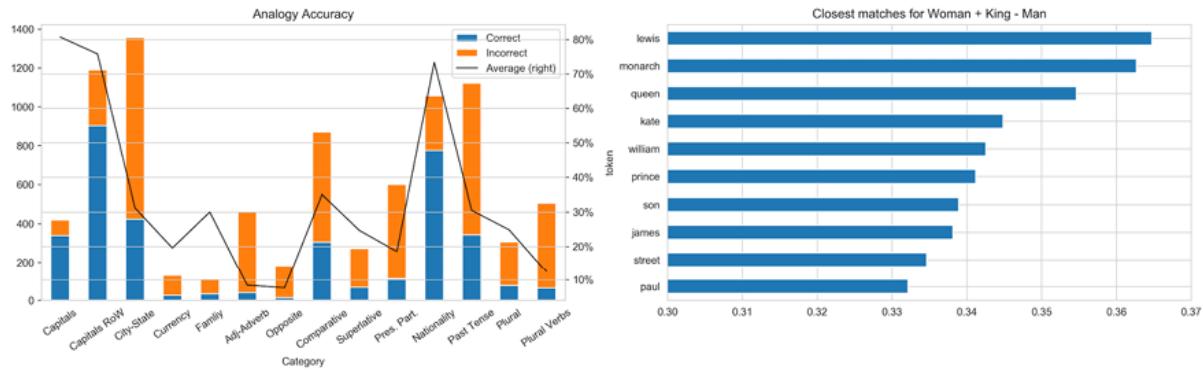


Figure 16.6: Analogy accuracy by category and for a specific example

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()
similar = pd.DataFrame()
for token in np.random.choice(most_common.token, size=10, p=most_common.p):
    similar[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	4
profiles	profile	users	political_consultancy_cambridge_analytica	sophisticated	facebook

divestments	divestitures	acquisitions	takeovers	bayer	consolidation
readiness	training	military	command	air_force	preparations
arsenal	nuclear_weapons	russia	ballistic_missile	weapons	hezbollah
supply_disruptions	disruptions	raw_material	disruption	prices	downturn

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', '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 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. 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:

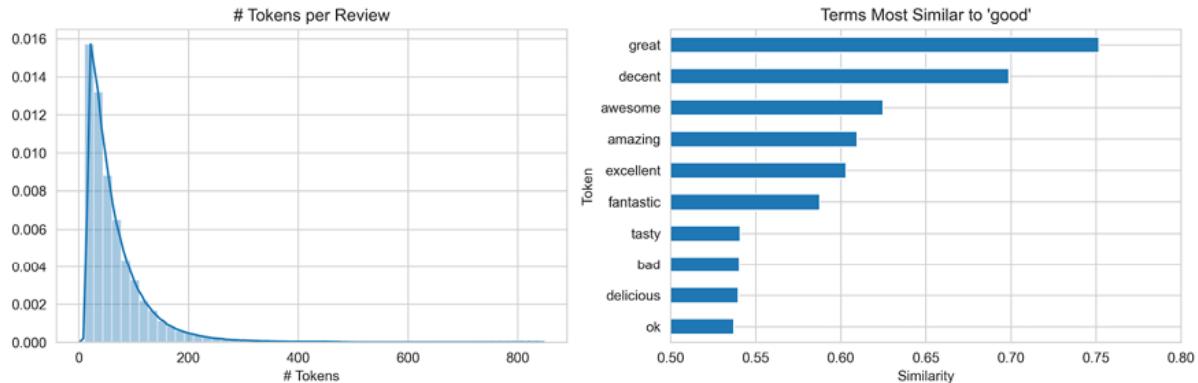


Figure 16.7: Histogram of the number of tokens per review (left) and terms most similar to the token 'good'

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:

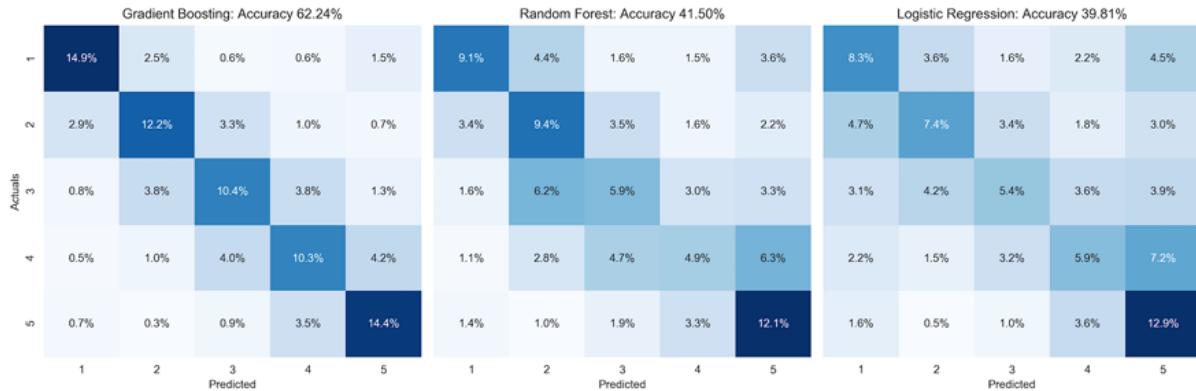


Figure 16.8: Confusion matrix and test accuracy for alternative models

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 $24 \times 16 = 384$ 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 pretrained 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.

Deep Learning for Trading

This chapter kicks off Part 4, which covers how several **deep learning (DL)** modeling techniques can be useful for investment and trading. DL has achieved numerous **breakthroughs in many domains**, ranging from image and speech recognition to robotics and intelligent agents that have drawn widespread attention and revived large-scale research into **artificial intelligence (AI)**. The expectations are high that the rapid development will continue and many more solutions to difficult practical problems will emerge.

In this chapter, we will present **feedforward neural networks** to introduce key elements of working with neural networks relevant to the various DL architectures covered in the following chapters. More specifically, we will demonstrate how to train large models efficiently using the **backpropagation algorithm** and manage the risks of overfitting. We will also show how to use the popular TensorFlow 2 and PyTorch frameworks, which we will leverage throughout Part 4.

Finally, we will develop, backtest, and evaluate a trading strategy based on signals generated by a deep feedforward neural network. We will design and tune the neural network and analyze how key hyperparameter choices affect its performance.

In summary, after reading this chapter and reviewing the accompanying notebooks, you will know about:

- How DL solves AI challenges in complex domains
- Key innovations that have propelled DL to its current popularity
- How feedforward networks learn representations from data
- Designing and training deep **neural networks (NNs)** in Python
- Implementing deep NNs using Keras, TensorFlow, and PyTorch
- Building and tuning a deep NN to predict asset returns
- Designing and backtesting a trading strategy based on deep NN signals

In the following chapters, we will build on this foundation to design various architectures suitable for different investment applications with a particular focus on alternative text and image data.

These include **recurrent neural networks (RNNs)** tailored to sequential data such as time series or natural language, and **convolutional neural networks (CNNs)**, which are particularly well suited to image data but can also be used with time-series data. We will also cover deep unsupervised learning, including autoencoders and **generative adversarial networks (GANs)** as well as reinforcement learning to train agents that interactively learn from their environment.



You can find the code samples for this chapter and links to additional resources in the corresponding directory of the GitHub repository. The notebooks include color versions of the images.

Deep learning – what's new and why it matters

The **machine learning** (ML) algorithms covered in *Part 2* work well on a wide variety of important problems, including on text data, as demonstrated in *Part 3*. They have been less successful, however, in solving central AI problems such as recognizing speech or classifying objects in images. These limitations have motivated the development of DL, and the recent DL breakthroughs have greatly contributed to a resurgence of interest in AI. For a comprehensive introduction that includes and expands on many of the points in this section, see Goodfellow, Bengio, and Courville (2016), or for a much shorter version, see LeCun, Bengio, and Hinton (2015).

In this section, we outline how DL overcomes many of the limitations of other ML algorithms. These limitations particularly constrain performance on high-dimensional and unstructured data that requires sophisticated efforts to extract informative features.

The ML techniques we covered in *Parts 2* and *3* are best suited for processing structured data with well-defined features. We saw, for example, how to convert text data into tabular data using the document-text matrix in *Chapter 14, Text Data for Trading – Sentiment Analysis*. DL overcomes the **challenge of designing informative features**, possibly by hand, by learning a representation of the data that better captures its characteristics with respect to the outcome.

More specifically, we'll see how DL learns a **hierarchical representation of the data**, and why this approach works well for high-dimensional, unstructured data. We will describe how NNs

employ a multilayered, deep architecture to compose a set of nested functions and discover a hierarchical structure. These functions compute successive and increasingly abstract representations of the data in each layer based on the learning of the previous layer. We will also look at how the backpropagation algorithm adjusts the network parameters so that these representations best meet the model's objective.

We will also briefly outline how DL fits into the evolution of AI and the diverse set of approaches that aim to achieve the current goals of AI.

Hierarchical features tame high-dimensional data

As discussed throughout *Part 2*, the key challenge of supervised learning is to generalize from training data to new samples. Generalization becomes exponentially more difficult as the dimensionality of the data increases. We encountered the root causes of these difficulties as the curse of dimensionality in *Chapter 13, Data-Driven Risk Factors and Asset Allocation with Unsupervised Learning*.

One aspect of this curse is that volume grows exponentially with the number of dimensions: for a hypercube with edge length 10, volume increases from 10^3 to 10^4 as its dimensionality increases from three to four. Conversely, the **data density for a given sample size drops exponentially**. In other words, the number of observations required to maintain a certain density grows exponentially.

Another aspect is that functional relationships between the features and the output can become more complex when they are allowed to

vary across a growing number of dimensions. As discussed in *Chapter 6, The Machine Learning Process*, ML algorithms struggle to learn **arbitrary functions in a high-dimensional space** because the number of candidates grows exponentially while the density of the data available to infer the relationship drops simultaneously. To mitigate this problem, algorithms hypothesize that the target function belongs to a certain class and impose constraints on the search for the optimal solution within that class for the problem at hand.

Furthermore, algorithms typically assume that the output at a new point should be similar to the output at nearby training points. This prior **assumption of smoothness** or local constancy posits that the learned function will not change much in a small region, as illustrated by the k-nearest neighbor algorithm (see *Chapter 6, The Machine Learning Process*). However, as data density drops exponentially with a growing number of dimensions, the distance between training samples naturally rises. The notion of nearby training examples thus becomes less meaningful as the potential complexity of the target function increases.

For traditional ML algorithms, the number of parameters and required training samples is generally proportional to the number of regions in the input space that the algorithm is able to distinguish. DL is designed to overcome the challenges of learning an exponential number of regions from a limited number of training points by assuming that a hierarchy of features generates the data.

DL as representation learning

Many AI tasks like image or speech recognition require knowledge about the world. One of the key challenges is to encode this knowledge so a computer can utilize it. For decades, the development of ML systems required considerable domain expertise to transform the raw data (such as image pixels) into an internal representation that a learning algorithm could use to detect or classify patterns.

Similarly, how much value an ML algorithm adds to a trading strategy depends greatly on our ability to engineer features that represent the predictive information in the data so that the algorithm can process it. Ideally, the features capture independent drivers of the outcome, as discussed in *Chapter 4, Financial Feature Engineering – How to Research Alpha Factors*, and throughout *Parts 2 and 3* when designing and evaluating factors that capture trading signals.

Rather than relying on hand-designed features, representation learning allows an ML algorithm to automatically discover the representation of the data most useful for detecting or classifying patterns. DL combines this technique with specific assumptions about the nature of the features. See Bengio, Courville, and Vincent (2013) for additional information.

How DL extracts hierarchical features from data

The core idea behind DL is that a multi-level hierarchy of features has generated the data. Consequently, a DL model encodes the prior belief that the target function is composed of a nested set of simpler functions. This assumption permits an exponential gain in the number of regions that can be distinguished for a given number of training samples.

In other words, DL is a representation learning method that extracts a hierarchy of concepts from the data. It learns this hierarchical representation by **composing simple but non-linear functions** that successively transform the representation of one level (starting with the input data) into a new representation at a higher, slightly more abstract level. By combining enough of these transformations, DL is able to learn very complex functions.

Applied to a **classification task**, for example, higher levels of representation tend to amplify the aspects of the data most helpful for discriminating objects while suppressing irrelevant sources of variation. As we will see in more detail in *Chapter 18, CNNs for Financial Time Series and Satellite Images*, raw image data is just a two- or three-dimensional array of pixel values. The first layer of representation typically learns features that focus on the presence or absence of edges at particular orientations and locations. The second layer often learns motifs that depend on particular edge arrangements, regardless of small variations in their positions. The following layer may assemble motifs to represent parts of relevant objects, and subsequent layers would detect objects as combinations of these parts.

The **key breakthrough of DL** is that a general-purpose learning algorithm can extract hierarchical features suitable for modeling high-dimensional, unstructured data in a way that is infinitely more scalable than human engineering. It is thus no surprise that the rise of DL parallels the large-scale availability of unstructured image or text data. To the extent that these data sources also figure prominently among alternative data, DL has become highly relevant for algorithmic trading.

Good and bad news – the universal approximation theorem

The **universal approximation theorem** formalizes the ability of NNs to capture arbitrary relationships between input and output data. George Cybenko (1989) demonstrated that single-layer NNs using sigmoid activation functions can represent any continuous function on a closed and bounded subset of \mathbb{R}^n . Kurt Hornik (1991) further showed that it is not the specific shape of the activation function but rather the **multilayered architecture** that enables the hierarchical feature representation, which in turn allows NNs to approximate universal functions.

However, the theorem does not help us identify the network architecture required to represent a specific target function. We will see in the last section of this chapter that there are numerous parameters to optimize, including the network's width and depth, the number of connections between neurons, and the type of activation functions.

Furthermore, the ability to represent arbitrary functions does not imply that a network can actually learn the parameters for a given function. It took over two decades for backpropagation, the most popular learning algorithm for NNs to become effective at scale. Unfortunately, given the highly nonlinear nature of the optimization problem, there is no guarantee that it will find the absolute best rather than just a relatively good solution.

How DL relates to ML and AI

AI has a long history, going back at least to the 1950s as an academic field and much longer as a subject of human inquiry, but has

experienced several waves of ebbing and flowing enthusiasm since (see Nilsson, 2009, for an in-depth survey). ML is an important subfield with a long history in related disciplines such as statistics and became prominent in the 1980s. As we have just discussed, and as depicted in *Figure 17.1*, DL is a form of representation learning and is itself a subfield of ML.

The initial goal of AI was to achieve **general AI**, conceived as the ability to solve problems considered to require human-level intelligence, and to reason and draw logical conclusions about the world and automatically improve itself. AI applications that do not involve ML include knowledge bases that encode information about the world, combined with languages for logical operations.

Historically, much AI effort went into developing **rule-based systems** that aimed to capture expert knowledge and decision-making rules, but hard-coding these rules frequently failed due to excessive complexity. In contrast, ML implies a **probabilistic approach** that learns rules from data and aims at circumventing the limitations of human-designed rule-based systems. It also involves a shift to narrower, task-specific objectives.

The following figure sketches the relationship between the various AI subfields, outlines their goals, and highlights their relevance on a timeline.

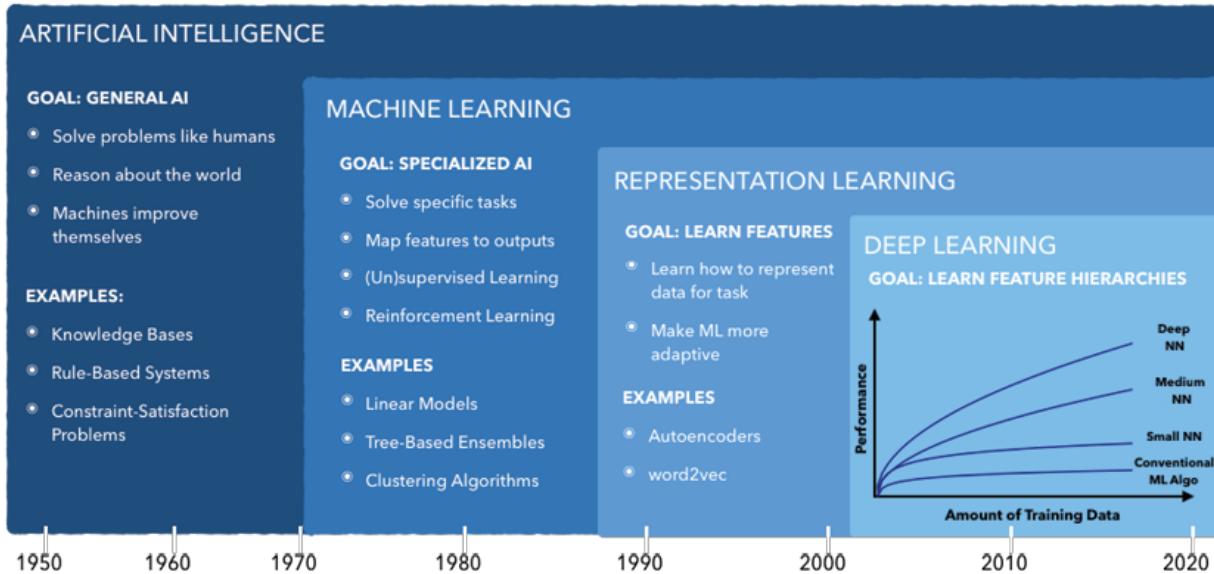


Figure 17.1: AI timeline and subfields

In the next section, we will see how to actually build a neural network.

Designing an NN

DL relies on **NNs**, which consist of a few key building blocks, which in turn can be configured in a multitude of ways. In this section, we introduce how NNs work and illustrate their most important components used to design different architectures.

(Artificial) NNs were originally inspired by biological models of learning like the human brain, either in an attempt to mimic how it works and achieve similar success, or to gain a better understanding through simulation. Current NN research draws less on neuroscience, not least since our understanding of the brain has not yet reached a sufficient level of granularity. Another constraint is overall size: even if the number of neurons used in NNs continued to

double every year since their inception in the 1950s, they would only reach the scale of the human brain around 2050.

We will also explain how **backpropagation**, often simply called **backprop**, uses gradient information (the value of the partial derivative of the cost function with respect to a parameter) to adjust all neural network parameters based on training errors. The composition of various nonlinear modules implies that the optimization of the objective function can be quite challenging. We also introduce refinements of backpropagation that aim to accelerate the learning process.

A simple feedforward neural network architecture

In this section, we introduce **feedforward NNs**, which are based on the **multilayer perceptron (MLP)** and consist of one or more hidden layers that connect the input to the output layer. In feedforward NNs, information only flows from input to output, such that they can be represented as directed acyclic graphs, as in the following figure. In contrast, **recurrent neural networks (RNNs)** (see *Chapter 19, RNNs for Multivariate Time Series and Sentiment Analysis*) include loops from the output back to the input to track or memorize past patterns and events.

We will first describe the feedforward NN architecture and how to implement it using NumPy. Then we will explain how backpropagation learns the NN weights and implement it in Python to train a binary classification network that produces perfect results even though the classes are not linearly separable. See the notebook `build_and_train_feedforward_nn` for implementation details.

A feedforward NN consists of several **layers**, each of which receives a sample of input data and produces an output. The **chain of transformations** starts with the input layer, which passes the source data to one of several internal or hidden layers, and ends with the output layer, which computes a result for comparison with the sample's output value.

The hidden and output layers consist of nodes or neurons. Nodes of a **fully connected** or dense layer connect to some or all nodes of the previous layer. The network architecture can be summarized by its depth, measured by the number of hidden layers, or the width and the number of nodes of each layer.

Each connection has a **weight** used to compute a linear combination of the input values. A layer may also have a **bias** node that always outputs a 1 and is used by the nodes in the subsequent layer, like a constant in linear regression. The goal of the training phase is to learn values for these weights that optimize the network's predictive performance.

Each node of the hidden layers computes the **dot product** of the weights and the output of the previous layer. An **activation function** transforms the result, which becomes the input to the subsequent layer. This transformation is typically nonlinear (like the sigmoid function used for logistic regression; see *Chapter 7, Linear Models – From Risk Factors to Return Forecasts*, on linear models) so that the network can learn nonlinear relationships; we'll discuss common activation functions in the next section. The output layer computes the linear combination of the output of the last hidden layer with its weights and uses an activation function that matches the type of ML problem.

The computation of the network output from the inputs thus flows through a chain of nested functions and is called **forward propagation**. Figure 17.2 illustrates a single-layer feedforward NN with a two-dimensional input vector, a hidden layer of width three, and two nodes in the output layer. This architecture is simple enough, so we can still easily graph it yet illustrate the key concepts.

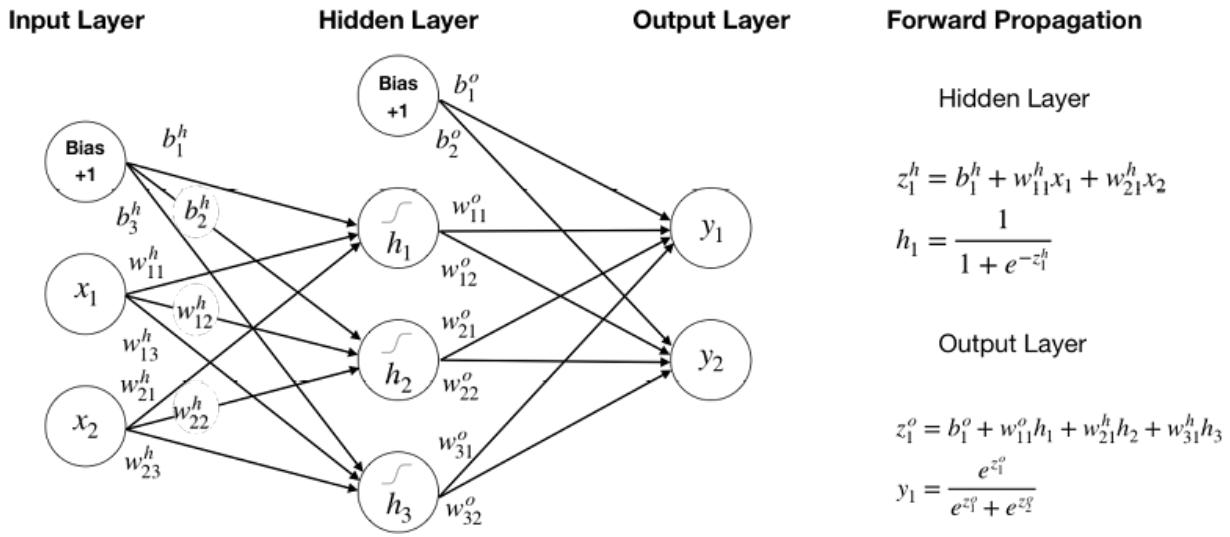


Figure 17.2: A feedforward architecture with one hidden layer

The **network graph** shows that each of the three hidden layer nodes (not counting the bias) has three weights, one for the input layer bias and two for each of the two input variables. Similarly, each output layer node has four weights to compute the product sum or dot product of the hidden layer bias and activations. In total, there are 17 parameters to be learned.

The **forward propagation** panel on the right of the figure lists the computations for an example node at the hidden and output layers, h and o , respectively. The first node in the hidden layer applies the sigmoid function to the linear combination z of its weights and inputs akin to logistic regression. The hidden layer thus runs three logistic regressions in parallel, while the backpropagation algorithm

ensures that their parameters will most likely differ to best inform subsequent layers.

The output layer uses a **softmax** activation function (see *Chapter 6, The Machine Learning Process*) that generalizes the logistic sigmoid function to multiple classes. It adjusts the dot product of the hidden layer output with its weight to represent probabilities for the classes (only two in this case to simplify the presentation).

The forward propagation can also be expressed as nested functions, where h again represents the hidden layer and o the output layer to produce the NN estimate of the output: $\hat{y} = o(h(x))$.

Key design choices

Some NN design choices resemble those for other supervised learning models. For example, the output is dictated by the type of the ML problem such as regression, classification, or ranking. Given the output, we need to select a cost function to measure prediction success and failure, and an algorithm that optimizes the network parameters to minimize the cost.

NN-specific choices include the numbers of layers and nodes per layer, the connections between nodes of different layers, and the type of activation functions.

A key concern is **training efficiency**: the functional form of activations can facilitate or hinder the flow of the gradient information available to the backpropagation algorithm that adjusts the weights in response to training errors. Functions with flat regions for large input value ranges have a very low gradient and

can impede training progress when parameter values get stuck in such a range.

Some architectures add **skip connections** that establish direct links beyond neighboring layers to facilitate the flow of gradient information. On the other hand, the deliberate omission of connections can reduce the number of parameters to limit the network's capacity and possibly lower the generalization error, while also cutting the computational cost.

Hidden units and activation functions

Several nonlinear activation functions besides the sigmoid function have been used successfully. Their design remains an area of research because they are the key element that allows the NN to learn nonlinear relationships. They also have a critical impact on the training process because their derivatives determine how errors translate into weight adjustments.

A very popular activation function is the **rectified linear unit (ReLU)**. The activation is computed as $g(z) = \max(0, z)$ for a given activation z , resulting in a functional form similar to the payoff for a call option. The derivative is constant whenever the unit is active. ReLUs are usually combined with an affine input transformation that requires the presence of a bias node. Their discovery has greatly improved the performance of feedforward networks compared to sigmoid units, and they are often recommended as the default. There are several ReLU extensions that aim to address the limitations of ReLU to learn via gradient descent when they are not active and their gradient is zero (Goodfellow, Bengio, and Courville, 2016).

Another alternative to the logistic function σ is the **hyperbolic tangent function tanh**, which produces output values in the ranges $[-1, 1]$. They are closely related because $\tanh(z) = 2\sigma(2z) - 1$. Both functions suffer from saturation because their gradient becomes very small for very low and high input values. However, tanh often performs better because it more closely resembles the identity function so that for small activation values, the network behaves more like a linear model, which in turn facilitates training.

Output units and cost functions

The choice of NN output format and cost function depends on the type of supervised learning problem:

- **Regression problems** use a linear output unit that computes the dot product of its weights with the final hidden layer activations, typically in conjunction with mean squared error cost
- **Binary classification** uses sigmoid output units to model a Bernoulli distribution just like logistic regression with hidden activations as input
- **Multiclass problems** rely on softmax units that generalize the logistic sigmoid and model a discrete distribution over more than two classes, as demonstrated earlier

Binary and multiclass problems typically use cross-entropy loss, which significantly improves training efficacy compared to mean squared error (see *Chapter 6, The Machine Learning Process*, for additional information on loss functions).

How to regularize deep NNs

The downside of the capacity of NNs to approximate arbitrary functions is the greatly increased risk of overfitting. The best **protection against overfitting** is to train the model on a larger dataset. Data augmentation, such as creating slightly modified versions of images, is a powerful alternative approach. The generation of synthetic financial training data for this purpose is an active research area that we will address in *Chapter 20, Autoencoders for Conditional Risk Factors and Asset Pricing* (see, for example, Fu et al. 2019).

As an alternative or complement to obtaining more data, regularization can help mitigate the risk of overfitting. For all models discussed so far in this book, there is some form of regularization that modifies the learning algorithm to reduce its generalization error without negatively affecting its training error. Examples include the penalties added to the ridge and lasso regression objectives and the split or depth constraints used with decision trees and tree-based ensemble models.

Frequently, regularization takes the form of a soft constraint on the parameter values that trades off some additional bias for lower variance. A common practical finding is that the model with the lowest generalization error is not the model with the exact right size of parameters, but rather a larger model that has been well regularized. Popular NN regularization techniques that can be used in combination include parameter norm penalties, early stopping, and dropout.

Parameter norm penalties

We encountered **parameter norm penalties** for lasso and ridge regression as **L1 and L2 regularization**, respectively, in *Chapter 7*,

Linear Models – From Risk Factors to Return Forecasts. In the NN context, parameter norm penalties similarly modify the objective function by adding a term that represents the L1 or L2 norm of the parameters, weighted by a hyperparameter that requires tuning. For NN, the bias parameters are usually not constrained, only the weights.

L1 regularization can produce sparse parameter estimates by reducing weights all the way to zero. L2 regularization, in contrast, preserves directions along which the parameters significantly reduce the cost function. Penalties or hyperparameter values can vary across layers, but the added tuning complexity quickly becomes prohibitive.

Early stopping

We encountered **early stopping** as a regularization technique in *Chapter 12, Boosting Your Trading Strategy*. It is perhaps the most common NN regularization method because it is both effective and simple to use: it monitors the model's performance on a validation set and stops training when the performance ceases to improve for a certain number of observations to prevent overfitting.

Early stopping can be viewed as **efficient hyperparameter selection** that automatically determines the correct amount of regularization, whereas parameter penalties require hyperparameter tuning to identify the ideal weight decay. Just be careful to avoid **lookahead bias**: backtest results will be exceedingly positive when early stopping uses out-of-sample data that would not be available during a real-life implementation of the strategy.

Dropout

Dropout refers to the randomized omission of individual units with a given probability during forward or backward propagation. As a result, these omitted units do not contribute to the training error or receive updates.

The technique is computationally inexpensive and does not constrain the choice of model or training procedure. While more iterations are necessary to achieve the same amount of learning, each iteration is faster due to the lower computational cost. Dropout reduces the risk of overfitting by preventing units from compensating for mistakes made by other units during the training process.

Training faster – optimizations for deep learning

Backprop refers to the computation of the gradient of the cost function with respect to the internal parameter we wish to update and the use of this information to update the parameter values. The gradient is useful because it indicates the direction of parameter change that causes the maximal increase in the cost function. Hence, adjusting the parameters according to the negative gradient produces an optimal cost reduction, at least for a region very close to the observed samples. See Ruder (2017) for an excellent overview of key gradient descent optimization algorithms.

Training deep NNs can be time-consuming due to the nonconvex objective function and the potentially large number of parameters. Several challenges can significantly delay convergence, find a poor optimum, or cause oscillations or divergence from the target:

- **Local minima** can prevent convergence to a global optimum and cause poor performance
- **Flat regions with low gradients** that are not a local minimum can also prevent convergence while most likely being distant from the global optimum
- **Steep regions with high gradients** resulting from multiplying several large weights can cause excessive adjustments
- Deep architectures or long-term dependencies in an RNN require the multiplication of many weights during backpropagation, leading to **vanishing gradients** so that at least parts of the NN receive few or no updates

Several algorithms have been developed to address some of these challenges, namely variations of stochastic gradient descent and approaches that use adaptive learning rates. There is no single best algorithm, although adaptive learning rates have shown some promise.

Stochastic gradient descent

Gradient descent iteratively adjusts these parameters using the gradient information. For a given parameter θ , the basic gradient descent rule adjusts the value by the negative gradient of the loss function with respect to this parameter, multiplied by a learning rate η :

$$\theta = \theta - \underbrace{\eta}_{\text{Learning Rate}} \cdot \underbrace{\nabla_{\theta} J(\theta)}_{\text{Gradient}}$$

The gradient can be evaluated for all training data, a randomized batch of data, or individual observations (called online learning). Random samples give rise to **stochastic gradient descent (SGD)**,

which often leads to faster convergence if random samples are an unbiased estimate of the gradient direction throughout the training process.

However, there are numerous challenges: it can be difficult to define a learning rate or a rate schedule that facilitates efficient convergence *ex ante*—too low a rate prolongs the process, and too high a rate can lead to repeated overshooting and oscillation around or even divergence from a minimum. Furthermore, the same learning rate may not be adequate for all parameters, that is, in all directions of change.

Momentum

A popular refinement of basic gradient descent adds momentum to **accelerate the convergence to a local minimum**. Illustrations of momentum often use the example of a local optimum at the center of an elongated ravine (while in practice the dimensionality would be much higher than three). It implies a minimum inside a deep and narrow canyon with very steep walls that have a large gradient on one side and a much gentler slope towards a local minimum at the bottom of this region on the other side. Gradient descent naturally follows the steep gradient and will make repeated adjustments up and down the walls of the canyons with much slower movements towards the minimum.

Momentum aims to address such a situation by **tracking recent directions** and adjusting the parameters by a weighted average of the most recent gradient and the currently computed value. It uses a momentum term γ to weigh the contribution of the latest adjustment to this iteration's update v_t :

$$v_t = \gamma v_{t-1} + \eta \nabla_{\theta} J(\theta)$$

Nesterov momentum is a simple change to normal momentum. Here, the gradient term is not computed at the current parameter space position θ_t but instead from an intermediate position. The goal is to correct for the momentum term overshooting or pointing in the wrong direction (Sutskever et al. 2013).

Adaptive learning rates

The choice of the appropriate learning rate is very challenging as highlighted in the previous subsection on stochastic gradient descent. At the same time, it is one of the most important parameters that strongly impacts training time and generalization performance.

While momentum addresses some of the issues with learning rates, it does so at the expense of introducing another hyperparameter, the **momentum rate**. Several algorithms aim to adapt the learning rate throughout the training process based on gradient information.

AdaGrad

AdaGrad accumulates all historical, parameter-specific gradient information and continues to rescale the learning rate inversely proportional to the squared cumulative gradient for a given parameter. The goal is to slow down changes for parameters that have already changed a lot and to encourage adjustments for those that haven't.

AdaGrad is designed to perform well on convex functions and has had a mixed performance in a DL context because it can reduce the learning rate too quickly based on early gradient information.

RMSProp

RMSProp modifies AdaGrad to use an exponentially weighted average of the cumulative gradient information. The goal is to put more emphasis on recent gradients. It also introduces a new hyperparameter that controls the length of the moving average.

RMSProp is a popular algorithm that often performs well, provided by the various libraries that we will introduce later and routinely used in practice.

Adam

Adam stands for **adaptive moment derivation** and combines aspects of RMSProp with Momentum. It is considered fairly robust and often used as the default optimization algorithm (Kingma and Ba, 2014).

Adam has several hyperparameters with recommended default values that may benefit from some tuning:

- **alpha**: The learning rate or step size determines how much weights are updated so that larger (smaller) values speed up (slow down) learning before the rate is updated; many libraries use the 0.001 default
- **beta₁**: The exponential decay rate for the first moment estimates; typically set to 0.9
- **beta₂**: The exponential decay rate for the second-moment estimates; usually set to 0.999
- **epsilon**: A very small number to prevent division by zero; often set to 1e-8

Summary – how to tune key hyperparameters

Hyperparameter optimization aims at **tuning the capacity of the model** so that it matches the complexity of the relationship between the input of the data. Excess capacity makes overfitting likely and requires either more data that introduces additional information into the learning process, reducing the size of the model, or more aggressive use of the various regularization tools just described.

The **principal diagnostic tool** is the behavior of training and validation error described in *Chapter 6, The Machine Learning Process*: if the validation error worsens while the training error continues to drop, the model is overfitting because its capacity is too high. On the other hand, if performance falls short of expectations, increasing the size of the model may be called for.

The most important aspect of parameter optimization is the architecture itself as it largely determines the number of parameters: other things being equal, more or wider hidden layers increase the capacity. As mentioned before, the best performance is often associated with models that have excess capacity but are well regularized using mechanisms like dropout or L1/L2 penalties.

In addition to **balancing model size and regularization**, it is important to tune the **learning rate** because it can undermine the optimization process and reduce the effective model capacity. The adaptive optimization algorithms offer a good starting point as described for Adam, the most popular option.

A neural network from scratch in Python

To gain a better understanding of how NNs work, we will formulate the single-layer architecture and forward propagation computations displayed in *Figure 17.2* using matrix algebra and implement it using NumPy. You can find the code samples in the notebook

`build_and_train_feedforward_nn`.

The input layer

The architecture shown in *Figure 17.2* is designed for two-dimensional input data X that represents two different classes Y . In matrix form, both X and Y are of shape $N \times 2$:

$$X = \begin{bmatrix} x_{11} & x_{12} \\ \vdots & \vdots \\ x_{N1} & x_{N2} \end{bmatrix} \quad Y = \begin{bmatrix} y_{11} & y_{12} \\ \vdots & \vdots \\ y_{N1} & y_{N2} \end{bmatrix}$$

We will generate 50,000 random binary samples in the form of two concentric circles with different radius using scikit-learn's `make_circles` function so that the classes are not linearly separable:

```
N = 50000
factor = 0.1
noise = 0.1
X, y = make_circles(n_samples=N, shuffle=True,
                     factor=factor, noise=noise)
```

We then convert the one-dimensional output into a two-dimensional array:

```

Y = np.zeros((N, 2))
for c in [0, 1]:
    Y[y == c, c] = 1
'Shape of: X: (50000, 2) | Y: (50000, 2) | y: (50000,)'

```

Figure 17.3 shows a scatterplot of the data that is clearly not linearly separable:

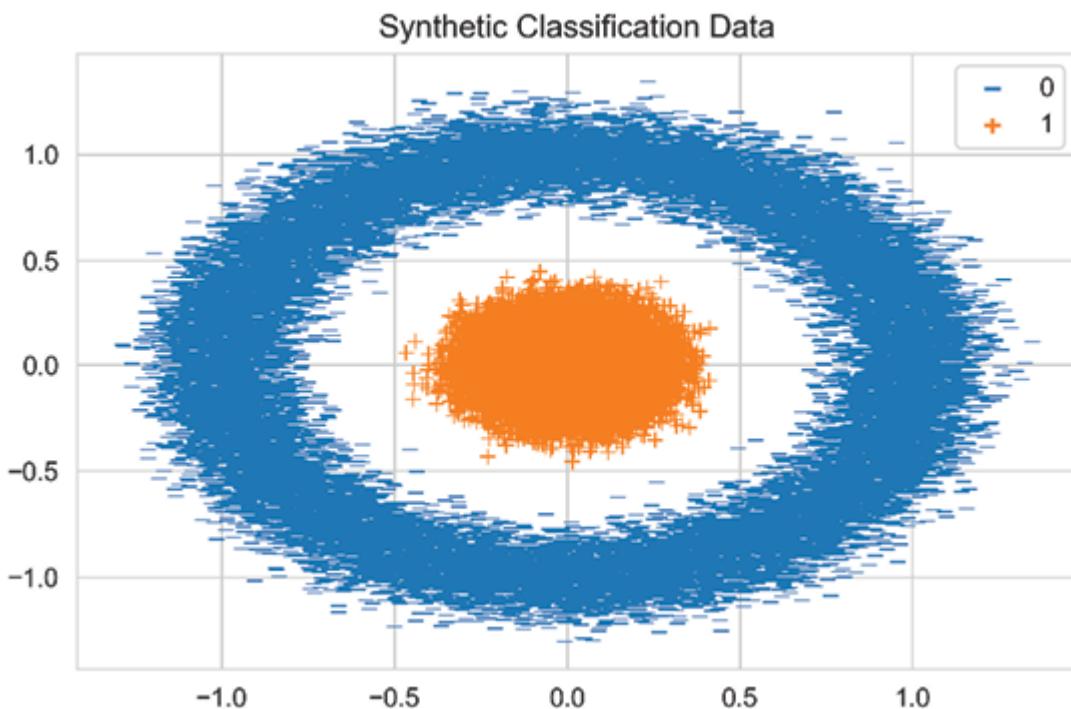


Figure 17.3: Synthetic data for binary classification

The hidden layer

The hidden layer h projects the two-dimensional input into a three-dimensional space using the weights W^h and translates the result by the bias vector b^h . To perform this affine transformation, the hidden layer weights are represented by a 2×3 matrix W^h , and the hidden layer bias vector by a three-dimensional vector:

$$W^h_{2 \times 3} = \begin{bmatrix} w^h_{11} & w^h_{12} & w^h_{13} \\ w^h_{21} & w^h_{22} & w^h_{23} \end{bmatrix} \quad \mathbf{b}^h_{1 \times 3} [b^h_1 \quad b^h_2 \quad b^h_3]$$

The hidden layer activations H result from the application of the sigmoid function to the dot product of the input data and the weights after adding the bias vector:

$$H_{N \times 3} = \sigma(X \cdot W^h + b^h) = \frac{1}{1 + e^{-(X \cdot W^h + b^h)}} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ \vdots & \vdots & \vdots \\ h_{N1} & h_{N2} & h_{N3} \end{bmatrix}$$

To implement the hidden layer using NumPy, we first define the `logistic` sigmoid function:

```
def logistic(z):
    """Logistic function."""
    return 1 / (1 + np.exp(-z))
```

We then define a function that computes the hidden layer activations as a function of the relevant inputs, weights, and bias values:

```
def hidden_layer(input_data, weights, bias):
    """Compute hidden activations"""
    return logistic(input_data @ weights + bias)
```

The output layer

The output layer compresses the three-dimensional hidden layer activations H back to two dimensions using a 3×2 weight matrix W^o and a two-dimensional bias vector \mathbf{b}^o :

$$3 \times 2 = \begin{bmatrix} w^o_{11} & w^o_{12} \\ w^o_{21} & w^o_{22} \\ w^o_{31} & w^o_{32} \end{bmatrix} \quad 1 \times 2 = [b^o_1 \quad b^o_2]$$

The linear combination of the hidden layer outputs results in an $N \times 2$ matrix \mathbf{Z}^o :

$$N \times 2 = N \times 3 \cdot 3 \times 2 + 1 \times 2$$

The output layer activations are computed by the softmax function ς that normalizes the \mathbf{Z}^o to conform to the conventions used for discrete probability distributions:

$$N \times 2 = \varsigma(\mathbf{H} \cdot \mathbf{W}^o + \mathbf{b}^o) = \begin{bmatrix} y_{11} & y_{12} \\ \vdots & \vdots \\ y_{n1} & y_{n2} \end{bmatrix}$$

We create a softmax function in Python as follows:

```
def softmax(z):
    """Softmax function"""
    return np.exp(z) / np.sum(np.exp(z), axis=1, keepdims=True)
```

As defined here, the output layer activations depend on the hidden layer activations and the output layer weights and biases:

```
def output_layer(hidden_activations, weights, bias):
    """Compute the output y_hat"""
    return softmax(hidden_activations @ weights + bias)
```

Now we have all the components we need to integrate the layers and compute the NN output directly from the input.

Forward propagation

The `forward_prop` function combines the previous operations to yield the output activations from the input data as a function of weights and biases:

```
def forward_prop(data, hidden_weights, hidden_bias, output_weights, ou
    """Neural network as function."""
    hidden_activations = hidden_layer(data, hidden_weights, hidden_bias)
    return output_layer(hidden_activations, output_weights, output_bias)
```

The `predict` function produces the binary class predictions given weights, biases, and input data:

```
def predict(data, hidden_weights, hidden_bias, output_weights, output_
    """Predicts class 0 or 1"""
    y_pred_proba = forward_prop(data,
                                hidden_weights,
                                hidden_bias,
                                output_weights,
                                output_bias)
    return np.around(y_pred_proba)
```

The cross-entropy cost function

The final piece is the cost function to evaluate the NN output based on the given label. The cost function J uses the cross-entropy loss ξ , which sums the deviations of the predictions for each class c from the actual outcome:

$$J(\mathbf{Y}, \hat{\mathbf{Y}}) = \sum_{i=1}^n \xi(y_i, \hat{y}_i) = - \sum_{i=1}^N \sum_{c=1}^C y_{ic} \cdot \log(\hat{y}_{ic})$$

It takes the following form in Python:

```
def loss(y_hat, y_true):  
    """Cross-entropy"""  
    return - (y_true * np.log(y_hat)).sum()
```

How to implement backprop using Python

To update the NN weights and bias values using backprop, we need to compute the gradient of the cost function. The gradient represents the partial derivative of the cost function with respect to the target parameter.

How to compute the gradient

The NN composes a set of nested functions as highlighted earlier. Hence, the gradient of the loss function with respect to internal, hidden parameters is computed using the chain rule of calculus.

For scalar values, given the functions $z = h(x)$ and $y = o(h(x)) = o(z)$, we compute the derivative of y with respect to x using the chain rule as follows:

$$\frac{dy}{dx} = \frac{dy}{dz} \frac{dz}{dx}$$

For vectors, with $z \in \mathbb{R}^m$ and $x \in \mathbb{R}^n$ so that the hidden layer h maps from \mathbb{R}^n to \mathbb{R}^m and $z = h(x)$ and $y = o(z)$, we get:

$$\frac{\partial y}{\partial x_i} = \sum_j \frac{\partial y}{\partial z_j} \frac{\partial z_j}{\partial x_i}$$

We can express this more concisely using matrix notation using the $m \times n$ Jacobian matrix of h :

$$m \times n \frac{dz}{dx}$$

which contains the partial derivatives for each of the m components of z with respect to each of the n inputs x . The gradient ∇ of y with respect to x contains all partial derivatives and can thus be written as:

$$\nabla_x y = \left(\frac{dz}{dx} \right)^T \nabla_z y$$

The loss function gradient

The derivative of the cross-entropy loss function J with respect to each output layer activation $i = 1, \dots, N$ is a very simple expression (see the notebook for details), shown below on the left for scalar values and on the right in matrix notation:

$$\frac{\partial J}{\partial z_i} = \hat{y}_i - y_i \quad \nabla_{z^0} J = \hat{\mathbf{Y}} - \mathbf{Y} = \delta^0$$

We define `loss_gradient` function accordingly:

```
def loss_gradient(y_hat, y_true):  
    """output layer gradient"""  
    return y_hat - y_true
```

The output layer gradients

To propagate the update back to the output layer weights, we use the gradient of the loss function J with respect to the weight matrix:

$$\frac{\partial J}{\partial \mathbf{W}^0} = H^T \cdot (\widehat{\mathbf{Y}} - \mathbf{Y}) = H^T \cdot \delta^0$$

and for the bias:

$$\frac{\partial J}{\partial \mathbf{b}^0} = \frac{\partial J}{\partial Y} \frac{\partial Y}{\partial Z^0} \frac{\partial Z^0}{\partial \mathbf{b}^0} = \sum_{i=1}^N 1 \cdot (\widehat{\mathbf{y}}_i - \mathbf{y}_i) = \sum_{i=1}^N \delta_i^0$$

We can now define `output_weight_gradient` and `output_bias_gradient` accordingly, both taking the loss gradient δ^0 as input:

```
def output_weight_gradient(H, loss_grad):
    """Gradients for the output layer weights"""
    return H.T @ loss_grad
def output_bias_gradient(loss_grad):
    """Gradients for the output layer bias"""
    return np.sum(loss_grad, axis=0, keepdims=True)
```

The hidden layer gradients

The gradient of the loss function with respect to the hidden layer values computes as follows, where \circ refers to the element-wise matrix product:

$$\nabla_{\mathbf{Z}^h} J = \mathbf{H} \circ (1 - \mathbf{H}) \circ [\delta_0 \cdot (\mathbf{W}^0)^T] = \delta_h$$

We define a `hidden_layer_gradient` function to encode this result:

```
def hidden_layer_gradient(H, out_weights, loss_grad):
    """Error at the hidden layer.
    H * (1-H) * (E . Wo^T)"""
    return H * (1 - H) * (loss_grad @ out_weights.T)
```

The gradients for hidden layer weights and biases are:

$$\nabla_{\mathbf{W}^h} J = \mathbf{X}^T \cdot \delta^h \quad \nabla_{\mathbf{b}^h} J = \sum_{j=1}^N \delta_{hj}$$

The corresponding functions are:

```
def hidden_weight_gradient(X, hidden_layer_grad):
    """Gradient for the weight parameters at the hidden layer"""
    return X.T @ hidden_layer_grad

def hidden_bias_gradient(hidden_layer_grad):
    """Gradient for the bias parameters at the output layer"""
    return np.sum(hidden_layer_grad, axis=0, keepdims=True)
```

Putting it all together

To prepare for the training of our network, we create a function that combines the previous gradient definition and computes the relevant weight and bias updates from the training data and labels, and the current weight and bias values:

```
def compute_gradients(X, y_true, w_h, b_h, w_o, b_o):
    """Evaluate gradients for parameter updates"""
    # Compute hidden and output layer activations
    hidden_activations = hidden_layer(X, w_h, b_h)
    y_hat = output_layer(hidden_activations, w_o, b_o)
    # Compute the output layer gradients
    loss_grad = loss_gradient(y_hat, y_true)
    out_weight_grad = output_weight_gradient(hidden_activations, loss_grad)
    out_bias_grad = output_bias_gradient(loss_grad)
    # Compute the hidden layer gradients
    hidden_layer_grad = hidden_layer_gradient(hidden_activations,
                                              w_o, loss_grad)
    hidden_weight_grad = hidden_weight_gradient(X, hidden_layer_grad)
    hidden_bias_grad = hidden_bias_gradient(hidden_layer_grad)
    return [hidden_weight_grad, hidden_bias_grad, out_weight_grad, out_bias_grad]
```

Testing the gradients

The notebook contains a test function that compares the gradient derived previously analytically using multivariate calculus to a numerical estimate that we obtain by slightly perturbing individual parameters. The test function validates that the resulting change in output value is similar to the change estimated by the analytical gradient.

Implementing momentum updates using Python

To incorporate momentum into the parameter updates, define an `update_momentum` function that combines the results of the `compute_gradients` function we just used with the most recent momentum updates for each parameter matrix:

```
def update_momentum(X, y_true, param_list, Ms, momentum_term, learning_rate):
    """Compute updates with momentum."""
    gradients = compute_gradients(X, y_true, *param_list)
    return [momentum_term * momentum - learning_rate * grads
            for momentum, grads in zip(Ms, gradients)]
```

The `update_params` function performs the actual updates:

```
def update_params(param_list, Ms):
    """Update the parameters."""
    return [P + M for P, M in zip(param_list, Ms)]
```

Training the network

To train the network, we first randomly initialize all network parameters using a standard normal distribution (see the notebook). For a given number of iterations or epochs, we run momentum updates and compute the training loss as follows:

```

def train_network(iterations=1000, lr=.01, mf=.1):
    # Initialize weights and biases
    param_list = list(initialize_weights())
    # Momentum Matrices = [MWh, Mbh, MWo, Mbo]
    Ms = [np.zeros_like(M) for M in param_list]
    train_loss = [loss(forward_prop(X, *param_list), Y)]
    for i in range(iterations):
        # Update the moments and the parameters
        Ms = update_momentum(X, Y, param_list, Ms, mf, lr)
        param_list = update_params(param_list, Ms)
        train_loss.append(loss(forward_prop(X, *param_list), Y))
    return param_list, train_loss

```

Figure 17.4 plots the training loss over 50,000 iterations for 50,000 training samples with a momentum term of 0.5 and a learning rate of 1e-4. It shows that it takes over 5,000 iterations for the loss to start to decline but then does so very fast. We have not used SGD, which would have likely accelerated convergence significantly.

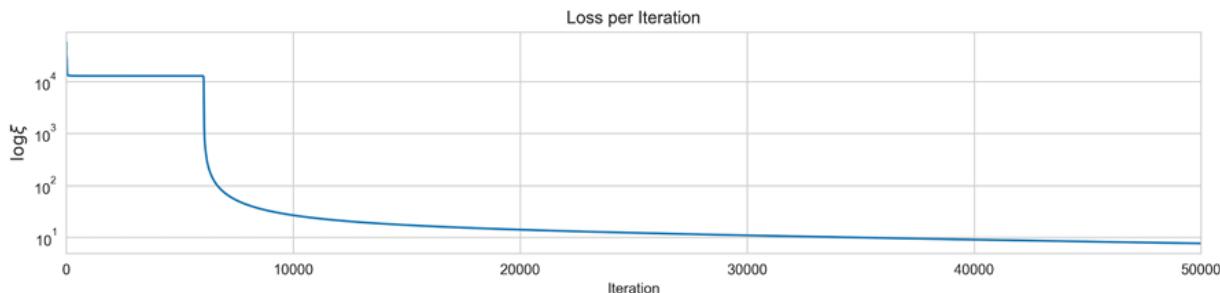


Figure 17.4: Training loss per iteration

The plots in *Figure 17.5* show the function learned by the neural network with a three-dimensional hidden layer from two-dimensional data with two classes that are not linearly separable. The left panel displays the source data and the decision boundary that misclassifies very few data points and would further improve with continued training.

The center panel shows the representation of the input data learned by the hidden layer. The network learns weights so that the projection of the input from two to three dimensions enables the linear separation of the two classes. The right plot shows how the output layer implements the linear separation in the form of a cutoff value of 0.5 in the output dimension:

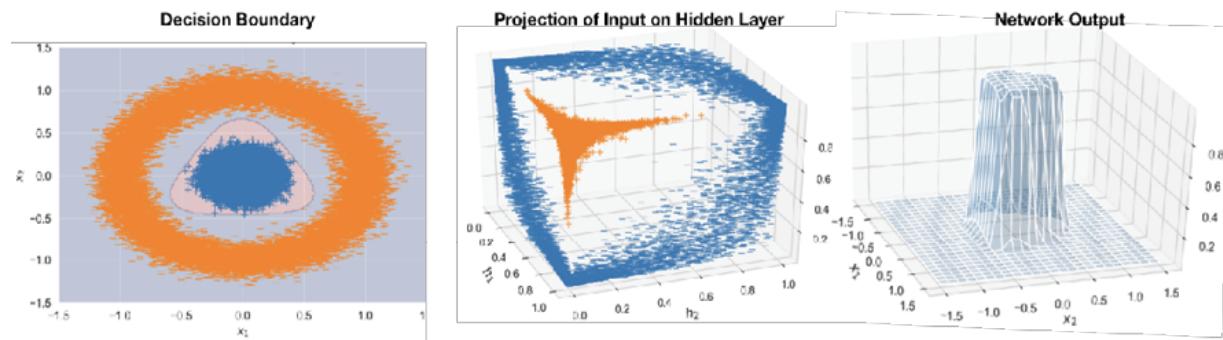


Figure 17.5: Visualizing the function learned by the neural network

To **sum up**: we have seen how a very simple network with a single hidden layer with three nodes and a total of 17 parameters is able to learn how to solve a nonlinear classification problem using backprop and gradient descent with momentum.

We will next review how to use popular DL libraries that facilitate the design and fast training of complex architectures while using sophisticated techniques to prevent overfitting and evaluate the results.

Popular deep learning libraries

Currently, the most popular DL libraries are TensorFlow (supported by Google), Keras (led by Francois Chollet, now at Google), and

PyTorch (supported by Facebook). Development is very active with PyTorch at version 1.4 and TensorFlow at 2.2 as of March 2020. TensorFlow 2.0 adopted Keras as its main interface, effectively combining both libraries into one.

All libraries provide the design choices, regularization methods, and backprop optimizations we discussed previously in this chapter. They also facilitate fast training on one or several **graphics processing units (GPUs)**. The libraries differ slightly in their focus with TensorFlow originally designed for deployment in production and prevalent in the industry, while PyTorch has been popular among academic researchers; however, the interfaces are gradually converging.

We will illustrate the use of TensorFlow and PyTorch using the same network architecture and dataset as in the previous section.

Leveraging GPU acceleration

DL is very computationally intensive, and good results often require large datasets. As a result, model training and evaluation can become rather time-consuming. GPUs are highly optimized for the matrix operations required by deep learning models and tend to have more processing power, rendering speedups of 10x or more not uncommon.

All popular deep learning libraries support the use of a GPU, and some also allow for parallel training on multiple GPUs. The most common types of GPU are produced by NVIDIA, and configuration requires installation and setup of the CUDA environment. The process continues to evolve and can be somewhat challenging depending on your computational environment.

A more straightforward way to leverage GPU is via the Docker virtualization platform. There are numerous images available that you can run in a local container managed by Docker that circumvents many of the driver and version conflicts that you may otherwise encounter. TensorFlow provides Docker images on its website that can also be used with Keras.

See GitHub for references and related instructions in the DL notebooks and the installation directory.

How to use TensorFlow 2

TensorFlow became the leading deep learning library shortly after its release in September 2015, one year before PyTorch. TensorFlow 2 simplified the API that had grown increasingly complex over time by making the Keras API its principal interface.

Keras was designed as a high-level API to accelerate the iterative workflow of designing and training deep neural networks with computational backends like TensorFlow, Theano, or CNTK. It has been integrated into TensorFlow in 2017. You can also combine code from both libraries to leverage Keras' high-level abstractions as well as customized TensorFlow graph operations.

In addition, TensorFlow adopts **eager execution**. Previously, you needed to define a complete computational graph for compilation into optimized operations. Running the compiled graph required the configuration of a session and the provision of the requisite data. Under eager execution, you can run TensorFlow operations on a line-by-line basis just like common Python code.

Keras supports both a slightly simpler Sequential API and a more flexible Functional API. We will introduce the former at this point and use the Functional API in more complex examples in the following chapters.

To create a model, we just need to instantiate a `Sequential` object and provide a list with the sequence of standard layers and their configurations, including the number of units, type of activation function, or name.

The first hidden layer needs information about the number of features in the matrix it receives from the input layer via the `input_shape` argument. In our simple case, there are just two. Keras infers the number of rows it needs to process during training, through the `batch_size` argument that we will pass to the `fit` method later in this section. TensorFlow infers the sizes of the inputs received by other layers from the previous layer's `units` argument:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
model = Sequential([
    Dense(units=3, input_shape=(2,), name='hidden'),
    Activation('sigmoid', name='logistic'),
    Dense(2, name='output'),
    Activation('softmax', name='softmax'),
])
```

The Keras API provides numerous standard building blocks, including recurrent and convolutional layers, various options for regularization, a range of loss functions and optimizers, and also preprocessing, visualization, and logging (see the link to the TensorFlow documentation on GitHub for reference). It is also extensible.

The model's `summary` method produces a concise description of the network architecture, including a list of the layer types and shapes and the number of parameters:

```
model.summary()
Layer (type)          Output Shape         Param #
=====
hidden (Dense)        (None, 3)            9
logistic (Activation) (None, 3)            0
output (Dense)         (None, 2)            8
softmax (Activation)  (None, 2)            0
=====
Total params: 17
Trainable params: 17
Non-trainable params: 0
```

Next, we compile the Sequential model to configure the learning process. To this end, we define the optimizer, the loss function, and one or several performance metrics to monitor during training:

```
model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy'])
```

Keras uses callbacks to enable certain functionality during training, such as logging information for interactive display in TensorBoard (see the next section):

```
tb_callback = TensorBoard(log_dir='./tensorboard',
                          histogram_freq=1,
                          write_graph=True,
                          write_images=True)
```

To train the model, we call its `fit` method and pass several parameters in addition to the training data:

```
model.fit(X, Y,
          epochs=25,
          validation_split=.2,
          batch_size=128,
          verbose=1,
          callbacks=[tb_callback])
```

See the notebook for a visualization of the decision boundary that resembles the result from our earlier manual network implementation. The training with TensorFlow runs orders of magnitude faster, though.

How to use TensorBoard

TensorBoard is a great suite of visualization tools that comes with TensorFlow. It includes visualization tools to simplify the understanding, debugging, and optimization of NNs.

You can use it to visualize the computational graph, plot various execution and performance metrics, and even visualize image data processed by the network. It also permits comparisons of different training runs.

When you run the `how_to_use_tensorflow` notebook, with TensorFlow installed, you can launch TensorBoard from the command line:

```
tensorboard --logdir=/full_path_to_your_logs ## e.g. ./tensorboard
```

Alternatively, you can use it within your notebook by first loading the extension and then starting TensorBoard similarly by referencing

the `log` directory:

```
%load_ext tensorboard  
%tensorboard --logdir tensorboard/
```

For starters, the visualizations include train and validation metrics (see the left panel of *Figure 17.6*).

In addition, you can view histograms of the weights and biases over various epochs (right panel of Figure 17.6; epochs evolve from back to front). This is useful because it allows you to monitor whether backpropagation succeeds in adjusting the weights as learning progresses and whether they are converging.

The values of weights should change from their initialization values over the course of several epochs and eventually stabilize:

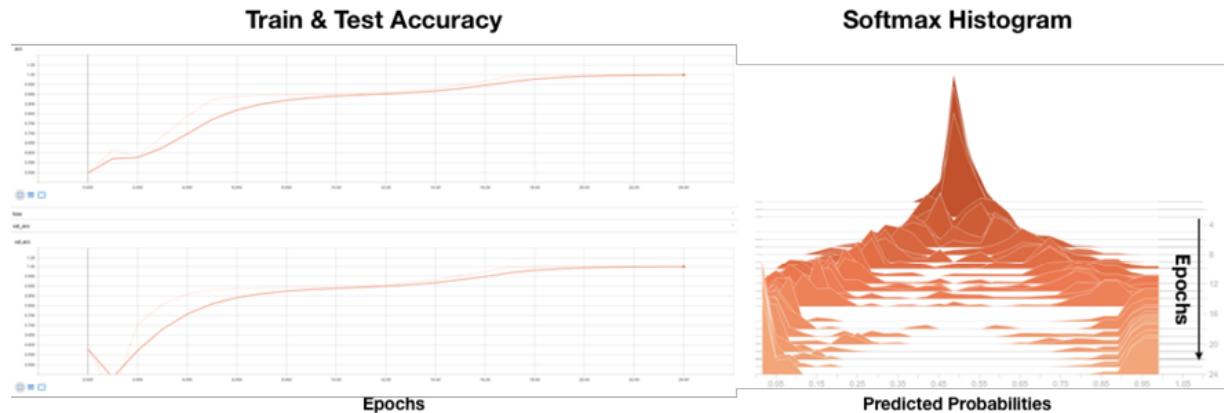


Figure 17.6: TensorBoard learning process visualization

TensorBoard also lets you display and interactively explore the computational graph of your network, drilling down from the high-level structure to the underlying operations by clicking on the various nodes. The visualization for our simple example architecture (see the notebook) already includes numerous components but is

very useful when debugging. For further reference, see the links on GitHub to more detailed tutorials.

How to use PyTorch 1.4

PyTorch was developed at the **Facebook AI Research (FAIR)** group led by Yann LeCunn, and the first alpha version released in September 2016. It provides deep integration with Python libraries like NumPy that can be used to extend its functionality, strong GPU acceleration, and automatic differentiation using its autograd system. It provides more granular control than Keras through a lower-level API and is mainly used as a deep learning research platform but can also replace NumPy while enabling GPU computation.

It employs eager execution, in contrast to the static computation graphs used by, for example, Theano or TensorFlow. Rather than initially defining and compiling a network for fast but static execution, it relies on its autograd package for automatic differentiation of tensor operations; that is, it computes gradients "on the fly" so that network structures can be partially modified more easily. This is called **define-by-run**, meaning that backpropagation is defined by how your code runs, which in turn implies that every single iteration can be different. The PyTorch documentation provides a detailed tutorial on this.

The resulting flexibility combined with an intuitive Python-first interface and speed of execution has contributed to its rapid rise in popularity and led to the development of numerous supporting libraries that extend its functionality.

Let's see how PyTorch and autograd work by implementing our simple network architecture (see the `how_to_use_pytorch` notebook for details).

How to create a PyTorch DataLoader

We begin by converting the NumPy or pandas input data to `torch` tensors. Conversion from and to NumPy is very straightforward:

```
import torch
X_tensor = torch.from_numpy(X)
y_tensor = torch.from_numpy(y)
X_tensor.shape, y_tensor.shape
(torch.Size([50000, 2]), torch.Size([50000]))
```

We can use these PyTorch tensors to instantiate first a `TensorDataset` and, in a second step, a `DataLoader` that includes information about `batch_size`:

```
import torch.utils.data as utils
dataset = utils.TensorDataset(X_tensor,y_tensor)
dataloader = utils.DataLoader(dataset,
                             batch_size=batch_size,
                             shuffle=True)
```

How to define the neural network architecture

PyTorch defines an NN architecture using the `Net()` class. The central element is the `forward` function. autograd automatically defines the corresponding `backward` function that computes the gradients.

Any legal tensor operation is fair game for the `forward` function, providing a log of design flexibility. In our simple case, we just link

the tensor through functional input-output relations after initializing their attributes:

```
import torch.nn as nn
class Net(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(Net, self).__init__() # Inherited from nn.Module
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.logistic = nn.LogSigmoid()
        self.fc2 = nn.Linear(hidden_size, num_classes)
        self.softmax = nn.Softmax(dim=1)

    def forward(self, x):
        """Forward pass: stacking each layer together"""
        out = self.fc1(x)
        out = self.logistic(out)
        out = self.fc2(out)
        out = self.softmax(out)
        return out
```

We then instantiate a `Net()` object and can inspect the architecture as follows:

```
net = Net(input_size, hidden_size, num_classes)
net
Net(
  (fc1): Linear(in_features=2, out_features=3, bias=True)
  (logistic): LogSigmoid()
  (fc2): Linear(in_features=3, out_features=2, bias=True)
  (softmax): Softmax()
)
```

To illustrate eager execution, we can also inspect the initialized parameters in the first tensor:

```
list(net.parameters())[0]
Parameter containing:
tensor([[ 0.3008, -0.2117],
        [-0.5846, -0.1690],
        [-0.6639,  0.1887]], requires_grad=True)
```

To enable GPU processing, you can use `net.cuda()`. See the PyTorch documentation for placing tensors on CPU and/or one or more GPU units.

We also need to define a loss function and the optimizer, using some of the built-in options:

```
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(net.parameters(), lr=learning_rate)
```

How to train the model

Model training consists of an outer loop for each epoch, that is, each pass over the training data, and an inner loop over the batches produced by the `Dataloader`. That executes the forward and backward passes of the learning algorithm. Some care needs to be taken to adjust data types to the requirements of the various objects and functions; for example, labels need to be integers and the features should be of type `float`:

```
for epoch in range(num_epochs):
    print(epoch)
    for i, (features, label) in enumerate(dataloader):

        features = Variable(features.float())
        label = Variable(label.long())
        # Initialize the hidden weights
        optimizer.zero_grad()

        # Forward pass: compute output given features
        outputs = net(features)

        # Compute the loss
        loss = criterion(outputs, label)
        # Backward pass: compute the gradients
        loss.backward()
        # Update the weights
        optimizer.step()
```

The notebook also contains an example that uses the `livelossplot` package to plot losses throughout the training process as provided by Keras out of the box.

How to evaluate the model predictions

To obtain predictions from our trained model, we pass it feature data and convert the prediction to a NumPy array. We get softmax probabilities for each of the two classes:

```
test_value = Variable(torch.from_numpy(X)).float()
prediction = net(test_value).data.numpy()
Prediction.shape
(50000, 2)
```

From here on, we can proceed as before to compute loss metrics or visualize the result that again reproduces a version of the decision boundary we found earlier.

Alternative options

The huge interest in DL has led to the development of several competing libraries that facilitate the design and training of NNs. The most prominent include the following examples (also see references on GitHub).

Apache MXNet

MXNet, incubated at the Apache Foundation, is an open source DL software framework used to train and deploy deep NNs. It focuses on scalability and fast model training. They included the Gluon

high-level interface to make it easy to prototype, train, and deploy DL models. MXNet has been picked by Amazon for deep learning on AWS.

Microsoft Cognitive Toolkit (CNTK)

The Cognitive Toolkit, previously known as CNTK, is Microsoft's contribution to the deep learning library collection. It describes an NN as a series of computational steps via a directed graph, similar to TensorFlow. In this directed graph, leaf nodes represent input values or network parameters, while other nodes represent matrix operations upon their inputs. CNTK allows users to build and combine popular model architectures ranging from deep feedforward NNs, convolutional networks, and recurrent networks (RNNs/LSTMs).

Fastai

The fastai library aims to simplify training NNs that are fast and accurate using modern best practices. These practices have emerged from research into DL at the company that makes both the software and accompanying courses available for free. Fastai includes support for models that process image, text, tabular, and collaborative filtering data.

Optimizing an NN for a long-short strategy

In practice, we need to explore variations for the design options for the NN architecture and how we train it from those we outlined previously because we can never be sure from the outset which configuration best suits the data. In this section, we will explore various architectures for a simple feedforward NN to predict daily stock returns using the dataset developed in *Chapter 12* (see the notebook `preparing_the_model_data` in the GitHub directory for that chapter).

To this end, we will define a function that returns a TensorFlow model based on several architectural input parameters and cross-validate alternative designs using the `MultipleTimeSeriesCV` we introduced in *Chapter 7, Linear Models – From Risk Factors to Return Forecasts*. To assess the signal quality of the model predictions, we build a simple ranking-based long-short strategy based on an ensemble of the models that perform best during the in-sample cross-validation period. To limit the risk of false discoveries, we then evaluate the performance of this strategy for an out-of-sample test period.

See the `optimizing_a_NN_architecture_for_trading` notebook for details.

Engineering features to predict daily stock returns

To develop our trading strategy, we use the daily stock returns for 995 US stocks for the eight-year period from 2010 to 2017. We will use the features developed in *Chapter 12, Boosting Your Trading Strategy* that include volatility and momentum factors, as well as lagged returns with cross-sectional and sectoral rankings. We load the data as follows:

```
data = pd.read_hdf('../12_gradient_boosting_machines/data/data.h5',
                   'model_data').dropna()
outcomes = data.filter(like='fwd').columns.tolist()
lookahead = 1
outcome= f'r{lookahead:02}_fwd'
X = data.loc[idx[:, :2017], :].drop(outcomes, axis=1)
y = data.loc[idx[:, :2017], outcome]
```

Defining an NN architecture framework

To automate the generation of our TensorFlow model, we create a function that constructs and compiles the model based on arguments that can later be passed during cross-validation iterations.

The following `make_model` function illustrates how to flexibly define various architectural elements for the search process. The `dense_layers` argument defines both the depth and width of the network as a list of integers. We also use `dropout` for regularization, expressed as a float in the range [0, 1] to define the probability that a given unit will be excluded from a training iteration:

```
def make_model(dense_layers, activation, dropout):
    '''Creates a multi-layer perceptron model

    dense_layers: List of layer sizes; one number per layer
    '''
    model = Sequential()
    for i, layer_size in enumerate(dense_layers, 1):
        if i == 1:
            model.add(Dense(layer_size, input_dim=X_cv.shape[1]))
            model.add(Activation(activation))
        else:
            model.add(Dense(layer_size))
            model.add(Activation(activation))
    model.add(Dropout(dropout))
    model.add(Dense(1))
    model.compile(loss='mean_squared_error',
```

```
        optimizer='Adam')
return model
```

Now we can turn to the cross-validation process to evaluate various NN architectures.

Cross-validating design options to tune the NN

We use the `MultipleTimeSeriesCV` to split the data into rolling training and validation sets comprising of $24 * 12$ months of data, while keeping the final $12 * 21$ days of data (starting November 30, 2016) as a holdout test. We train each model for 48 21-day periods and evaluate its results over 3 21-day periods, implying 12 splits for cross-validation and test periods combined:

```
n_splits = 12
train_period_length=21 * 12 * 4
test_period_length=21 * 3
cv = MultipleTimeSeriesCV(n_splits=n_splits,
                           train_period_length=train_period_length,
                           test_period_length=test_period_length,
                           lookahead=lookahead)
```

Next, we define a set of configurations for cross-validation. These include several options for two hidden layers and dropout probabilities; we'll only use tanh activations because a trial run did not suggest significant differences compared to ReLU. (We could also try out different optimizers, but I recommend you do not run this experiment, to limit what is already a computationally intensive effort):

```
dense_layer_opts = [(16, 8), (32, 16), (32, 32), (64, 32)]
dropout_opts = [.0, .1, .2]
param_grid = list(product(dense_layer_opts, activation_opts, dropout_opts))
np.random.shuffle(param_grid)
len(param_grid)
12
```

To run the cross-validation, we define a function that produces the train and validation data based on the integer indices produced by the `MultipleTimeSeriesCV` as follows:

```
def get_train_valid_data(X, y, train_idx, test_idx):
    x_train, y_train = X.iloc[train_idx, :], y.iloc[train_idx]
    x_val, y_val = X.iloc[test_idx, :], y.iloc[test_idx]
    return x_train, y_train, x_val, y_val
```

During cross-validation, we train a model using one set of parameters from the previously defined grid for 20 epochs. After each epoch, we store a `checkpoint` that contains the learned weights that we can reload to quickly generate predictions for the best configuration without retraining.

After each epoch, we compute and store the **information coefficient (IC)** for the validation set by day:

```
ic = []
scaler = StandardScaler()
for params in param_grid:
    dense_layers, activation, dropout = params
    for batch_size in [64, 256]:
        checkpoint_path = checkpoint_dir / str(dense_layers) / activation / str(dropout) / str(batch_size)
        for fold, (train_idx, test_idx) in enumerate(cv.split(X_cv)):
            x_train, y_train, x_val, y_val = get_train_valid_data(X_cv, y_cv, train_idx, test_idx)
            x_train = scaler.fit_transform(x_train)
            x_val = scaler.transform(x_val)
            preds = y_val.to_frame('actual')
```

```

r = pd.DataFrame(index=y_val.groupby(level='date').size())
model = make_model(dense_layers, activation, dropout)
for epoch in range(20):
    model.fit(x_train, y_train,
               batch_size=batch_size,
               epochs=1, validation_data=(x_val, y_val))
model.save_weights(
    (checkpoint_path / f'ckpt_{fold}_{epoch}').as_posix())
preds[epoch] = model.predict(x_val).squeeze()
r[epoch] = preds.groupby(level='date').apply(lambda x:
ic.append(r.assign(dense_layers=str(dense_layers),
                     activation=activation,
                     dropout=dropout,
                     batch_size=batch_size,
                     fold=fold)))

```

With an NVIDIA GTX 1080 GPU, 20 epochs takes a bit over one hour with batches of 64 samples, and around 20 minutes with 256 samples.

Evaluating the predictive performance

Let's first take a look at the five models that achieved the highest median daily IC during the cross-validation period. The following code computes these values:

```

dates = sorted(ic.index.unique())
cv_period = 24 * 21
cv_dates = dates[:cv_period]
ic_cv = ic.loc[cv_dates]
(ic_cv.drop('fold', axis=1).groupby(params).median().stack()
 .to_frame('ic').reset_index().rename(columns={'level_3': 'epoch'})
 .nlargest(n=5, columns='ic'))

```

The resulting table shows that the architectures using 32 units in both layers and 16/8 in the first/second layer, respectively, performed best. These models also use `dropout` and were trained

with batch sizes of 64 samples with the given number of epochs for all folds. The median IC values vary between 0.0236 and 0.0246:

Dense Layers	Dropout	Batch Size	Epoch	IC
(32, 32)	0.1	64	7	0.0246
(16, 8)	0.2	64	14	0.0241
(16, 8)	0.1	64	3	0.0238
(32, 32)	0.1	64	10	0.0237
(16, 8)	0.2	256	3	0.0236

Next, we'll take a look at how the parameter choices impact the predictive performance.

First, we visualize the daily information coefficient (averaged per fold) for different configurations by epoch to understand how the duration of training affects the predictive accuracy. The plots in *Figure 17.7*, however, highlight few conclusive patterns; the IC varies little across models and not particularly systematically across epochs:

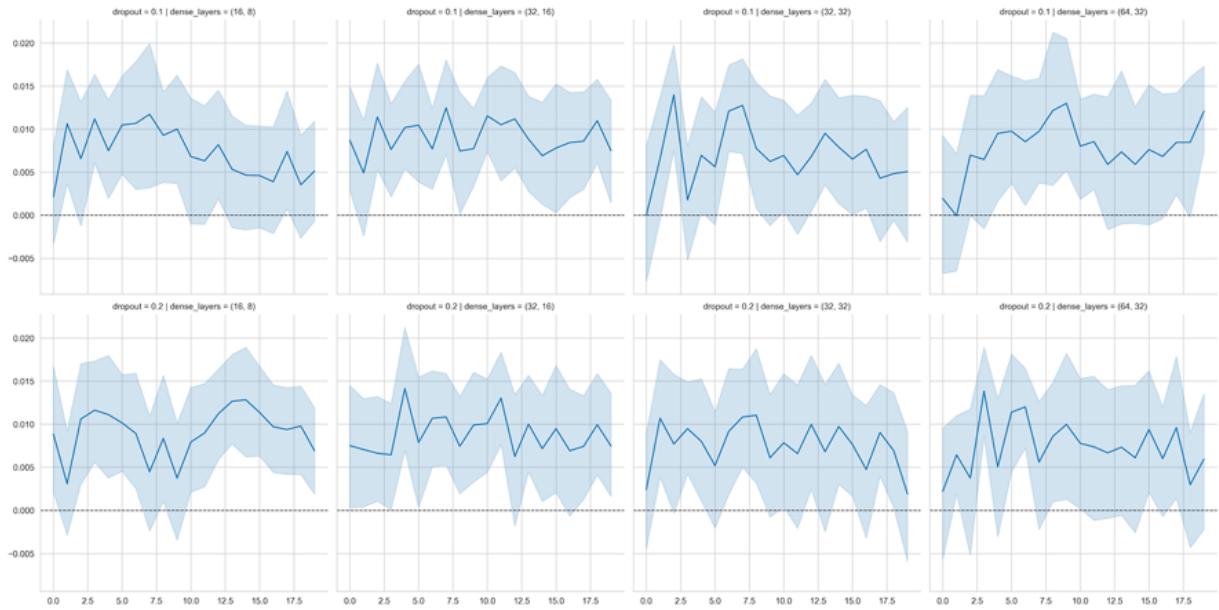


Figure 17.7: Information coefficients for various model configurations

For more statistically robust insights, we run a linear regression using **ordinary least squares (OLS)** (see *Chapter 7, Linear Models – From Risk Factors to Return Forecasts*) using dummy variables for the layer, dropout, and batch size choices as well as for each epoch:

```
data = pd.melt(ic, id_vars=params, var_name='epoch', value_name='ic')
data = pd.get_dummies(data, columns=['epoch'] + params, drop_first=True)
model = sm.OLS(endog=data.ic, exog=sm.add_constant(data.drop('ic', axis=1), drop=True))
```

The chart in *Figure 17.8* plots the confidence interval for each regression coefficient; if it does not include zero, then the coefficient is significant at the five percent level. The IC values on the y-axis reflect the differential from the constant (0.0027, p-value: 0.017) that represents the sample average over the configuration excluded while dropping one category of each dummy variable.

Across all configurations, batch size 256 and a dropout of 0.2 made significant (but small) positive contributions to performance.

Similarly, training for seven epochs yielded slightly superior results. The regression is overall significant according to the F statistic but has a very low R² value close to zero, underlining the high degree of noise in the data relative to the signal conveyed by the parameter choices.

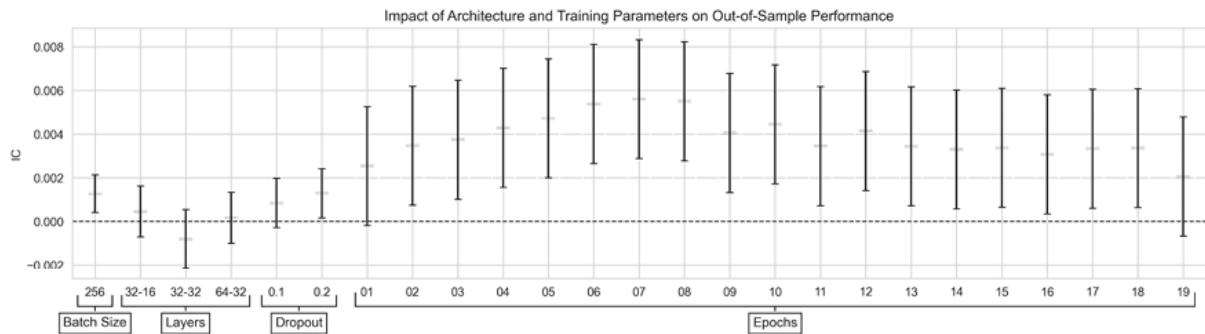


Figure 17.8: OLS coefficients and confidence intervals

Backtesting a strategy based on ensembled signals

To translate our NN model into a trading strategy, we generate predictions, evaluate their signal quality, create rules that define how to trade on these predictions, and backtest the performance of a strategy that implements these rules. See the notebook

`backtesting_with_zipline` for the code examples in this section.

Ensembling predictions to produce tradeable signals

To reduce the variance of the predictions and hedge against in-sample overfitting, we combine the predictions of the best three models listed in the table in the previous section and average the result.

To this end, we define the following `generate_predictions()` function, which receives the model parameters as inputs, loads the weights for the models for the desired epoch, and creates forecasts for the cross-validation and out-of-sample periods (showing only the essentials here to save some space):

```
def generate_predictions(dense_layers, activation, dropout,
                        batch_size, epoch):
    checkpoint_dir = Path('logs')
    checkpoint_path = checkpoint_dir / dense_layers / activation /
                      str(dropout) / str(batch_size)

    for fold, (train_idx, test_idx) in enumerate(cv.split(X_cv)):
        x_train, y_train, x_val, y_val = get_train_valid_data(X_cv, y_cv,
                                                               train_idx,
                                                               test_idx)
        x_val = scaler.fit(x_train).transform(x_val)
        model = make_model(dense_layers, activation, dropout, input_dim)
        status = model.load_weights(
            (checkpoint_path / f'ckpt_{fold}_{epoch}').as_posix())
        status.expect_partial()
        predictions.append(pd.Series(model.predict(x_val).squeeze(),
                                      index=y_val.index))
    return pd.concat(predictions)
```

We store the results for evaluation with Alphalens and a Zipline backtest.

Evaluating signal quality using Alphalens

To gain some insight into the signal content of the ensembled model predictions, we use Alphalens to compute the return differences for investments into five equal-weighted portfolios differentiated by the forecast quantiles (see *Figure 17.9*). The spread between the top and the bottom quintile equals around 8 bps for a one-day holding period, which implies an alpha of 0.094 and a beta of 0.107:

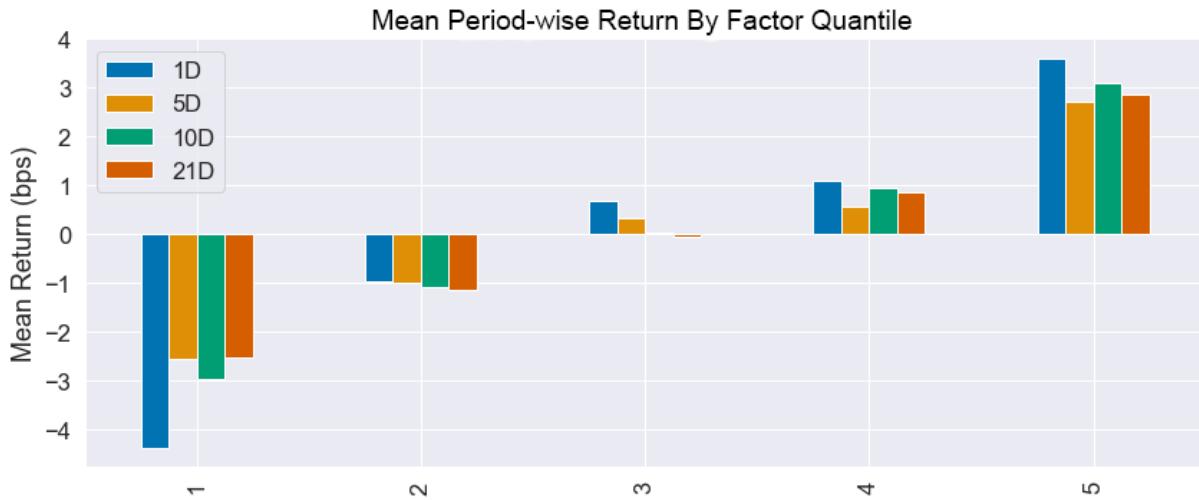


Figure 17.9: Signal quality evaluation

Backtesting the strategy using Zipline

Based on the Alphalens analysis, our strategy will enter long and short positions for the 50 stocks with the highest positive and lowest negative predicted returns, respectively, as long as there are at least 10 options on either side. The strategy trades every day.

The charts in *Figure 17.10* show that the strategy performs well in- and out-of-sample (before transaction costs):

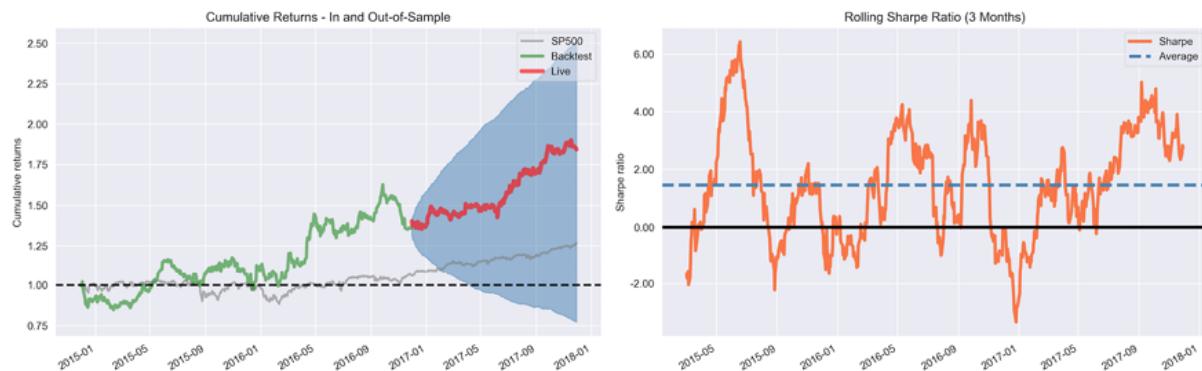


Figure 17.10: In- and out-of-sample backtest performance

It produces annualized returns of 22.8 percent over the 36-month period, 16.5 percent for the 24 in-sample months, and 35.7 percent for the 12 out-of-sample months. The Sharpe ratio is 0.72 in-sample and 2.15 out-of-sample, delivering an alpha of 0.18 (0.29) and a beta of 0.24 (0.16) in/out of sample.

How to further improve the results

The relatively simple architecture yields some promising results. To further improve performance, you can first and foremost add new features and more data to the model.

Alternatively, you can use more sophisticated architectures, including RNNs and CNNs, which are well suited to sequential data, whereas vanilla feedforward NNs are not designed to capture the ordered nature of the features.

We will turn to these specialized architectures in the following chapter.

Summary

In this chapter, we introduced DL as a form of representation learning that extracts hierarchical features from high-dimensional, unstructured data. We saw how to design, train, and regularize feedforward neural networks using NumPy. We demonstrated how to use the popular DL libraries PyTorch and TensorFlow that are suitable for use cases from rapid prototyping to production deployments.

Most importantly, we designed and tuned an NN using TensorFlow and were able to generate tradeable signals that delivered attractive returns during both the in-sample and out-of-sample periods.

In the next chapter, we will explore CNNs, which are particularly well suited for image data but are also well-suited for sequential data.

CNNs for Financial Time Series and Satellite Images

In this chapter, we introduce the first of several specialized deep learning architectures that we will cover in *Part 4*. Deep **convolutional neural networks (CNNs)** have enabled superhuman performance in various computer vision tasks such as classifying images and video and detecting and recognizing objects in images. CNNs can also extract signals from time-series data that shares certain characteristics with image data and have been successfully applied to speech recognition (Abdel-Hamid et al. 2014). Moreover, they have been shown to deliver state-of-the-art performance on time-series classification across various domains (Ismail Fawaz et al. 2019).

CNNs are named after a linear algebra operation called a **convolution** that replaces the general matrix multiplication typical of feedforward networks (discussed in the last chapter) in at least one of their layers. We will show how convolutions work and why they are particularly well suited to data with a certain regular structure typically found in images but also present in time series.

Research into **CNN architectures** has proceeded very rapidly, and new architectures that improve benchmark performance continue to emerge. We will describe a set of building blocks consistently used by successful applications. We will also demonstrate how **transfer**

learning can speed up learning by using pretrained weights for CNN layers closer to the input while fine-tuning the final layers to a specific task. We will also illustrate how to use CNNs for the specific computer vision task of **object detection**.

CNNs can help build a **trading strategy** by generating signals from images or (multiple) time-series data:

- **Satellite data** may signal future commodity trends, including the supply of certain crops or raw materials via aerial images of agricultural areas, mines, or transport networks like oil tankers. **Surveillance camera** footage, for example, from shopping malls, could be used to track and predict consumer activity.
- **Time-series data** encompasses a very broad range of data sources and CNNs have been shown to deliver high-quality classification results by exploiting their structural similarity with images.

We will create a trading strategy based on predictions of a CNN that uses time-series data that's been deliberately formatted like images and demonstrate how to build a CNN to classify satellite images.

More specifically, in this chapter, you will learn about the following:

- How CNNs employ several building blocks to efficiently model grid-like data
- Training, tuning, and regularizing CNNs for images and time-series data using TensorFlow
- Using transfer learning to streamline CNNs, even with less data
- Designing a trading strategy using return predictions by a CNN trained on time-series data formatted like images
- How to classify satellite images



You can find the code samples for this chapter and links to additional resources in the corresponding directory of the GitHub repository. The notebooks include color versions of the images.

How CNNs learn to model grid-like data

CNNs are conceptually similar to feedforward **neural networks (NNs)**: they consist of units with parameters called weights and biases, and the training process adjusts these parameters to optimize the network's output for a given input according to a loss function. They are most commonly used for classification. Each unit uses its parameters to apply a linear operation to the input data or activations received from other units, typically followed by a nonlinear transformation.

The overall network models a **differentiable function** that maps raw data, such as image pixels, to class probabilities using an output activation function like softmax. CNNs use an objective function such as cross-entropy loss to measure the quality of the output with a single metric. They also rely on the gradients of the loss with respect to the network parameter to learn via backpropagation.

Feedforward NNs with fully connected layers do not scale well to high-dimensional image data with a large number of pixel values. Even the low-resolution images included in the CIFAR-10 dataset that we'll use in the next section contain 32×32 pixels with up to 256 different color values represented by 8 bits each. With three channels, for example, for the red, green, and blue channels of the

RGB color model, a single unit in a fully connected input layer implies $32 \times 32 \times 3 = 3,072$ weights. A more standard resolution of 640×480 pixels already yields closer to 1 million weights for a single input unit. Deep architectures with several layers of meaningful width quickly lead to an exploding number of parameters that make overfitting during training all but certain.

A fully connected feedforward NN makes no assumptions about the local structure of the input data so that arbitrarily reordering the features has no impact on the training result. By contrast, CNNs make the **key assumption** that the **data has a grid-like topology** and that the **local structure matters**. In other words, they encode the assumption that the input has a structure typically found in image data: pixels form a two-dimensional grid, possibly with several channels to represent the components of the color signal.

Furthermore, the values of nearby pixels are likely more relevant to detect key features such as edges and corners than faraway data points. Naturally, initial CNN applications such as handwriting recognition focused on image data.

Over time, however, researchers recognized **similar characteristics in time-series data**, broadening the scope for the productive use of CNNs. Time-series data consists of measurements at regular intervals that create a one-dimensional grid along the time axis, such as the lagged returns for a given ticker. There can also be a second dimension with additional features for this ticker and the same time periods. Finally, we could represent additional tickers using the third dimension.

A common CNN use case beyond images includes audio data, either in a one-dimensional waveform in the time domain or, after a Fourier transform, as a two-dimensional spectrum in the frequency

domain. CNNs also play a key role in AlphaGo, the first algorithm to win a game of Go against humans, where they evaluated different positions on the grid-like board.

The most important element to encode the **assumption of a grid-like topology** is the **convolution** operation that gives CNNs their name, combined with **pooling**. We will see that the specific assumptions about the functional relationship between input and output data imply that CNNs need far fewer parameters and compute more efficiently.

In this section, we will explain how convolution and pooling layers learn filters that extract local features and why these operations are particularly suitable for data with the structure just described. State-of-the-art CNNs combine many of these basic building blocks to achieve the layered representation learning described in the previous chapter. We conclude by describing key architectural innovations over the last decade that saw enormous performance improvements.

From hand-coding to learning filters from data

For image data, this local structure has traditionally motivated the development of hand-coded filters that extract such patterns for the use as features in **machine learning (ML)** models.

Figure 18.1 displays the effect of simple filters designed to detect certain edges. The notebook `filter_example.ipynb` illustrates how to use hand-coded filters in a convolutional network and visualizes the resulting transformation of the image. The filters are simple $[-1, 1]$ patterns arranged in a 2×2 matrix, shown in the upper right of the

figure. Below each filter, its effects are shown; they are a bit subtle and will be easier to spot in the accompanying notebook.

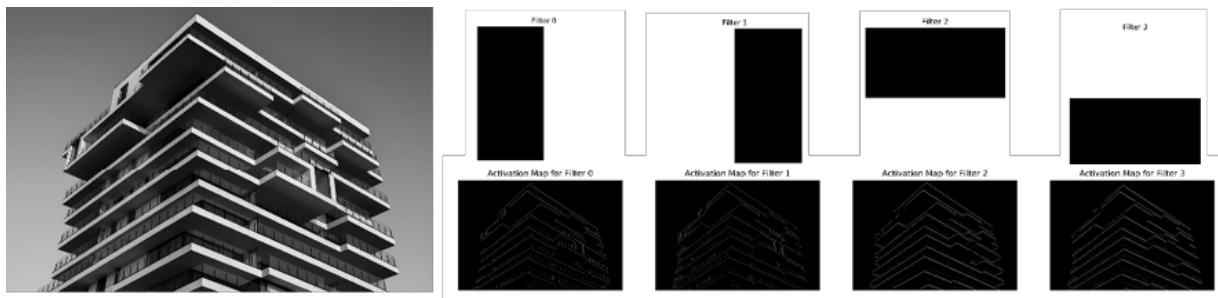


Figure 18.1: The result of basic edge filters applied to an image

Convolutional layers, by contrast, are designed to learn such local feature representations from the data. A key insight is to restrict their input, called the **receptive field**, to a small area of the input so it captures basic pixel constellations that reflect common patterns like edges or corners. Such patterns may occur anywhere in an image, though, so CNNs also need to recognize similar patterns in different locations and possibly with small variations.

Subsequent layers then learn to synthesize these local features to detect **higher-order features**. The linked resources on GitHub include examples of how to visualize the filters learned by a deep CNN using some of the deep architectures that we present in the next section on reference architectures.

How the elements of a convolutional layer operate

Convolutional layers integrate **three architectural ideas** that enable the learning of feature representations that are to some degree invariant to shifts, changes in scale, and distortion:

- Sparse rather than dense connectivity
- Weight sharing
- Spatial or temporal downsampling

Moreover, convolutional layers allow for inputs of variable size. We will walk through a typical convolutional layer and describe each of these ideas in turn.

Figure 18.2 outlines the set of operations that typically takes place in a three-dimensional convolutional layer, assuming image data is input with the three dimensions of height, width, and depth, or the number of channels. The range of pixel values depends on the bit representation, for example, $[0, 255]$ for 8 bits. Alternatively, the width axis could represent time, the height different features, and the channels could capture observations on distinct objects such as tickers.

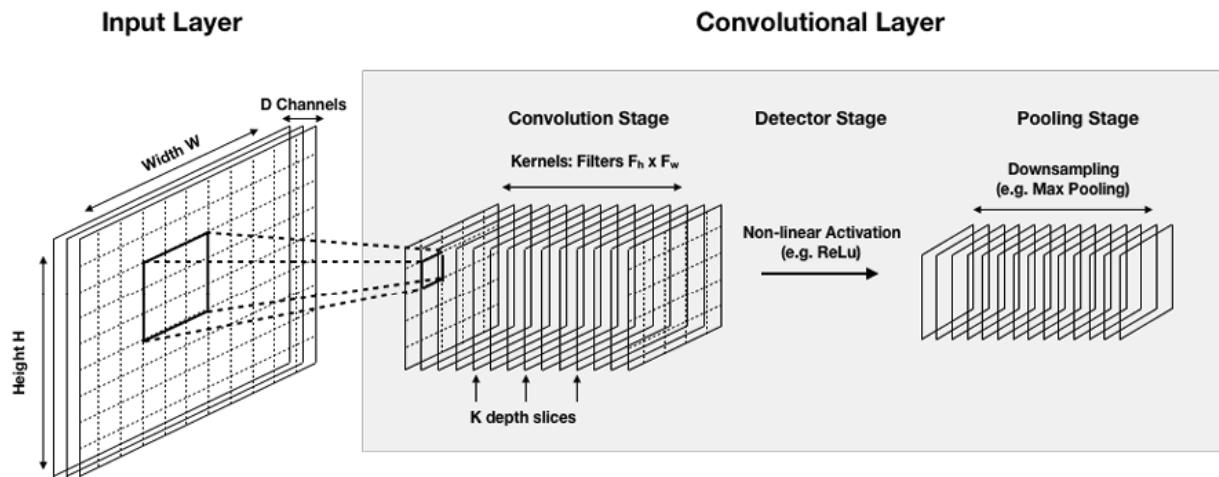


Figure 18.2: Typical operations in a two-dimensional convolutional layer

Successive computations process the input through the convolutional, detector, and pooling stages that we describe in the next three sections. In the example depicted in *Figure 18.2*, the

convolutional layer receives three-dimensional input and produces an output of the same dimensionality.

State-of-the-art CNNs are composed of several such layers of varying sizes that are either stacked on top of each other or operate in parallel on different branches. With each layer, the network can detect higher-level, more abstract features.

The convolution stage – extracting local features

The first stage applies a filter, also called the **kernel**, to overlapping patches of the input image. The filter is a matrix of a much smaller size than the input so that its receptive field is limited to a few contiguous values such as pixels or time-series values. As a result, it focuses on local patterns and dramatically reduces the number of parameters and computations relative to a fully connected layer.

A complete convolutional layer has several **feature maps** organized as depth slices (depicted in *Figure 18.2*) so that each layer can extract multiple features.

From filters to feature maps

While scanning the input, the kernel is convolved with each input segment covered by its receptive field. The convolution operation is simply the dot product between the filter weights and the values of the matching input area after both have been reshaped to vectors. Each convolution thus produces a single number, and the entire scan yields a feature map. Since the dot product is maximized for identical vectors, the feature map indicates the degree of activation for each input region.

Figure 18.3 illustrates the result of the scan of a 5×5 input using a 3×3 filter with given values, and how the activation in the upper-right corner of the feature map results from the dot product of the flattened input region and the kernel:

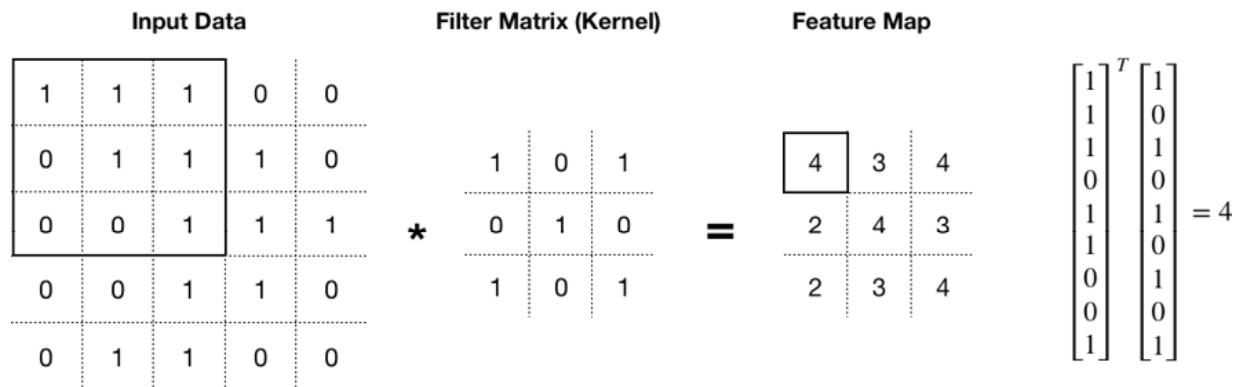


Figure 18.3: From convolutions to a feature map

The most important aspect is that the **filter values are the parameters** of the convolutional layers, **learned from the data** during training to minimize the chosen loss function. In other words, CNNs learn useful feature representations by finding kernel values that activate input patterns that are most useful for the task at hand.

How to scan the input – strides and padding

The **stride** defines the step size used for scanning the input, that is, the number of pixels to shift horizontally and vertically. Smaller strides scan more (overlapping) areas but are computationally more expensive. Four options are commonly used when the filter does not fit the input perfectly and partially crosses the image boundary during the scan:

- **Valid convolution:** Discards scans where the image and filter do not perfectly match

- **Same convolution:** Zero-pads the input to produce a feature map of equal size
- **Full convolution:** Zero-pads the input so that each pixel is scanned an equal number of times, including pixels at the border (to avoid oversampling pixels closer to the center)
- **Causal:** Zero-pads the input only on the left so that the output does not depend on an input from a later period; maintains the temporal order for time-series data

The choices depend on the nature of the data and where useful features are most likely located. In combination with the number of depth slices, they determine the output size of the convolution stage. The Stanford lecture notes by Andrew Karpathy (see GitHub) contain helpful examples using NumPy.

Parameter sharing for robust features and fast computation

The location of salient features may vary due to distortion or shifts. Furthermore, elementary feature detectors are likely useful across the entire image. CNNs encode these assumptions by sharing or tying the weights for the filter in a given depth slice.

As a result, each depth slice specializes in a certain pattern and the number of parameters is further reduced. Weight sharing works less well, however, when images are spatially centered and key patterns are less likely to be uniformly distributed across the input area.

The detector stage – adding nonlinearity

The feature maps are usually passed through a nonlinear transformation. The **rectified linear unit (ReLU)** that we encountered in the last chapter is a common function for this

purpose. ReLUs replace negative activations element-wise by zero and mitigate the risk of vanishing gradients found in other activation functions such as tanh (see *Chapter 17, Deep Learning for Trading*).

A popular alternative is the **softplus function**:

$$f(x) = \ln(1 + e^x)$$

In contrast to ReLU, it has a derivative everywhere, namely the sigmoid function that we used for logistic regression (see *Chapter 7, Linear Models – From Risk Factors to Return Forecasts*).

The pooling stage – downsampling the feature maps

The last stage of the convolutional layer may downsample the feature map's input representation to do the following:

- Reduce its dimensionality and prevent overfitting
- Lower the computational cost
- Enable basic translation invariance

This assumes that the precise location of the features is not only less important for identifying a pattern but can even be harmful because it will likely vary for different instances of the target. Pooling lowers the spatial resolution of the feature map as a simple way to render the location information less precise. However, this step is optional and many architectures use pooling only for some layers or not at all.

A common pooling operation is **max pooling**, which uses only the maximum activation value from (typically) non-overlapping subregions. For a small 4×4 feature map, for instance, 2×2 max pooling outputs the maximum for each of the four non-overlapping

2×2 areas. Less common pooling operators use the average or the median. Pooling does not add or learn new parameters but the size of the input window and possibly the stride are additional hyperparameters.

The evolution of CNN architectures – key innovations

Several CNN architectures have pushed performance boundaries over the past two decades by introducing important innovations. Predictive performance growth accelerated dramatically with the arrival of big data in the form of ImageNet (Fei-Fei 2015) with 14 million images assigned to 20,000 classes by humans via Amazon's Mechanical Turk. The **ImageNet Large Scale Visual Recognition Challenge (ILSVRC)** became the focal point of CNN progress around a slightly smaller set of 1.2 million images from 1,000 classes.

It is useful to be familiar with the **reference architectures** dominating these competitions for practical reasons. As we will see in the next section on working with CNNs for image data, they offer a good starting point for standard tasks. Moreover, **transfer learning** allows us to address many computer vision tasks by building on a successful architecture with pretrained weights. Transfer learning not only speeds up architecture selection and training but also enables successful applications on much smaller datasets.

In addition, many publications refer to these architectures, and they often serve as a basis for networks tailored to segmentation or localization tasks. We will further describe some landmark architectures in the section on image classification and transfer learning.

Performance breakthroughs and network size

The left side of *Figure 18.4* plots the top-1 accuracy against the computational cost of a variety of network architectures. It suggests a positive relationship between the number of parameters and performance, but also shows that the marginal benefit of more parameters declines and that architectural design and innovation also matter.

The right side plots the top-1 accuracy per parameter for all networks. Several new architectures target use cases on less powerful devices such as mobile phones. While they do not achieve state-of-the-art performance, they have found much more efficient implementations. See the resources on GitHub for more details on these architectures and the analysis behind these charts.

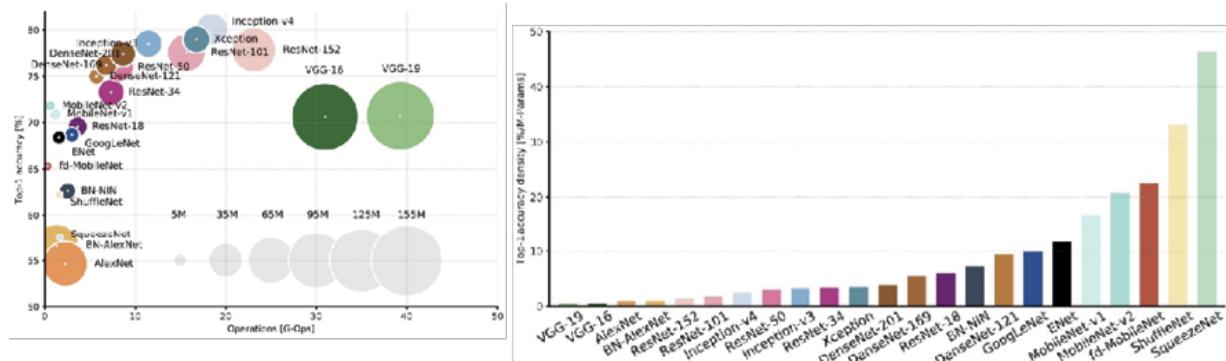


Figure 18.4: Predictive performance and computational complexity

Lessons learned

Some of the lessons learned from 20 years of CNN architecture developments, especially since 2012, include the following:

- **Smaller convolutional filters** perform better (possibly except at the first layer) because several small filters can substitute for a

larger filter at a lower computational cost.

- **1×1 convolutions** reduce the dimensionality of feature maps so that the network can learn a larger number overall.
- **Skip connections** are able to create multiple paths through the network and enable the training of much higher-capacity CNNs.

CNNs for satellite images and object detection

In this section, we demonstrate how to solve key computer vision tasks such as image classification and object detection. As mentioned in the introduction and in *Chapter 3, Alternative Data for Finance – Categories and Use Cases*, image data can inform a trading strategy by providing clues about future trends, changing fundamentals, or specific events relevant to a target asset class or investment universe. Popular examples include exploiting satellite images for clues about the supply of agricultural commodities, consumer and economic activity, or the status of manufacturing or raw material supply chains. Specific tasks might include the following, for example:

- **Image classification:** Identifying whether cultivated land for certain crops is expanding, or predicting harvest quality and quantities
- **Object detection:** Counting the number of oil tankers on a certain transport route or the number of cars in a parking lot, or identifying the locations of shoppers in a mall

In this section, we'll demonstrate how to design CNNs to automate the extraction of such information, both from scratch using popular

architectures and via transfer learning that fine-tunes pretrained weights to a given task. We'll also demonstrate how to detect objects in a given scene.

We will introduce key CNN architectures for these tasks, explain why they work well, and show how to train them using TensorFlow 2. We will also demonstrate how to source pretrained weights and fine-tune time. Unfortunately, satellite images with information directly relevant for a trading strategy are very costly to obtain and are not readily available. We will, however, demonstrate how to work with the EuroSat dataset to build a classifier that identifies different land uses. This brief introduction to CNNs for computer vision aims to demonstrate how to approach common tasks that you will likely need to tackle when aiming to design a trading strategy based on images relevant to the investment universe of your choice.

All the libraries we introduced in the last chapter provide support for convolutional layers; we'll focus on the Keras interface of TensorFlow 2. We are first going to illustrate the LeNet5 architecture using the MNIST handwritten digit dataset. Next, we'll demonstrate the use of data augmentation with AlexNet on CIFAR-10, a simplified version of the original ImageNet. Then we'll continue with transfer learning based on state-of-the-art architectures before we apply what we've learned to actual satellite images. We conclude with an example of object detection in real-life scenes.

LeNet5 – The first CNN with industrial applications

Yann LeCun, now the Director of AI Research at Facebook, was a leading pioneer in CNN development. In 1998, after several

iterations starting in the 1980s, LeNet5 became the first modern CNN used in real-world applications that introduced several architectural elements still relevant today.

LeNet5 was published in a very instructive paper, *Gradient-Based Learning Applied to Document Recognition* (LeCun et al. 1989), that laid out many of the central concepts. Most importantly, it promoted the insight that convolutions with learnable filters are effective at extracting related features at multiple locations with few parameters. Given the limited computational resources at the time, efficiency was of paramount importance.

LeNet5 was designed to recognize the handwriting on checks and was used by several banks. It established a new benchmark for classification accuracy, with a result of 99.2 percent on the MNIST handwritten digit dataset. It consists of three convolutional layers, each containing a nonlinear tanh transformation, a pooling operation, and a fully connected output layer. Throughout the convolutional layers, the number of feature maps increases while their dimensions decrease. It has a total of 60,850 trainable parameters (Lecun et al. 1998).

"Hello World" for CNNs – handwritten digit classification

In this section, we'll implement a slightly simplified version of LeNet5 to demonstrate how to build a CNN using a TensorFlow implementation. The original MNIST dataset contains 60,000 grayscale images in 28×28 pixel resolution, each containing a single handwritten digit from 0 to 9. A good alternative is the more challenging but structurally similar Fashion MNIST dataset that we encountered in *Chapter 13, Data-Driven Risk Factors and Asset*

Allocation with Unsupervised Learning. See the [digit_classification_with_lenet5](#) notebook for implementation details.

We can load it in Keras out of the box:

```
from tensorflow.keras.datasets import mnist
(X_train, y_train), (X_test, y_test) = mnist.load_data()
X_train.shape, X_test.shape
((60000, 28, 28), (10000, 28, 28))
```

Figure 18.5 shows the first ten images in the dataset and highlights significant variation among instances of the same digit. On the right, it shows how the pixel values for an individual image range from 0 to 255:

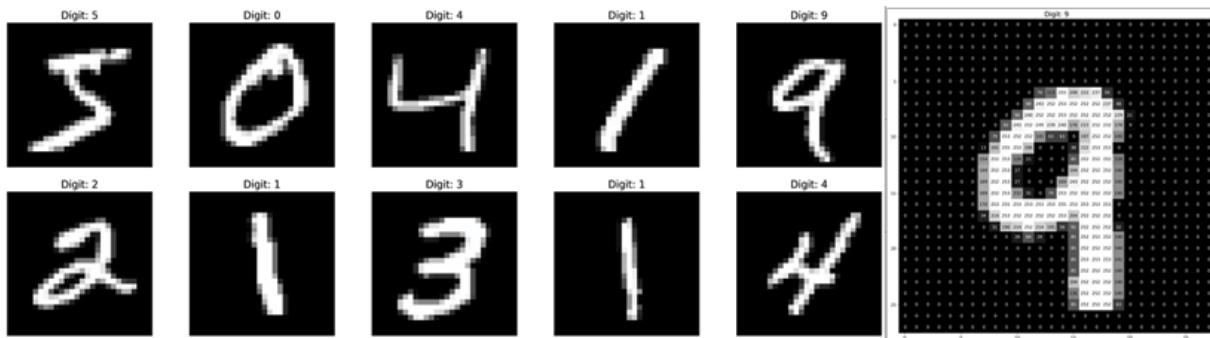


Figure 18.5: MNIST sample images

We rescale the pixel values to the range [0, 1] to normalize the training data and facilitate the backpropagation process and convert the data to 32-bit floats, which reduce memory requirements and computational cost while providing sufficient precision for our use case:

```
X_train = X_train.astype('float32')/255
X_test = X_test.astype('float32')/255
```

Defining the LeNet5 architecture

We can define a simplified version of LeNet5 that omits the original final layer containing radial basis functions as follows, using the default "valid" padding and single-step strides unless defined otherwise:

```
lenet5 = Sequential([
    Conv2D(filters=6, kernel_size=5, activation='relu',
           input_shape=(28, 28, 1), name='CONV1'),
    AveragePooling2D(pool_size=(2, 2), strides=(1, 1),
                     padding='valid', name='POOL1'),
    Conv2D(filters=16, kernel_size=(5, 5), activation='tanh', name='CONV2'),
    AveragePooling2D(pool_size=(2, 2), strides=(2, 2), name='POOL2'),
    Conv2D(filters=120, kernel_size=(5, 5), activation='tanh', name='CONV3'),
    Flatten(name='FLAT'),
    Dense(units=84, activation='tanh', name='FC6'),
    Dense(units=10, activation='softmax', name='FC7')
])
```

The summary indicates that the model thus defined has over 300,000 parameters:

Layer (type)	Output Shape	Param #	
CONV1 (Conv2D)	(None, 24, 24, 6)	156	
POOL1 (AveragePooling2D)	(None, 23, 23, 6)	0	
CONV2 (Conv2D)	(None, 19, 19, 16)	2416	
POOL2 (AveragePooling2D)	(None, 9, 9, 16)	0	
CONV3 (Conv2D)	(None, 5, 5, 120)	48120	
FLAT (Flatten)	(None, 3000)	0	
FC6 (Dense)	(None, 84)	252084	
FC7 (Dense)	(None, 10)	850	=====
Total params:	303,626		
Trainable params:	303,626		

We compile with `sparse_crossentropy_loss`, which accepts integers rather than one-hot-encoded labels and the original stochastic gradient optimizer:

```
lenet5.compile(loss='sparse_categorical_crossentropy',
                optimizer='SGD',
                metrics=['accuracy'])
```

Training and evaluating the model

Now we are ready to train the model. The model expects four-dimensional input, so we reshape accordingly. We use the standard batch size of 32 and an 80:20 train-validation split. Furthermore, we leverage checkpointing to store the model weights if the validation error improves, and make sure the dataset is randomly shuffled. We also define an `early_stopping` callback to interrupt training once the validation accuracy no longer improves for 20 iterations:

```
lenet_history = lenet5.fit(X_train.reshape(-1, 28, 28, 1),
                           y_train,
                           batch_size=32,
                           epochs=100,
                           validation_split=0.2, # use 0 to train on all
                           callbacks=[checkpointer, early_stopping],
                           verbose=1,
                           shuffle=True)
```

The training history records the last improvement after 81 epochs that take around 4 minutes on a single GPU. The test accuracy of this sample run is 99.09 percent, almost exactly the same result as for the original LeNet5:

```
accuracy = lenet5.evaluate(X_test.reshape(-1, 28, 28, 1), y_test, verbose=0)
print('Test accuracy: {:.2%}'.format(accuracy))
```



For comparison, a simple two-layer feedforward network achieves "only" 97.04 percent test accuracy (see the notebook). The LeNet5 improvement on MNIST is, in fact, modest. Non-neural methods have also achieved classification accuracies greater than or equal to 99 percent, including K-nearest neighbors and support vector machines. CNNs really shine with more challenging datasets as we will see next.

AlexNet – reigniting deep learning research

AlexNet, developed by Alex Krizhevsky, Ilya Sutskever, and Geoff Hinton at the University of Toronto, dramatically reduced the error rate and significantly outperformed the runner-up at the 2012 ILSVRC, achieving a top-5 error of 16 percent versus 26 percent (Krizhevsky, Sutskever, and Hinton 2012). This breakthrough triggered a renaissance in ML research and put deep learning for computer vision firmly on the global technology map.

The AlexNet architecture is similar to LeNet, but much deeper and wider. It is often credited with discovering **the importance of depth** with around 60 million parameters, exceeding LeNet5 by a factor of 1,000, a testament to increased computing power, especially the use of GPUs, and much larger datasets.

It included convolutions stacked on top of each other rather than combining each convolution with a pooling stage, and successfully used dropout for regularization and ReLU for efficient nonlinear transformations. It also employed data augmentation to increase the number of training samples, added weight decay, and used a more

efficient implementation of convolutions. It also accelerated training by distributing the network over two GPUs.

The notebook `image_classification_with_alexnet.ipynb` has a slightly simplified version of AlexNet tailored to the CIFAR-10 dataset that contains 60,000 images from 10 of the original 1,000 classes. It has been compressed to a 32×32 pixel resolution from the original 224×224 , but still has three color channels.

See the notebook `image_classification_with_alexnet` for implementation details; we will skip over some repetitive steps here.

Preprocessing CIFAR-10 data using image augmentation

CIFAR-10 can also be downloaded using TensorFlow's Keras interface, and we rescale the pixel values and one-hot encode the ten class labels as we did with MNIST in the previous section.

We first train a two-layer feedforward network on 50,000 training samples for 45 epochs to achieve a test accuracy of 45.78 percent. We also experiment with a three-layer convolutional net with over 528,000 parameters that achieves 74.51 percent test accuracy (see the notebook).

A common trick to enhance performance is to artificially increase the size of the training set by creating synthetic data. This involves randomly shifting or horizontally flipping the image or introducing noise into the image. TensorFlow includes an `ImageDataGenerator` class for this purpose. We can configure it and fit the training data as follows:

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator
datagen = ImageDataGenerator(
    width_shift_range=0.1, # randomly horizontal shift
```

```
height_shift_range=0.1, # randomly vertical shift
horizontal_flip=True) # randomly horizontal flip
datagen.fit(X_train)
```

The result shows how the augmented images (in low 32×32 resolution) have been altered in various ways as expected:



Figure 18.6: Original and augmented samples

The test accuracy for the three-layer CNN improves modestly to 76.71 percent after training on the larger, augmented data.

Defining the model architecture

We need to adapt the AlexNet architecture to the lower dimensionality of CIFAR-10 images relative to the ImageNet samples used in the competition. To this end, we use the original number of filters but make them smaller (see the notebook for implementation details).

The summary (see the notebook) shows the five convolutional layers followed by two fully connected layers with frequent use of batch normalization, for a total of 21.5 million parameters.

Comparing AlexNet performance

In addition to AlexNet, we trained a 2-layer feedforward NN and a 3-layer CNN, the latter with and without image augmentation. After 100 epochs (with early stopping if the validation accuracy does not improve for 20 rounds), we obtain the cross-validation trajectories and test accuracy for the four models, as displayed in *Figure 18.7*:

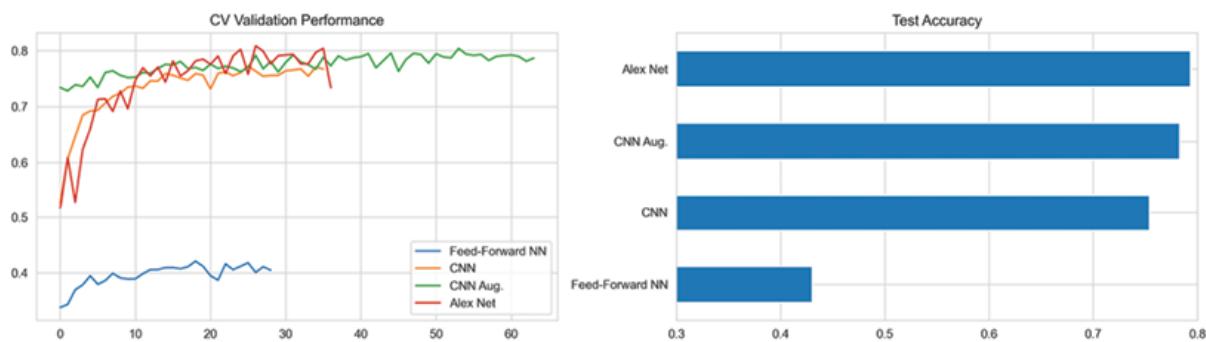


Figure 18.7: Validation performance and test accuracy on CIFAR-10

AlexNet achieves the highest test accuracy with 79.33 percent after some 35 epochs, closely followed by the shallower CNN with augmented images at 78.29 percent that trains for longer due to the larger dataset. The feedforward NN performs much worse than on MNIST on this more complex dataset, with a test accuracy of 43.05 percent.

Transfer learning – faster training with less data

In practice, sometimes we do not have enough data to train a CNN from scratch with random initialization. **Transfer learning** is an ML technique that repurposes a model trained on one set of data for another task. Naturally, it works if the learning from the first task

carries over to the task of interest. If successful, it can lead to better performance and faster training that requires less labeled data than training a neural network from scratch on the target task.

Alternative approaches to transfer learning

The transfer learning approach to CNN relies on pretraining on a very large dataset like ImageNet. The goal is for the convolutional filters to extract a feature representation that generalizes to new images. In a second step, it leverages the result to either initialize and retrain a new CNN or use it as input to a new network that tackles the task of interest.

As discussed, CNN architectures typically use a sequence of convolutional layers to detect hierarchical patterns, adding one or more fully connected layers to map the convolutional activations to the outcome classes or values. The output of the last convolutional layer that feeds into the fully connected part is called the **bottleneck features**. We can use the **bottleneck features** of a pretrained network as inputs into a new fully connected network, usually after applying a ReLU activation function.

In other words, we freeze the convolutional layers and **replace the dense part of the network**. An additional benefit is that we can then use inputs of different sizes because it is the dense layers that constrain the input size.

Alternatively, we can use the bottleneck features as **inputs into a different machine learning algorithm**. In the AlexNet architecture, for instance, the bottleneck layer computes a vector with 4,096 entries for each 224×224 input image. We then use this vector as features for a new model.

We also can go a step further and not only replace and retrain the final layers using new data but also **fine-tune the weights of the pretrained CNN**. To achieve this, we continue training, either only for later layers while freezing the weights of some earlier layers, or for all layers. The motivation is presumably to preserve more generic patterns learned by lower layers, such as edge or color blob detectors, while allowing later layers of the CNN to adapt to the details of a new task. ImageNet, for example, contains a wide variety of dog breeds, which may lead to feature representations specifically useful for differentiating between these classes.

Building on state-of-the-art architectures

Transfer learning permits us to leverage top-performing architectures without incurring the potentially fairly GPU- and data-intensive training. We briefly outline the key characteristics of a few additional popular architectures that are popular starting points.

VGGNet – more depth and smaller filters

The runner-up in ILSVRC 2014 was developed by Oxford University's Visual Geometry Group (VGG, Simonyan 2015). It demonstrated the effectiveness of **much smaller 3×3 convolutional filters** combined in sequence and reinforced the importance of depth for strong performance. VGG16 contains 16 convolutional and fully connected layers that only perform 3×3 convolutions and 2×2 pooling (see *Figure 18.5*).

VGG16 has **140 million parameters** that increase the computational costs of training and inference as well as the memory requirements. However, most parameters are in the fully connected layers that were since discovered not to be essential so that removing them

greatly reduces the number of parameters without negatively impacting performance.

GoogLeNet – fewer parameters through Inception

Christian Szegedy at Google reduced the computational costs using more efficient CNN implementations to facilitate practical applications at scale. The resulting GoogLeNet (Szegedy et al. 2015) won the ILSVRC 2014 with only 4 million parameters due to the **Inception module**, compared to AlexNet's 60 million and VGG16's 140 million.

The Inception module builds on the **network-in-network concept** that uses 1×1 convolutions to compress a deep stack of convolutional filters and thus reduce the cost of computation. The module uses parallel 1×1 , 3×3 , and 5×5 filters, combining the latter two with 1×1 convolutions to reduce the dimensionality of the filters passed in by the previous layer.

In addition, it uses average pooling instead of fully connected layers on top of the convolutional layers to eliminate many of the less impactful parameters. There have been several enhanced versions, most recently Inception-v4.

ResNet – shortcut connections beyond human performance

The **residual network (ResNet)** architecture was developed at Microsoft and won the ILSVRC 2015. It pushed the top-5 error to 3.7 percent, below the level of human performance on this task of around 5 percent (He et al. 2015).

It introduces identity shortcut connections that skip several layers and overcome some of the challenges of training deep networks,

enabling the use of hundreds or even over a thousand layers. It also heavily uses batch normalization, which was shown to allow higher learning rates and be more forgiving about weight initialization. The architecture also omits the fully connected final layers.

As mentioned in the last chapter, the training of deep networks faces the notorious vanishing gradient challenge: as the gradient propagates to earlier layers, repeated multiplication of small weights risks shrinking the gradient toward zero. Hence, increasing depth may limit learning.

The shortcut connection that skips two or more layers has become one of the most popular developments in CNN architectures and triggered numerous research efforts to refine and explain its performance. See the references on GitHub for additional information.

Transfer learning with VGG16 in practice

Modern CNNs can take weeks to train on multiple GPUs on ImageNet, but fortunately, many researchers share their final weights. TensorFlow 2, for example, contains pretrained models for several of the reference architectures discussed previously, namely VGG16 and its larger version, VGG19, ResNet50, InceptionV3, and InceptionResNetV2, as well as MobileNet, DenseNet, NASNet, and MobileNetV2.

How to extract bottleneck features

The notebook `bottleneck_features.ipynb` illustrates how to download the pretrained VGG16 model, either with the final layers to generate predictions or without the final layers, as illustrated in *Figure 18.8*, to extract the outputs produced by the bottleneck features:

Transfer Learning with the VGG Architecture

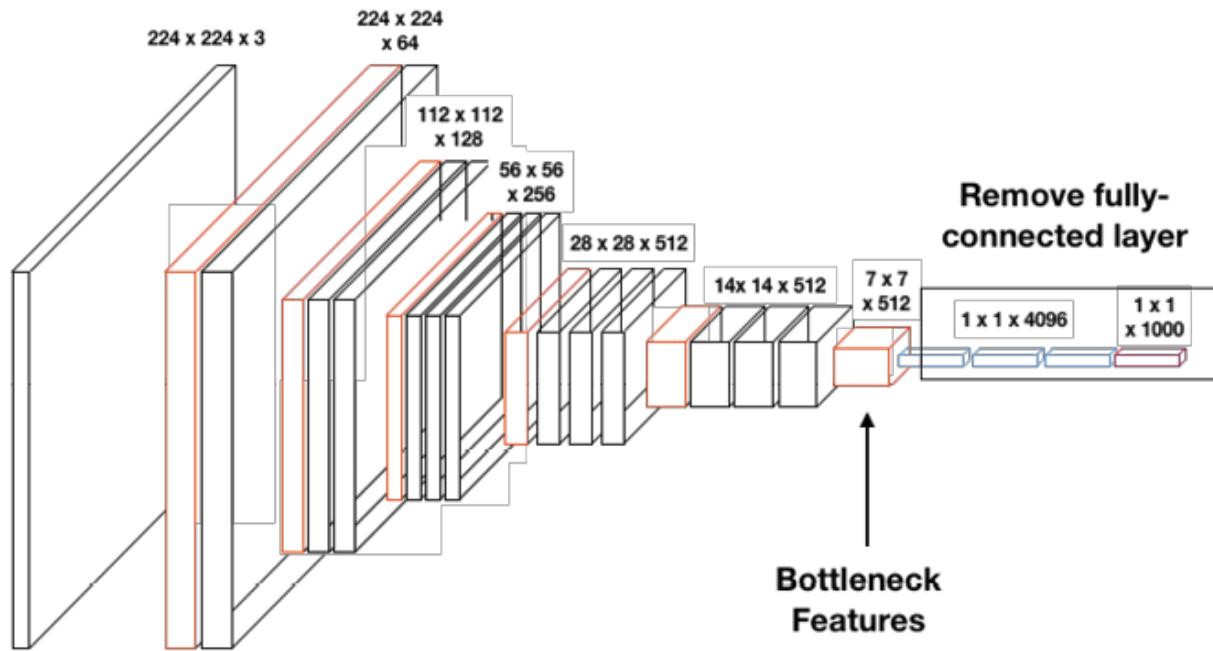


Figure 18.8: The VGG16 architecture

TensorFlow 2 makes it very straightforward to download and use pretrained models:

```
from tensorflow.keras.applications.vgg16 import VGG16
vgg16 = VGG16()
vgg16.summary()
Layer (type)          Output Shape         Param #
input_1 (InputLayer)  (None, 224, 224, 3)  0
... several layers omitted...
block5_conv4 (Conv2D)  (None, 14, 14, 512)  2359808
block5_pool (MaxPooling2D)  (None, 7, 7, 512)  0
flatten (Flatten)      (None, 25088)        0
fc1 (Dense)            (None, 4096)         102764544
fc2 (Dense)            (None, 4096)         16781312
predictions (Dense)    (None, 1000)          4097000
Total params: 138,357,544
Trainable params: 138,357,544
```

You can use this model for predictions like any other Keras model: we pass in seven sample images and obtain class probabilities for each of the 1,000 ImageNet categories:

```
y_pred = vgg16.predict(img_input)
Y_pred.shape
(7, 1000)
```

To exclude the fully connected layers, just add the keyword `include_top=False`. Predictions are now output by the final convolutional layer `block5_pool` and match this layer's shape:

```
vgg16 = VGG16(include_top=False)
vgg16.predict(img_input).shape
(7, 7, 7, 512)
```

By omitting the fully connected layers and keeping only the convolutional modules, we are no longer forced to use a fixed input size for the model such as the original 224×224 ImageNet format. Instead, we can adapt the model to arbitrary input sizes.

How to fine-tune a pretrained model

We will demonstrate how to freeze some or all of the layers of a pretrained model and continue training using a new fully-connected set of layers and data with a different format (see the notebook `transfer_learning.ipynb` for code examples, adapted from a TensorFlow 2 tutorial).

We use the VGG16 weights, pretrained on ImageNet with TensorFlow's built-in cats versus dogs images (see the notebook on how to source the dataset).

Preprocessing resizes all images to 160×160 pixels. We indicate the new input size as we instantiate the pretrained VGG16 instance and then freeze all weights:

```
vgg16 = VGG16(input_shape=IMG_SHAPE, include_top=False, weights='imagenet')
vgg16.trainable = False
vgg16.summary()
Layer (type)                  Output Shape                 Param #
... omitted layers...
block5_conv3 (Conv2D)          (None, 10, 10, 512)        2359808
block5_pool (MaxPooling2D)     (None, 5, 5, 512)           0
Total params: 14,714,688
Trainable params: 0
Non-trainable params: 14,714,688
```

The shape of the model output for 32 sample images now matches that of the last convolutional layer in the headless model:

```
feature_batch = vgg16(image_batch)
Feature_batch.shape
TensorShape([32, 5, 5, 512])
```

We can append new layers to the headless model using either the Sequential or the Functional API. For the Sequential API, adding `GlobalAveragePooling2D`, `Dense`, and `Dropout` layers works as follows:

We set `from_logits=True` for the `BinaryCrossentropy` loss because the model provides a linear output. The summary shows how the new model combines the pretrained VGG16 convolutional layers and the new final layers:

```
seq_model.summary()
Layer (type)          Output Shape         Param #
vgg16 (Model)        (None, 5, 5, 512)      14714688
global_average_pooling2d (Gl) (None, 512)      0
dense_7 (Dense)       (None, 64)            32832
dropout_3 (Dropout)   (None, 64)            0
dense_8 (Dense)       (None, 1)             65
Total params: 14,747,585
Trainable params: 11,831,937
Non-trainable params: 2,915,648
```

See the notebook for the Functional API version.

Prior to training the new final layer, the pretrained VGG16 delivers a validation accuracy of 48.75 percent. Now we proceed to train the model for 10 epochs as follows, adjusting only the final layer weights:

```
history = transfer_model.fit(train_batches,
                             epochs=initial_epochs,
                             validation_data=validation_batches)
```

10 epochs boost validation accuracy above 94 percent. To fine-tune the model, we can unfreeze the VGG16 models and continue training. Note that you should only do so after training the new final layers: randomly initialized classification layers will likely produce large gradient updates that can eliminate the pretraining results.

To unfreeze parts of the model, we select a layer, after which we set the weights to `trainable`; in this case, layer 12 of the total 19 layers in the VGG16 architecture:

```
vgg16.trainable = True
len(vgg16.layers)
19
# Fine-tune from this Layer onward
start_fine_tuning_at = 12
# Freeze all the layers before the 'fine_tune_at' layer
for layer in vgg16.layers[:start_fine_tuning_at]:
    layer.trainable = False
```

Now just recompile the model and continue training for up to 50 epochs using early stopping, starting in epoch 10 as follows:

```
fine_tune_epochs = 50
total_epochs = initial_epochs + fine_tune_epochs
history_fine_tune = transfer_model.fit(train_batches,
                                         epochs=total_epochs,
                                         initial_epoch=history.epoch[-1],
                                         validation_data=validation_batches,
                                         callbacks=[early_stopping])
```

Figure 18.9 shows how the validation accuracy increases substantially, reaching 97.89 percent after another 22 epochs:

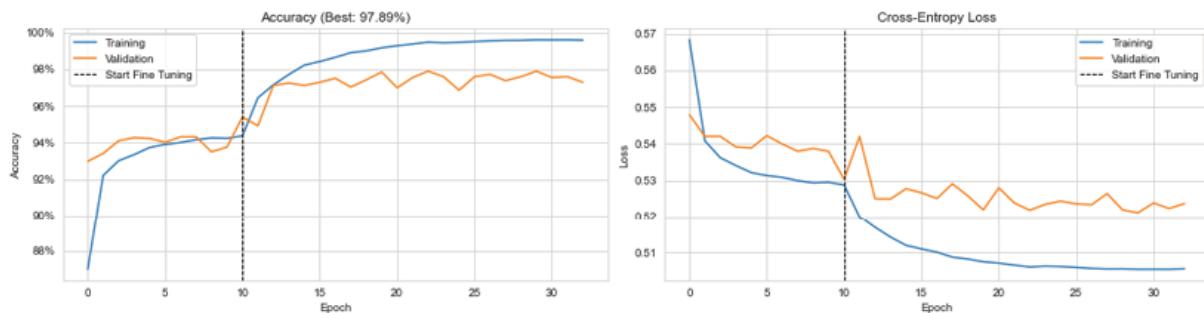


Figure 18.9: Cross-validation performance: accuracy and cross-entropy loss

Transfer learning is an important technique when training data is limited as is very often the case in practice. While cats and dogs are unlikely to produce tradeable signals, transfer learning could certainly help improve the accuracy of predictions on a relevant alternative dataset, such as the satellite images that we'll tackle next.

Classifying satellite images with transfer learning

Satellite images figure prominently among alternative data (see *Chapter 3, Alternative Data for Finance – Categories and Use Cases*). For instance, commodity traders may rely on satellite images to predict the supply of certain crops or resources by monitoring activity on farms, at mining sites, or oil tanker traffic.

The EuroSat dataset

To illustrate working with this type of data, we load the EuroSat dataset included in the TensorFlow 2 datasets (Helber et al. 2019). The EuroSat dataset includes around 27,000 images in 64×64 format that represent 10 different types of land uses. *Figure 18.10* displays an example for each label:



Figure 18.10: Ten types of land use contained in the dataset

A time series of similar data could be used to track the relative sizes of cultivated, industrial, and residential areas or the status of specific crops to predict harvest quantities or quality, for example, for wine.

Fine-tuning a very deep CNN – DenseNet201

Huang et al. (2018) developed a new architecture dubbed **densely connected** based on the insight that CNNs can be deeper, more accurate, and more efficient to train if they contain shorter connections between layers close to the input and those close to the output.

One architecture, labeled **DenseNet201**, connects each layer to every other layer in a feedforward fashion. It uses the feature maps of all preceding layers as inputs, while each layer's own feature maps become inputs into all subsequent layers.

We download the DenseNet201 architecture from

`tensorflow.keras.applications` and replace its final layers with the following dense layers interspersed with batch normalization to mitigate exploding or vanishing gradients in this very deep network with over 700 layers:

Layer (type)	Output Shape	Param #
densenet201 (Model)	(None, 1920)	18321984
batch_normalization (BatchNo)	(None, 1920)	7680
dense (Dense)	(None, 2048)	3934208
batch_normalization_1 (Batch	(None, 2048)	8192
dense_1 (Dense)	(None, 2048)	4196352
batch_normalization_2 (Batch	(None, 2048)	8192
dense_2 (Dense)	(None, 2048)	4196352
batch_normalization_3 (Batch	(None, 2048)	8192
dense_3 (Dense)	(None, 2048)	4196352
batch_normalization_4 (Batch	(None, 2048)	8192
dense_4 (Dense)	(None, 10)	20490

```
Total params: 34,906,186
Trainable params: 34,656,906
Non-trainable params: 249,280
```

Model training and results evaluation

We use 10 percent of the training images for validation purposes and achieve the best out-of-sample classification accuracy of 97.96 percent after 10 epochs. This exceeds the performance cited in the original paper for the best-performing ResNet-50 architecture with a 90-10 split.

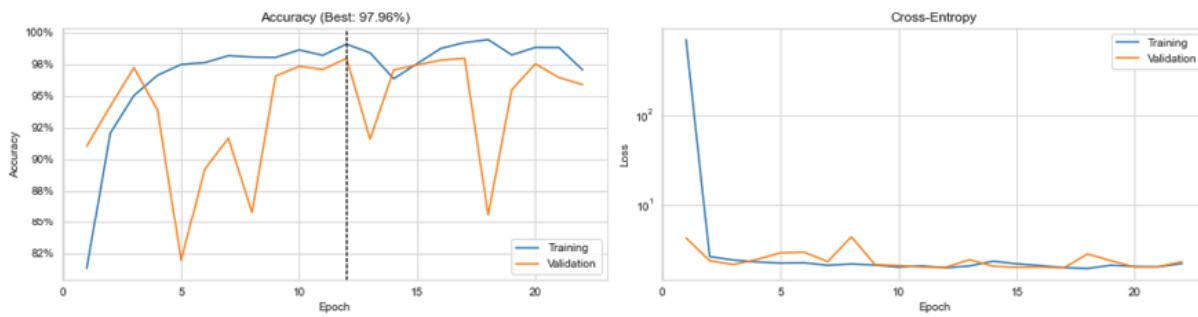


Figure 18.11: Cross-validation performance

There would likely be additional performance gains from augmenting the relatively small training set.

Object detection and segmentation

Image classification is a fundamental computer vision task that requires labeling an image based on certain objects it contains. Many practical applications, including investment and trading strategies, require additional information:

- The **object detection** task requires not only the identification but also the spatial location of all objects of interest, typically using

bounding boxes. Several algorithms have been developed to overcome the inefficiency of brute-force sliding-window approaches, including region proposal methods (R-CNN; see for example Ren et al. 2015) and the **You Only Look Once (YOLO)** real-time object detection algorithm (Redmon 2016).

- The **object segmentation** task goes a step further and requires a class label and an outline of every object in the input image. This may be useful to count objects such as oil tankers, individuals, or cars in an image and evaluate a level of activity.
- **Semantic segmentation**, also called scene parsing, makes dense predictions to assign a class label to each pixel in the image. As a result, the image is divided into semantic regions and each pixel is assigned to its enclosing object or region.

Object detection requires the ability to distinguish between several classes of objects and to decide how many and which of these objects are present in an image.

Object detection in practice

A prominent example is Ian Goodfellow's identification of house numbers from Google's **Street View House Numbers (SVHN)** dataset (Goodfellow 2014). It requires the model to identify the following:

- How many of up to five digits make up the house number
- The correct digit for each component
- The proper order of the constituent digits

We will show how to preprocess the irregularly shaped source images, adapt the VGG16 architecture to produce multiple outputs,

and train the final layer, before fine-tuning the pretrained weights to address the task.

Preprocessing the source images

The notebook `svhn_preprocessing.ipynb` contains code to produce a simplified, cropped dataset that uses bounding box information to create regularly shaped 32×32 images containing the digits; the original images are of arbitrary shape (Netzer 2011).



Figure 18.12: Cropped sample images of the SVHN dataset

The SVHN dataset contains house numbers with up to five digits and uses the class 10 if a digit is not present. However, since there are very few examples with five digits, we limit the images to those including up to four digits only.

Transfer learning with a custom final layer

The notebook `svhn_object_detection.ipynb` illustrates how to apply transfer learning to a deep CNN based on the VGG16 architecture, as outlined in the previous section. We will describe how to create new final layers that produce several outputs to meet the three SVHN task objectives, including one prediction of how many digits are present, and one for the value of each digit in the order they appear.

The best-performing architecture on the original dataset has eight convolutional layers and two final fully connected layers. We will use **transfer learning**, departing from the VGG16 architecture. As before, we import the VGG16 network pretrained on ImageNet weights, remove the layers after the convolutional blocks, freeze the weights, and create new dense and predictive layers as follows using the Functional API:

```
vgg16 = VGG16(input_shape=IMG_SHAPE, include_top=False, weights='imagenet')
vgg16.trainable = False
x = vgg16.output
x = Flatten()(x)
x = BatchNormalization()(x)
x = Dense(256)(x)
x = BatchNormalization()(x)
x = Activation('relu')(x)
x = Dense(128)(x)
x = BatchNormalization()(x)
x = Activation('relu')(x)
n_digits = Dense(SEQ_LENGTH, activation='softmax', name='n_digits')(x)
digit1 = Dense(N_CLASSES-1, activation='softmax', name='d1')(x)
digit2 = Dense(N_CLASSES, activation='softmax', name='d2')(x)
digit3 = Dense(N_CLASSES, activation='softmax', name='d3')(x)
digit4 = Dense(N_CLASSES, activation='softmax', name='d4')(x)
predictions = Concatenate()([n_digits, digit1, digit2, digit3, digit4]
```

The prediction layer combines the four-class output for the number of digits `n_digits` with four outputs that predict which digit is present at that position.

Creating a custom loss function and evaluation metrics

The custom output requires us to define a loss function that captures how well the model is meeting its objective. We would also like to measure accuracy in a way that reflects predictive accuracy tailored to the specific labels.

For the custom loss, we average the cross-entropy over the five categorical outputs, namely the number of digits and their respective values:

```
def weighted_entropy(y_true, y_pred):
    cce = tf.keras.losses.SparseCategoricalCrossentropy()
    n_digits = y_pred[:, :SEQ_LENGTH]
    digits = {}
    for digit, (start, end) in digit_pos.items():
        digits[digit] = y_pred[:, start:end]
    return (cce(y_true[:, 0], n_digits) +
            cce(y_true[:, 1], digits[1]) +
            cce(y_true[:, 2], digits[2]) +
            cce(y_true[:, 3], digits[3]) +
            cce(y_true[:, 4], digits[4])) / 5
```

To measure predictive accuracy, we compare the five predictions with the corresponding label values and average the share of correct matches over the batch of samples:

```
def weighted_accuracy(y_true, y_pred):
    n_digits_pred = K.argmax(y_pred[:, :SEQ_LENGTH], axis=1)
    digit_preds = {}
    for digit, (start, end) in digit_pos.items():
        digit_preds[digit] = K.argmax(y_pred[:, start:end], axis=1)
    preds = tf.dtypes.cast(tf.stack((n_digits_pred,
                                      digit_preds[1],
                                      digit_preds[2],
                                      digit_preds[3],
                                      digit_preds[4])), axis=1), tf.float32
    return K.mean(K.sum(tf.dtypes.cast(K.equal(y_true, preds), tf.int32), axis=1) / 5)
```

Finally, we integrate the base and final layers and compile the model with the custom loss and accuracy metric as follows:

```
model = Model(inputs=vgg16.input, outputs=predictions)
model.compile(optimizer='adam',
              loss=weighted_entropy,
              metrics=[weighted_accuracy])
```

Fine-tuning the VGG16 weights and final layer

We train the new final layers for 14 periods and continue fine-tuning all VGG16 weights, as in the previous section, for another 23 epochs (using early stopping in both cases).

The following charts show the training and validation accuracy and the loss over the entire training period. As we unfreeze the VGG16 weights after the initial training period, the accuracy drops and then improves, achieving a validation performance of 94.52 percent:

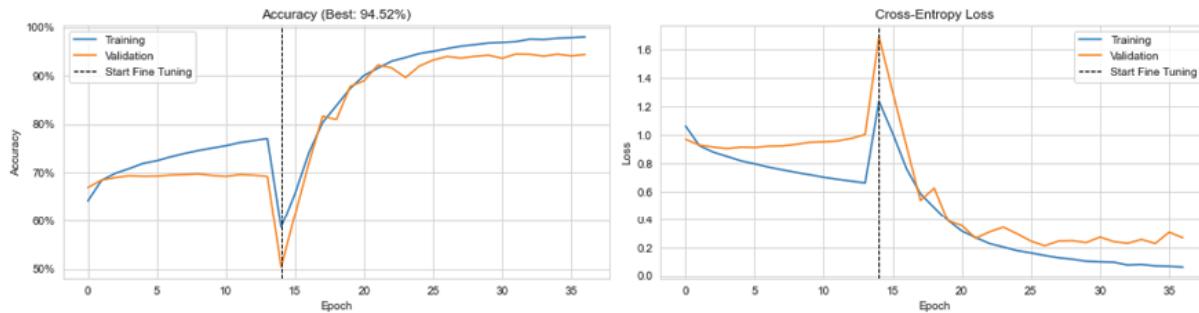


Figure 18.13: Cross-validation performance

See the notebook for additional implementation details and an evaluation of the results.

Lessons learned

We can achieve decent levels of accuracy using only the small training set. However, state-of-the-art performance achieves an error rate of only 1.02 percent (<https://benchmarks.ai/svhn>). To get closer, the most important step is to increase the amount of training data.

There are two easy ways to accomplish this: we can include the larger number of samples included in the **extra** dataset, and we can use image augmentation (see the *AlexNet: reigniting deep learning research* section). The currently best-performing approach relies heavily on augmentation learned from data (Cubuk 2019).

CNNs for time-series data – predicting returns

CNNs were originally developed to process image data and have achieved superhuman performance on various computer vision tasks. As discussed in the first section, time-series data has a grid-like structure similar to that of images, and CNNs have been successfully applied to one-, two- and three-dimensional representations of temporal data.

The application of CNNs to time series will most likely bear fruit if the data meets the model's key assumption that local patterns or relationships help predict the outcome. In the time-series context, local patterns could be autocorrelation or similar non-linear relationships at relevant intervals. Along the second and third dimensions, local patterns imply systematic relationships among different components of a multivariate series or among these series for different tickers. Since locality matters, it is important that the data is organized accordingly, in contrast to feed-forward networks where shuffling the elements of any dimension does not negatively affect the learning process.

In this section, we provide a relatively simple example using a one-dimensional convolution to model an autoregressive process (see

Chapter 9, Time-Series Models for Volatility Forecasts and Statistical Arbitrage) that predicts future returns based on lagged returns. Then we replicate a recent research paper that achieved good results by formatting multivariate time-series data like images to predict returns. We will also develop and test a trading strategy based on the signals contained in the predictions.

An autoregressive CNN with 1D convolutions

We will introduce the time series use case for CNN using a univariate autoregressive asset return model. More specifically, the model receives the most recent 12 months of returns and uses a single layer of one-dimensional convolutions to predict the subsequent month.

The requisite steps are as follows:

1. Creating the rolling 12 months of lagged returns and corresponding outcomes
2. Defining the model architecture
3. Training the model and evaluating the results

In the following sections, we'll describe each step in turn; the notebook `time_series_prediction` contains the code samples for this section.

Preprocessing the data

First, we'll select the adjusted close price for all Quandl Wiki stocks since 2000 as follows:

```

prices = (pd.read_hdf('../data/assets.h5', 'quandl/wiki/prices')
          .adj_close
          .unstack().loc['2000':])
prices.info()
DatetimeIndex: 2896 entries, 2007-01-01 to 2018-03-27
Columns: 3199 entries, A to ZUMZ

```

Next, we resample the price data to month-end frequency, compute returns, and set monthly returns over 100 percent to missing as they likely represent data errors. Then we drop tickers with missing observations, retaining 1,511 stocks with 215 observations each:

```

returns = (prices
           .resample('M')
           .last()
           .pct_change()
           .dropna(how='all')
           .loc['2000': '2017']
           .dropna(axis=1)
           .sort_index(ascending=False))
# remove outliers likely representing data errors
returns = returns.where(returns<1).dropna(axis=1)
returns.info()
DatetimeIndex: 215 entries, 2017-12-31 to 2000-02-29
Columns: 1511 entries, A to ZQK

```

To create the rolling series of 12 lagged monthly returns with their corresponding outcome, we iterate over rolling 13-month slices and append the transpose of each slice to a list after assigning the outcome date to the index. After completing the loop, we concatenate the DataFrames in the list as follows:

```

n = len(returns)
nlags = 12
lags = list(range(1, nlags + 1))
cnn_data = []
for i in range(n-nlags-1):
    df = returns.iloc[i:i+nlags+1]           # select outcome and Lags
    date = df.index.max()                   # use outcome date
    . . .

```

```

cnn_data.append(dt.reset_index(drop=True) # append transposed ser
                .transpose()
                .assign(date=date)
                .set_index('date', append=True)
                .sort_index(1, ascending=True))
cnn_data = (pd.concat(cnn_data)
            .rename(columns={0: 'label'})
            .sort_index())

```

We end up with over 305,000 pairs of outcomes and lagged returns for the 2001-2017 period:

```

cnn_data.info(null_counts=True)
MultiIndex: 305222 entries, ('A', Timestamp('2001-03-31 00:00:00')) to
('ZQK', Timestamp('2017-12-31 00:00:00'))
Data columns (total 13 columns):
 ...

```

When we compute the information coefficient for each lagged return and the outcome, we find that only lag 5 is not statistically significant:

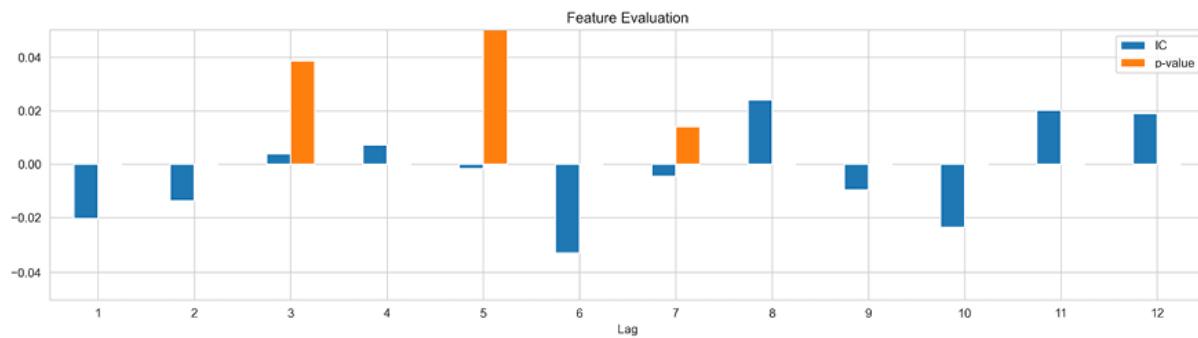


Figure 18.14: Information coefficient with respect to forward return by lag

Defining the model architecture

Now we'll define the model architecture using TensorFlow's Keras interface. We combine a one-dimensional convolutional layer with max pooling and batch normalization to produce a real-valued scalar output:

```
model = Sequential([Conv1D(filters=32,
                           kernel_size=4,
                           activation='relu',
                           padding='causal',
                           input_shape=(12, 1),
                           use_bias=True,
                           kernel_regularizer=regularizers.l1_l2(l1=1e-05,
                                                               l2=1e-05),
                           MaxPooling1D(pool_size=4),
                           Flatten(),
                           BatchNormalization(),
                           Dense(1, activation='linear')])
```

The one-dimensional convolution computes the sliding dot product of a (regularized) vector of length 4 with each input sequence of length 12, using causal padding to maintain the temporal order (see the *How to scan the input: strides and padding* section). The resulting 32 feature maps have the same length, 12, as the input that max pooling in groups of size 4 reduces to 32 vectors of length 3.

The model outputs the weighted average plus the bias of the flattened and normalized single vector of length 96, and has 449 trainable parameters:

Layer (type)	Output Shape	Param #
conv1d (Conv1D)	(None, 12, 32)	160
max_pooling1d (MaxPooling1D)	(None, 3, 32)	0
flatten (Flatten)	(None, 96)	0
batch_normalization (BatchNormalization)	(None, 96)	384
dense (Dense)	(None, 1)	97
Total params:	641	
Trainable params:	449	
Non-trainable params:	192	

The notebook wraps the model generation and subsequent compilation into a `get_model()` function that parametrizes the model configuration to facilitate experimentation.

Model training and performance evaluation

We train the model on five years of data for each ticker to predict the first month after this period and repeat this procedure 36 times using the `MultipleTimeSeriesCV` we developed in *Chapter 7, Linear Models – From Risk Factors to Return Forecasts*. See the notebook for the training loop that follows the pattern demonstrated in the previous chapter.

We use early stopping after five epochs to simplify the exposition, resulting in a positive bias so that the results have only illustrative character. Training length varies from 1 to 27 epochs, with a median of 5 epochs, which demonstrates that the model can often only learn very limited amounts of systematic information from the past returns. Thus cherry-picking the results yields a cumulative average information coefficient of around 4, as shown in *Figure 18.15*:

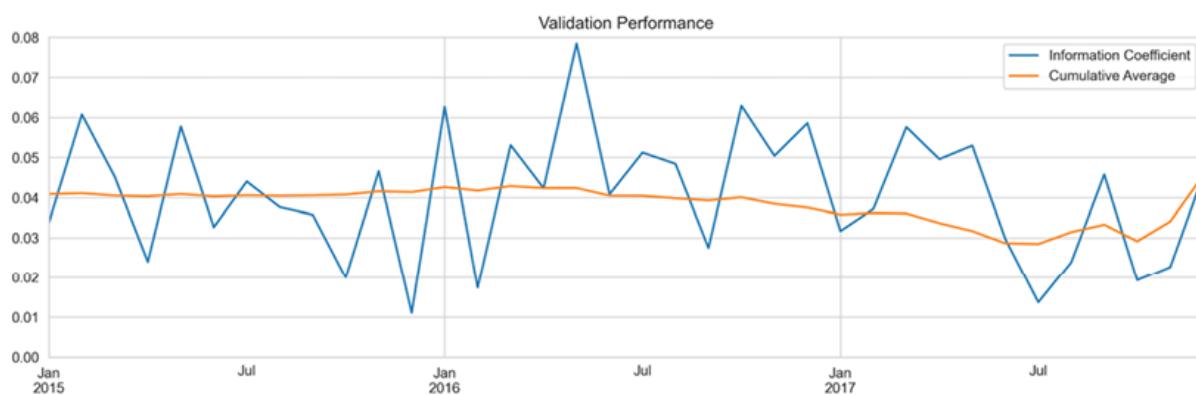


Figure 18.15: (Biased) out-of-sample information coefficients for best epochs

We'll now proceed to a more complex example of using CNNs for multiple time-series data.

CNN-TA – clustering time series in 2D format

To exploit the grid-like structure of time-series data, we can use CNN architectures for univariate and multivariate time series. In the latter case, we consider different time series as channels, similar to the different color signals.

An alternative approach converts a time series of alpha factors into a two-dimensional format to leverage the ability of CNNs to detect local patterns. Sezer and Ozbayoglu (2018) propose **CNN-TA**, which computes 15 technical indicators for different intervals and uses hierarchical clustering (see *Chapter 13, Data-Driven Risk Factors and Asset Allocation with Unsupervised Learning*) to locate indicators that behave similarly close to each other in a two-dimensional grid.

The authors train a CNN similar to the CIFAR-10 example we used earlier to predict whether to buy, hold, or sell an asset on a given day. They compare the CNN performance to "buy-and-hold" and other models and find that it outperforms all alternatives using daily price series for Dow 30 stocks and the nine most-traded ETFs over the 2007-2017 time period.

In this section, we experiment with this approach using daily US equity price data and demonstrate how to compute and convert a similar set of indicators into image format. Then we train a CNN to predict daily returns and evaluate a simple long-short strategy based on the resulting signals.

Creating technical indicators at different intervals

We first select a universe of the 500 most-traded US stocks from the Quandl Wiki dataset by dollar volume for rolling five-year periods for 2007-2017. See the notebook `engineer_cnn_features.ipynb` for the code examples in this section and some additional implementation details.

Our features consist of 15 technical indicators and risk factors that we compute for 15 different intervals and then arrange them in a 15×15 grid. The following table lists some of the technical indicators; in addition, we follow the authors in using the following metrics (see the *Appendix* for additional information):

- **Weighted and exponential moving averages (WMA and EMA)** of the close price
- **Rate of change (ROC)** of the close price
- **Chande Momentum Oscillator (CMO)**
- **Chaikin A/D Oscillators (ADOSC)**
- **Average Directional Movement Index (ADX)**

Indicator	Name	Formula
Relative Strength Index (RSI)	Oscillates in $[0, 100]$ range; below 30: oversold, over 70: overbought	See Chapter 4
Williams %R	Momentum-based in $[-100, 0]$ range, below -80: oversold, above -20: overbought	$R = \frac{\max(\text{high}) - \text{close}}{\max(\text{high}) - \min(\text{low})}$
Bollinger Bands	20-day moving average plus/minus daily standard deviation; prices above/below these bands indicate overbought/sold	See chapter 4.
Normalized Average True Range (NATR)	Avg. true range: max of current high-low, current high-prev.close or absolute of prev. close - current low, averaged over t days.	$\text{NATR} = \frac{\text{ATR}(t)}{\text{Close}}$
Percentage Price Oscillator (PPO)	Momentum: compares two exponential moving averages (EMA) in percentage terms	$\text{PPO} = \frac{\text{EMA}_{12} - \text{EMA}_{26}}{\text{EMA}_{26}}$
Commodity Channel Index (CCI)	Momentum-based: difference between current and simple moving average (SMA) of the historical average price, normalized by their mean difference	$\rho^{\text{hist}} = \sum_{t=1}^P (\text{high} + \text{low} + \text{close})/3$ $\text{CCI} = \frac{\rho^{\text{hist}} - \text{SMA}(\rho^{\text{hist}})}{0.15 \times \sqrt{\sum_{t=1}^P (\rho^{\text{hist}} - \text{SMA}(\rho^{\text{hist}}))^2 / P}}$

Figure 8.16: Technical indicators

For each indicator, we vary the time period from 6 to 20 to obtain 15 distinct measurements. For example, the following code example computes the **relative strength index (RSI)**:

```
T = list(range(6, 21))
for t in T:
    universe[f'{t:02}_RSI'] = universe.groupby(level='symbol').close.ewm(span=t).mean() - universe.groupby(level='symbol').close.ewm(span=t-14).mean()
    universe[f'{t:02}_RSI'] = 100 - 100 / (1 + universe[f'{t:02}_RSI'])
```

For the **Normalized Average True Range (NATR)** that requires several inputs, the computation works as follows:

```
for t in T:
    universe[f'{t:02}_NATR'] = universe.groupby(
        level='symbol', group_keys=False).apply(
            lambda x: NATR(x.high, x.low, x.close, timeperiod=t))
```

See the TA-Lib documentation for further details.

Computing rolling factor betas for different horizons

We also use **five Fama-French risk factors** (Fama and French, 2015; see *Chapter 4, Financial Feature Engineering – How to Research Alpha Factors*). They reflect the sensitivity of a stock's returns to factors consistently demonstrated to impact equity returns. We capture these factors by computing the coefficients of a rolling OLS regression of a stock's daily returns on the returns of portfolios designed to reflect the underlying drivers:

- **Equity risk premium:** Value-weighted returns of US stocks minus the 1-month US Treasury bill rate
- **Size (SMB):** Returns of stocks categorized as **Small** (by market cap) **Minus** those of **Big equities**

- **Value (HML)**: Returns of stocks with **High** book-to-market value **Minus** those with a **Low value**
- **Investment (CMA)**: Returns differences for companies with **Conservative** investment expenditures **Minus** those with **Aggressive spending**
- **Profitability (RMW)**: Similarly, return differences for stocks with **Robust** profitability **Minus** that with a **Weak** metric.

We source the data from Kenneth French's data library using `pandas_datareader` (see *Chapter 4, Financial Feature Engineering – How to Research Alpha Factors*):

```
import pandas_datareader.data as web
factor_data = (web.DataReader('F-F_Research_Data_5_Factors_2x3_daily',
                             'famafrench', start=START)[0])
```

Next, we apply `statsmodels`' `RollingOLS()` to run regressions over windowed periods of different lengths, ranging from 15 to 90 days. We set the `params_only` parameter on the `.fit()` method to speed up computation and capture the coefficients using the `.params` attribute of the fitted `factor_model`:

```
factors = ['Mkt-RF', 'SMB', 'HML', 'RMW', 'CMA']
windows = list(range(15, 90, 5))
for window in windows:
    betas = []
    for symbol, data in universe.groupby(level='symbol'):
        model_data = data[[ret]].merge(factor_data, on='date').dropna()
        model_data[ret] -= model_data.RF
        rolling_ols = RollingOLS(endog=model_data[ret],
                                exog=sm.add_constant(model_data[factors]),
                                window=window)
        factor_model = rolling_ols.fit(params_only=True).params.drop('Intercept')
        result = factor_model.assign(symbol=symbol).set_index('symbol')
```

```
append=1
    betas.append(result)
betas = pd.concat(betas).rename(columns=lambda x: f'{window:02}_{x}')
universe = universe.join(betas)
```

Features selecting based on mutual information

The next step is to select the 15 most relevant features from the 20 candidates to fill the 15×15 input grid. The code examples for the following steps are in the notebook

```
convert_cnn_features_to_image_format .
```

To this end, we estimate the mutual information for each indicator and the 15 intervals with respect to our target, the one-day forward returns. As discussed in *Chapter 4, Financial Feature Engineering – How to Research Alpha Factors*, scikit-learn provides the

`mutual_info_regression()` function that makes this straightforward, albeit time-consuming and memory-intensive. To accelerate the process, we randomly sample 100,000 observations:

```
df = features.join(targets[target]).dropna().sample(n=100000)
X = df.drop(target, axis=1)
y = df[target]
mi[t] = pd.Series(mutual_info_regression(X=X, y=y), index=X.columns)
```

The left panel in *Figure 18.16* shows the mutual information, averaged across the 15 intervals for each indicator. NATR, PPO, and Bollinger Bands are most important from this metric's perspective:

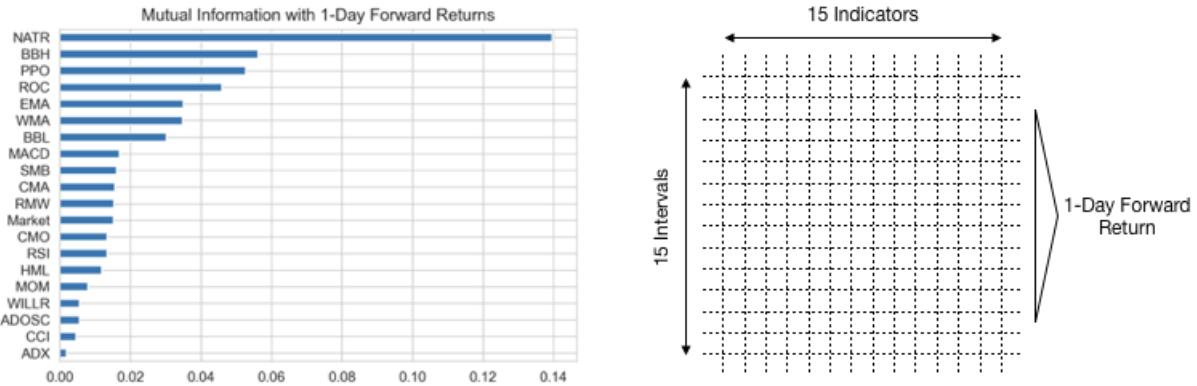


Figure 18.17: Mutual information and two-dimensional grid layout for time series

Hierarchical feature clustering

The right panel in *Figure 18.16* sketches the 15×15 two-dimensional feature grid that we will feed into our CNN. As discussed in the first section of this chapter, CNNs rely on the locality of relevant patterns that is typically found in images where nearby pixels are closely related and changes from one pixel to the next are often gradual.

To organize our indicators in a similar fashion, we will follow Sezer and Ozbayoglu's approach of applying hierarchical clustering. The goal is to identify features that behave similarly and order the columns and the rows of the grid accordingly.

We can build on SciPy's `pairwise_distance()`, `linkage()`, and `dendrogram()` functions that we introduced in *Chapter 13, Data-Driven Risk Factors and Asset Allocation with Unsupervised Learning* alongside other forms of clustering. We create a helper function that standardizes the input column-wise to avoid distorting distances among features due to differences in scale, and use the Ward criterion that merges clusters to minimize variance. The function returns the order of the leaf nodes in the dendrogram that in turn displays the successive formation of larger clusters:

```

def cluster_features(data, labels, ax, title):
    data = StandardScaler().fit_transform(data)
    pairwise_distance = pdist(data)
    Z = linkage(data, 'ward')
    dend = dendrogram(Z,
                       labels=labels,
                       orientation='top',
                       leaf_rotation=0.,
                       leaf_font_size=8.,
                       ax=ax)
    return dend['ivl']

```

To obtain the optimized order of technical indicators in the columns and the different intervals in the rows, we use NumPy's `.reshape()` method to ensure that the dimension we would like to cluster appears in the columns of the two-dimensional array we pass to

```
cluster_features():
```

```

labels = sorted(best_features)
col_order = cluster_features(features.dropna().values.reshape(-1, 15).
                             labels)
labels = list(range(1, 16))
row_order = cluster_features(
    features.dropna().values.reshape(-1, 15, 15).transpose((0, 2, 1)).

```

Figure 18.18 shows the dendograms for both the row and column features:

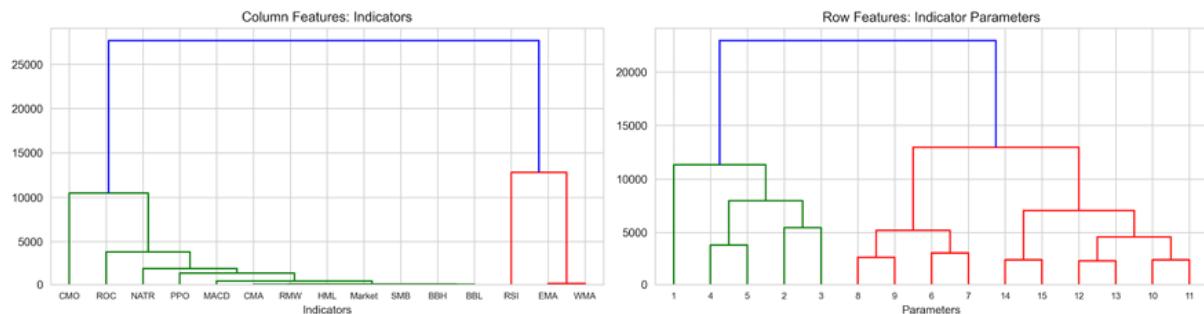


Figure 18.18: Dendograms for row and column features

We reorder the features accordingly and store the result as inputs for the CNN that we will create in the next step.

Creating and training a convolutional neural network

Now we are ready to design, train, and evaluate a CNN following the steps outlined in the previous section. The notebook

`cnn_for_trading.ipynb` contains the relevant code examples.

We again closely follow the authors in creating a CNN with 2 convolutional layers with kernel size 3 and 16 and 32 filters, respectively, followed by a max pooling layer of size 2. We flatten the output of the last stack of filters and connect the resulting 1,568 outputs to a dense layer of size 32, applying 25 and 50 percent dropout probability to the incoming and outgoing connections to mitigate overfitting. The following table summarizes the CNN structure that contains 55,041 trainable parameters:

Layer (type)	Output Shape	Param #
CONV1 (Conv2D)	(None, 15, 15, 16)	160
CONV2 (Conv2D)	(None, 15, 15, 32)	4640
POOL1 (MaxPooling2D)	(None, 7, 7, 32)	0
DROP1 (Dropout)	(None, 7, 7, 32)	0
FLAT1 (Flatten)	(None, 1568)	0
FC1 (Dense)	(None, 32)	50208
DROP2 (Dropout)	(None, 32)	0
FC2 (Dense)	(None, 1)	33
Total params:	55,041	
Trainable params:	55,041	
Non-trainable params:	0	

We cross-validate the model with the `MultipleTimeSeriesCV` train and validation set index generator introduced in *Chapter 7, Linear Models – From Risk Factors to Return Forecasts*. We provide 5 years of trading days during the training period in batches of 64 random samples

and validate using the subsequent 3 months, covering the years 2014-2017.

We scale the features to the range $[-1, 1]$ and again use NumPy's `.reshape()` method to create the requisite $N \times 15 \times 15 \times 1$ format:

```
def get_train_valid_data(X, y, train_idx, test_idx):
    x_train, y_train = X.iloc[train_idx, :], y.iloc[train_idx]
    x_val, y_val = X.iloc[test_idx, :], y.iloc[test_idx]
    scaler = MinMaxScaler(feature_range=(-1, 1))
    x_train = scaler.fit_transform(x_train)
    x_val = scaler.transform(x_val)
    return (x_train.reshape(-1, size, size, 1), y_train,
            x_val.reshape(-1, size, size, 1), y_val)
```

Training and validation follow the process laid out in *Chapter 17, Deep Learning for Trading*, relying on checkpointing to store weights after each epoch and generate predictions for the best-performing iterations without the need for costly retraining.

To evaluate the model's predictive accuracy, we compute the daily **information coefficient** (IC) for the validation set like so:

```
checkpoint_path = Path('models', 'cnn_ts')
for fold, (train_idx, test_idx) in enumerate(cv.split(features)):
    X_train, y_train, X_val, y_val = get_train_valid_data(features, train_idx, test_idx)
    preds = y_val.to_frame('actual')
    r = pd.DataFrame(index=y_val.index.unique(level='date')).sort_index()
    model = make_model(filter1=16, act1='relu', filter2=32,
                        act2='relu', do1=.25, do2=.5, dense=32)
    for epoch in range(n_epochs):
        model.fit(X_train, y_train,
                  batch_size=batch_size,
                  validation_data=(X_val, y_val),
                  epochs=1, verbose=0, shuffle=True)
    model.save_weights(
        (checkpoint_path / f'ckpt_{fold}_{epoch}').as_posix())
    preds[epoch] = model.predict(X_val).squeeze()
    r[epoch] = preds.groupby(level='date').apply(
        lambda x: spearmanr(x.actual, x[epoch])[0]).to_frame(epoch)
```

We train the model for up to 10 epochs using **stochastic gradient descent** with **Nesterov** momentum (see *Chapter 17, Deep Learning for Trading*) and find that the best performing epochs, 8 and 9, achieve a (low) daily average IC of around 0.009.

Assembling the best models to generate tradeable signals

To reduce the variance of the test-period forecasts, we generate and average the predictions for the 3 models that perform best during cross-validation, which here correspond to training for 4, 8, and 9 epochs. As in the previous time-series example, the relatively short training period underscores that the amount of signals in financial time series is low compared to the systematic information contained in, for example, image data.

The `generate_predictions()` function reloads the model weights and returns the forecasts for the target period:

```
def generate_predictions(epoch):
    predictions = []
    for fold, (train_idx, test_idx) in enumerate(cv.split(features)):
        X_train, y_train, X_val, y_val = get_train_valid_data(
            features, target, train_idx, test_idx)
        preds = y_val.to_frame('actual')
        model = make_model(filter1=16, act1='relu', filter2=32,
                            act2='relu', do1=.25, do2=.5, dense=32)
        status = model.load_weights(
            (checkpoint_path / f'ckpt_{fold}_{epoch}').as_posix())
        status.expect_partial()
        predictions.append(pd.Series(model.predict(X_val).squeeze(),
                                      index=y_val.index))

    return pd.concat(predictions)
preds = {}
for i, epoch in enumerate(ic.drop('fold', axis=1).mean().nlargest(3).i
    preds[i] = generate_predictions(epoch)
```

We store the predictions and proceed to backtest a trading strategy based on these daily return forecasts.

Backtesting a long-short trading strategy

To get a sense of the signal quality, we compute the spread between equally weighted portfolios invested in stocks selected according to the signal quintiles using Alphalens (see *Chapter 4, Financial Feature Engineering – How to Research Alpha Factors*).

Figure 18.19 shows that for a one-day investment horizon, this naive strategy would have earned a bit over four basis points per day during the 2013-2017 period:

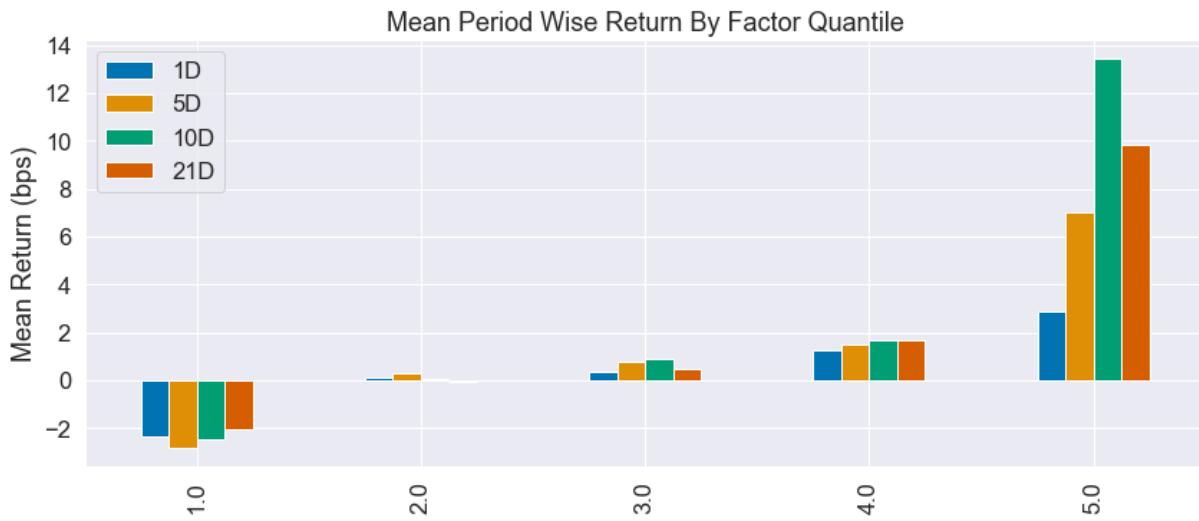


Figure 18.19: Alphalens signal quality evaluation

We translate this slightly encouraging result into a simple strategy that enters long (short) positions for the 25 stocks with the highest (lowest) return forecasts, trading on a daily basis. *Figure 18.20* shows that this strategy is competitive with the S&P 500 benchmark over much of the backtesting period (left panel), resulting in a 35.6

percent cumulative return and a Sharpe ratio of 0.53 (before transaction costs; right panel)

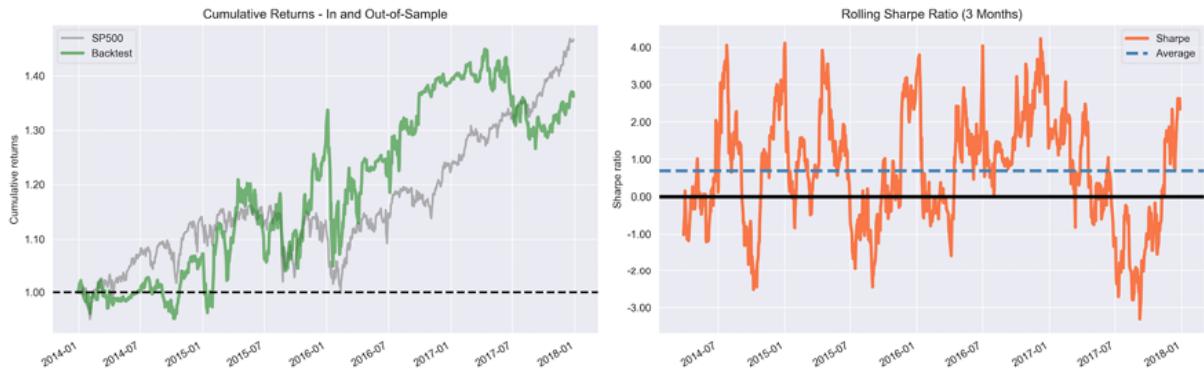


Figure 18.20: Backtest performance in- and out-of-sample

Summary and lessons learned

It appears that the CNN is able to extract meaningful information from the time series of alpha factors converted into a two-dimensional grid. Experimentation with different architectures and training parameters shows that the result is not very robust and slight modifications can yield significantly worse performance.

Tuning attempts also surface the notorious difficulties in successfully training a deep NN, especially when the signal-to-noise ratio is low: too complex a network or the wrong optimizer can lead the CNN to a local optimum where it always predicts a constant value.

The most important step to improve the results and obtain a performance closer to that achieved by the authors (using different outcomes) would be to revisit the features. There are many alternatives to different intervals of a limited set of technical indicators. Any appropriate number of time-series features could be arranged in a rectangular $n \times m$ format and benefit from the CNN's ability to learn local patterns. The choice of n indicators and m

intervals just makes it easier to organize the rows and the columns of the two-dimensional grid. Give it a shot!

Furthermore, the authors take a classification approach to the algorithmically labeled buy, hold, and sell outcomes (see the paper for an outline of the computation), whereas our experiment applied regression to the daily returns. The Alphalens chart in *Figure 18.18* suggests that longer holding periods (especially 10 days) might work better, so there is also scope for adjusting the strategy accordingly or switching to a classification approach.

Summary

In this chapter, we introduced CNNs, a specialized NN architecture that has taken cues from our (limited) understanding of human vision and performs particularly well on grid-like data. We covered the central operation of convolution or cross-correlation that drives the discovery of filters that in turn detect features useful to solve the task at hand.

We reviewed several state-of-the-art architectures that are good starting points, especially because transfer learning enables us to reuse pretrained weights and reduce the otherwise rather computationally and data-intensive training effort. We also saw that Keras makes it relatively straightforward to implement and train a diverse set of deep CNN architectures.

In the next chapter, we turn our attention to recurrent neural networks that are designed specifically for sequential data, such as time-series data, which is central to investment and trading.

RNNs for Multivariate Time Series and Sentiment Analysis

The previous chapter showed how **convolutional neural networks (CNNs)** are designed to learn features that represent the spatial structure of grid-like data, especially images, but also time series. This chapter introduces **recurrent neural networks (RNNs)** that specialize in sequential data where patterns evolve over time and learning typically requires memory of preceding data points.

Feedforward neural networks (FFNNs) treat the feature vectors for each sample as independent and identically distributed. Consequently, they do not take prior data points into account when evaluating the current observation. In other words, they have no memory.

The one- and two-dimensional convolutional filters used by CNNs can extract features that are a function of what is typically a small number of neighboring data points. However, they only allow shallow parameter-sharing: each output results from applying the same filter to the relevant time steps and features.

The major innovation of the RNN model is that each output is a function of both the previous output and new information. RNNs can thus incorporate information on prior observations into the computation they perform using the current feature vector. This

recurrent formulation enables parameter-sharing across a much deeper computational graph (Goodfellow, Bengio, and Courville, 2016). In this chapter, you will encounter **long short-term memory (LSTM)** units and **gated recurrent units (GRUs)**, which aim to overcome the challenge of vanishing gradients associated with learning long-range dependencies, where errors need to be propagated over many connections.

Successful RNN use cases include various tasks that require mapping one or more input sequences to one or more output sequences and prominently feature natural language applications. We will explore how RNNs can be applied to univariate and multivariate time series to predict asset prices using market or fundamental data. We will also cover how RNNs can leverage alternative text data using word embeddings, which we covered in *Chapter 16, Word Embeddings for Earnings Calls and SEC Filings*, to classify the sentiment expressed in documents. Finally, we will use the most informative sections of SEC filings to learn word embeddings and predict returns around filing dates.

More specifically, in this chapter, you will learn about the following:

- How recurrent connections allow RNNs to memorize patterns and model a hidden state
- Unrolling and analyzing the computational graph of RNNs
- How gated units learn to regulate RNN memory from data to enable long-range dependencies
- Designing and training RNNs for univariate and multivariate time series in Python
- How to learn word embeddings or use pretrained word vectors for sentiment analysis with RNNs

- Building a bidirectional RNN to predict stock returns using custom word embeddings

You can find the code examples and additional resources in the GitHub repository's directory for this chapter.

How recurrent neural nets work

RNNs assume that the input data has been generated as a sequence such that previous data points impact the current observation and are relevant for predicting subsequent values. Thus, they allow more complex input-output relationships than FFNNs and CNNs, which are designed to map one input vector to one output vector using a given number of computational steps. RNNs, in contrast, can model data for tasks where the input, the output, or both, are best represented as a sequence of vectors. For a good overview, refer to *Chapter 10* in Goodfellow, Bengio, and Courville (2016).

The diagram in *Figure 19.1*, inspired by Andrew Karpathy's 2015 blog post *The Unreasonable Effectiveness of Recurrent Neural Networks* (see GitHub for a link), illustrates mappings from input to output vectors using nonlinear transformations carried out by one or more neural network layers:

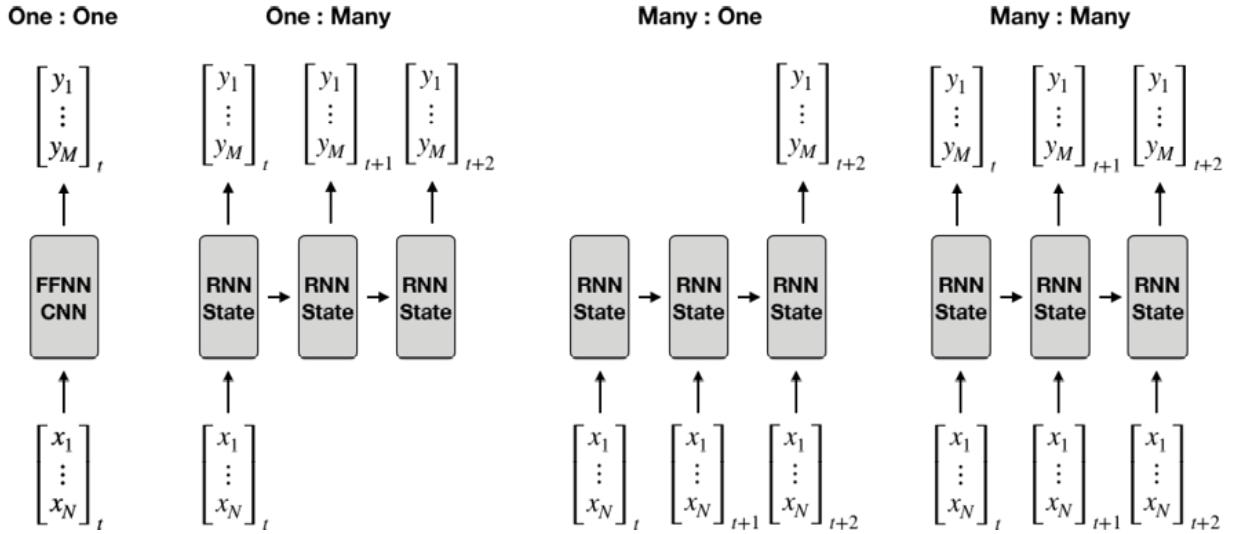


Figure 19.1: Various types of sequence-to-sequence models

The left panel shows a one-to-one mapping between vectors of fixed sizes, typical for FFNs and CNNs covered in the last two chapters. The other three panels show various RNN applications that map input vectors to output vectors by applying a recurrent transformation to the new input and the state produced by the previous iteration. The x input vectors to an RNN are also called **context**.

The vectors are time-indexed, as usually required by trading-related applications, but they could also be labeled by a different set of sequential values. Generic sequence-to-sequence mapping tasks and sample applications include:

- **One-to-many:** Image captioning, for example, takes a single vector of pixels (as in the previous chapter) and maps it to a sequence of words.
- **Many-to-one:** Sentiment analysis takes a sequence of words or tokens (see *Chapter 14, Text Data for Trading – Sentiment Analysis*) and maps it to an output scalar or vector.

- **Many-to-many:** Machine translation or labeling of video frame map sequences of input vectors to sequences of output vectors, either in a synchronized (as shown) or asynchronous fashion. Multistep prediction of multivariate time series also maps several input vectors to several output vectors.

Note that input and output sequences can be of arbitrary lengths because the recurrent transformation that is fixed but learned from the data can be applied as many times as needed.

Just as CNNs easily scale to large images and some CNNs can process images of variable size, RNNs scale to much longer sequences than networks not tailored to sequence-based tasks. Most RNNs can also process sequences of variable length.

Unfolding a computational graph with cycles

RNNs are called recurrent because they apply the same transformations to every element of a sequence in a way that the RNN's output depends on the outcomes of prior iterations. As a result, RNNs maintain an **internal state** that captures information about previous elements in the sequence, just like memory.

Figure 19.2 shows the **computational graph** implied by a single hidden RNN unit that learns two weight matrices during training:

- W_{hh} : applied to the previous hidden state, h_{t-1}
- W_{hx} : applied to the current input, x_t

The RNN's output, y_t , is a nonlinear transformation of the sum of the two matrix multiplications using, for example, the tanh or ReLU activation functions:

$$y_t = g(W_{hh}h_{t-1} + W_{xh}x_t)$$

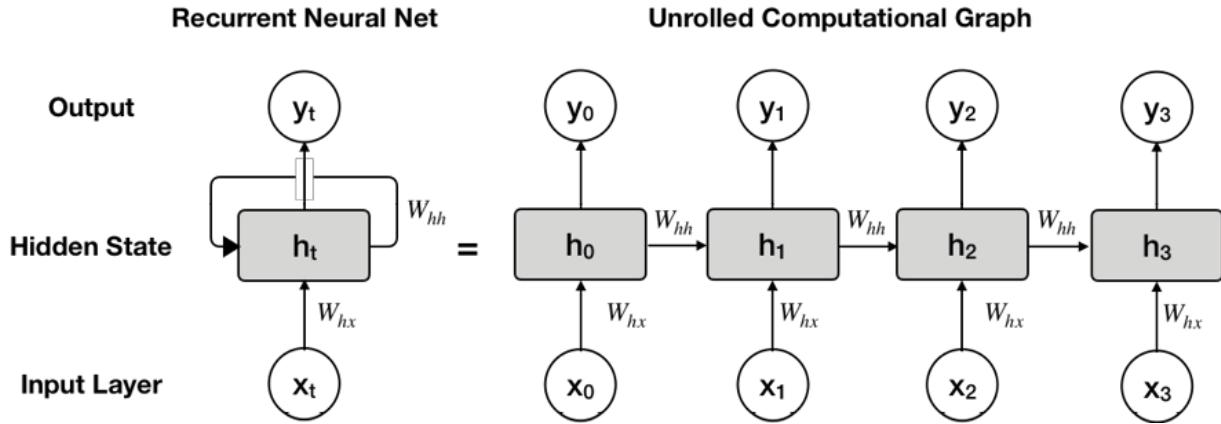


Figure 19.2: Recurrent and unrolled view of the computational graph of an RNN with a single hidden unit

The right side of the equation shows the effect of unrolling the recurrent relationship depicted in the right panel of the figure. It highlights the repeated linear algebra transformations and the resulting hidden state that combines information from past sequence elements with the current input, or context. An alternative formulation connects the context vector to the first hidden state only; we will outline additional options to modify this baseline architecture in the subsequent section.

Backpropagation through time

The unrolled computational graph in the preceding figure highlights that the learning process necessarily encompasses all time steps of the given input sequence. The backpropagation algorithm that updates the weights during training involves a forward pass from left to right along with the unrolled computational graph, followed by a backward pass in the opposite direction.

As discussed in *Chapter 17, Deep Learning for Trading*, the backpropagation algorithm evaluates a loss function and computes its gradient with respect to the parameters to update the weights accordingly. In the RNN context, backpropagation runs from right to left in the computational graph, updating the parameters from the final time step all the way to the initial time step. Therefore, the algorithm is called **backpropagation through time** (Werbos 1990).

It highlights both the power of an RNN to model long-range dependencies by sharing parameters across an arbitrary number of sequence elements while maintaining a corresponding state. On the other hand, it is computationally quite expensive, and the computations for each time step cannot be parallelized due to its inherently sequential nature.

Alternative RNN architectures

Just like the FFNN and CNN architectures we covered in the previous two chapters, RNNs can be optimized in a variety of ways to capture the dynamic relationship between input and output data.

In addition to modifying the recurrent connections between the hidden states, alternative approaches include recurrent output relationships, bidirectional RNNs, and encoder-decoder architectures. Refer to GitHub for background references to complement this brief summary.

Output recurrence and teacher forcing

One way to reduce the computational complexity of hidden state recurrences is to connect a unit's hidden state to the prior unit's output rather than its hidden state. The resulting RNN has a lower

capacity than the architecture discussed previously, but different time steps are now decoupled and can be trained in parallel.

However, to successfully learn relevant past information, the training output samples need to reflect this information so that backpropagation can adjust the network parameters accordingly. To the extent that asset returns are independent of their lagged values, financial data may not meet this requirement. The use of previous outcome values alongside the input vectors is called **teacher forcing** (Williams and Zipser, 1989).

Connections from the output to the subsequent hidden state can also be used in combination with hidden recurrence. However, training requires backpropagation through time and cannot be run in parallel.

Bidirectional RNNs

For some tasks, it can be realistic and beneficial for the output to depend not only on past sequence elements, but also on future elements (Schuster and Paliwal, 1997). Machine translation or speech and handwriting recognition are examples where subsequent sequence elements are both informative and realistically available to disambiguate competing outputs.

For a one-dimensional sequence, **bidirectional RNNs** combine an RNN that moves forward with another RNN that scans the sequence in the opposite direction. As a result, the output comes to depend on both the future and the past of the sequence. Applications in the natural language and music domains (Sigtia et al., 2014) have been very successful (see *Chapter 16, Word Embeddings for Earnings Calls*

and SEC Filings, and the last example in this chapter using SEC filings).

Bidirectional RNNs can also be used with two-dimensional image data. In this case, one pair of RNNs performs the forward and backward processing of the sequence in each dimension.

Encoder-decoder architectures, attention, and transformers

The architectures discussed so far assumed that the input and output sequences have equal length. Encoder-decoder architectures, also called **sequence-to-sequence (seq2seq)** architectures, relax this assumption and have become very popular for machine translation and other applications with this characteristic (Prabhavalkar et al., 2017).

The **encoder** is an RNN that maps the input space to a different space, also called **latent space**, whereas the **decoder** function is a complementary RNN that maps the encoded input to the target space (Cho et al., 2014). In the next chapter, we will cover autoencoders that learn a feature representation in an unsupervised setting using a variety of deep learning architectures.

Encoder and decoder RNNs are trained jointly so that the input of the final encoder hidden state becomes the input to the decoder, which, in turn, learns to match the training samples.

The **attention mechanism** addresses a limitation of using fixed-size encoder inputs when input sequences themselves vary in size. The mechanism converts raw text data into a distributed representation (see *Chapter 16, Word Embeddings for Earnings Calls and SEC Filings*), stores the result, and uses a weighted average of these feature

vectors as context. The weights are learned by the model and alternate between putting more weight or attention to different elements of the input.

A recent **transformer** architecture dispenses with recurrence and convolutions and exclusively relies on this attention mechanism to learn input-output mappings. It has achieved superior quality on machine translation tasks while requiring much less time for training, not least because it can be parallelized (Vaswani et al., 2017).

How to design deep RNNs

The **unrolled computational graph** in *Figure 19.2* shows that each transformation involves a linear matrix operation followed by a nonlinear transformation that could be jointly represented by a single network layer.

In the two preceding chapters, we saw how adding depth allows FFNNs, and CNNs in particular, to learn more useful hierarchical representations. RNNs also benefit from decomposing the input-output mapping into multiple layers. For RNNs, this mapping typically transforms:

- The input and the prior hidden state into the current hidden state
- The hidden state into the output

A common approach is to **stack recurrent layers** on top of each other so that they learn a hierarchical temporal representation of the input data. This means that a lower layer may capture higher-frequency patterns, synthesized by a higher layer into lower-frequency

characteristics that prove useful for the classification or regression task. We will demonstrate this approach in the next section.

Less popular alternatives include adding layers to the connections from input to the hidden state, between hidden states, or from the hidden state to the output. These designs employ skip connections to avoid a situation where the shortest path between time steps increases and training becomes more difficult.

The challenge of learning long-range dependencies

In theory, RNNs can make use of information in arbitrarily long sequences. However, in practice, they are limited to looking back only a few steps. More specifically, RNNs struggle to derive useful context information from time steps far apart from the current observation (Hochreiter et al., 2001).

The fundamental problem is the impact of repeated multiplication on gradients during backpropagation over many time steps. As a result, the **gradients tend to either vanish** and decrease toward zero (the typical case), **or explode** and grow toward infinity (less frequent, but rendering optimization very difficult).

Even if parameters allow stability and the network is able to store memories, long-term interactions will receive exponentially smaller weights due to the multiplication of many Jacobians, the matrices containing the gradient information. Experiments have shown that stochastic gradient descent faces serious challenges in training RNNs for sequences with only 10 or 20 elements.

Several RNN design techniques have been introduced to address this challenge, including **echo state networks** (Jaeger, 2001) and **leaky units** (Hinton and Bengio, 1996). The latter operate at different time scales, focusing part of the model on higher-frequency and other parts on lower-frequency representations to deliberately learn and combine different aspects from the data. Other strategies include connections that skip time steps or units that integrate signals from different frequencies.

The most successful approaches use gated units that are trained to regulate how much past information a unit maintains in its current state and when to reset or forget this information. As a result, they are able to learn dependencies over hundreds of time steps. The most popular examples include **long short-term memory (LSTM)** units and **gated recurrent units (GRUs)**. An empirical comparison by Chung et al. (2014) finds both units superior to simpler recurrent units such as tanh units, while performing equally well on various speech and music modeling tasks.

Long short-term memory – learning how much to forget

RNNs with an LSTM architecture have more complex units that maintain an internal state. They contain gates to keep track of dependencies between elements of the input sequence and regulate the cell's state accordingly. These gates recurrently connect to each other instead of the hidden units we encountered earlier. They aim to address the problem of vanishing and exploding gradients due to the repeated multiplication of possibly very small or very large values by letting gradients pass through unchanged (Hochreiter and Schmidhuber, 1996).

The diagram in *Figure 19.3* shows the information flow for an unrolled LSTM unit and outlines its typical gating mechanism:

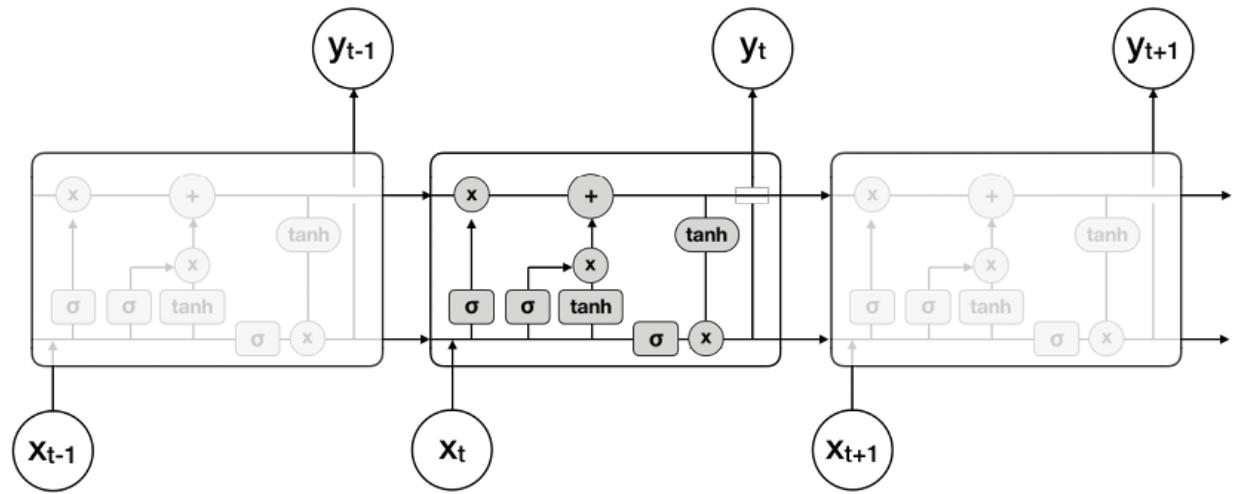


Figure 19.3: Information flow through an unrolled LSTM cell

A typical LSTM unit combines **four parameterized layers** that interact with each other and the cell state by transforming and passing along vectors. These layers usually involve an input gate, an output gate, and a forget gate, but there are variations that may have additional gates or lack some of these mechanisms. The white nodes in *Figure 19.4* identify element-wise operations, and the gray elements represent layers with weight and bias parameters learned during training:

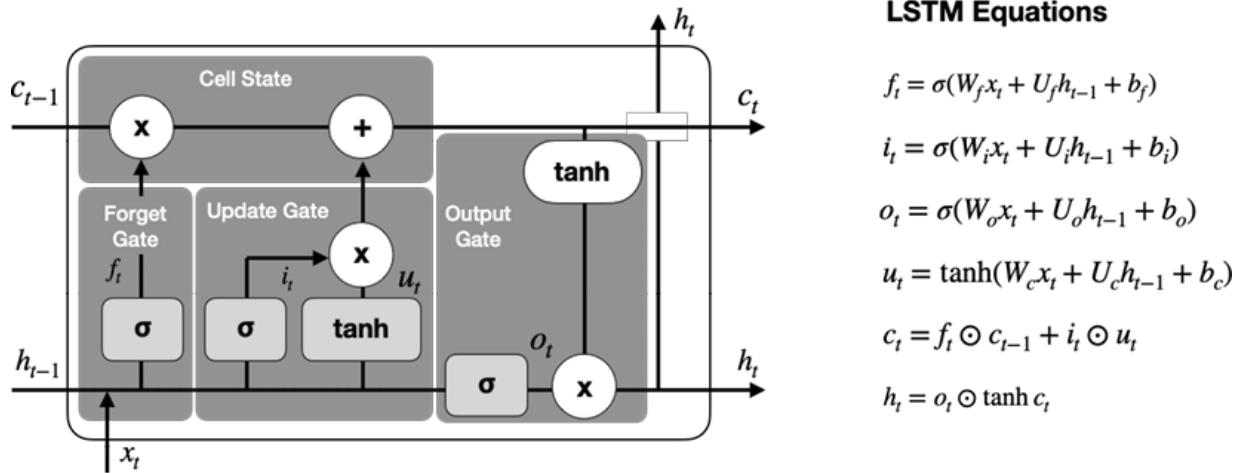


Figure 19.4: The logic of, and math behind, an LSTM cell

The **cell state**, c , passes along the horizontal connection at the top of the cell. The cell state's interaction with the various gates leads to a series of recurrent decisions:

1. The **forget gate** controls how much of the cell's state should be voided to regulate the network's memory. It receives the prior hidden state, h_{t-1} , and the current input, x_t , as inputs, computes a sigmoid activation, and multiplies the resulting value, f_t , which has been normalized to the $[0, 1]$ range, by the cell state, reducing or keeping it accordingly.
2. The **input gate** also computes a sigmoid activation from h_{t-1} and x_t that produces update candidates. A \tanh activation in the range from $[-1, 1]$ multiplies the update candidates, u_t , and, depending on the resulting sign, adds or subtracts the result from the cell state.
3. The **output gate** filters the updated cell state using a sigmoid activation, o_t , and multiplies it by the cell state normalized to the range $[-1, 1]$ using a \tanh activation.

Gated recurrent units

GRUs simplify LSTM units by omitting the output gate. They have been shown to achieve similar performance on certain language modeling tasks, but do better on smaller datasets.

GRUs aim for each recurrent unit to adaptively capture dependencies of different time scales. Similar to the LSTM unit, the GRU has gating units that modulate the flow of information inside the unit but discard separate memory cells (see references on GitHub for additional details).

RNNs for time series with TensorFlow 2

In this section, we illustrate how to build recurrent neural nets using the TensorFlow 2 library for various scenarios. The first set of models includes the regression and classification of univariate and multivariate time series. The second set of tasks focuses on text data for sentiment analysis using text data converted to word embeddings (see *Chapter 16, Word Embeddings for Earnings Calls and SEC Filings*).

More specifically, we'll first demonstrate how to prepare time-series data to predict the next value for **univariate time series** with a single LSTM layer to predict stock index values.

Next, we'll build a **deep RNN** with three distinct inputs to classify asset price movements. To this end, we'll combine a two-layer, **stacked LSTM** with learned **embeddings** and one-hot encoded

categorical data. Finally, we will demonstrate how to model **multivariate time series** using an RNN.

Univariate regression – predicting the S&P 500

In this subsection, we will forecast the S&P 500 index values (refer to the `univariate_time_series_regression` notebook for implementation details).

We'll obtain data for 2010-2019 from the Federal Reserve Bank's Data Service (FRED; see *Chapter 2, Market and Fundamental Data – Sources and Techniques*):

```
sp500 = web.DataReader('SP500', 'fred', start='2010', end='2020').dropna()
sp500.info()
DatetimeIndex: 2463 entries, 2010-03-22 to 2019-12-31
Data columns (total 1 columns):
 #   Column   Non-Null Count   Dtype  
 ---  --       --       --       --    
 0   SP500    2463 non-null    float64
```

We preprocess the data by scaling it to the $[0, 1]$ interval using scikit-learn's `MinMaxScaler()` class:

```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
sp500_scaled = pd.Series(scaler.fit_transform(sp500).squeeze(),
                         index=sp500.index)
```

How to get time series data into shape for an RNN

We generate sequences of 63 consecutive trading days, approximately three months, and use a single LSTM layer with 20 hidden units to predict the scaled index value one time step ahead.

The input to every LSTM layer must have three dimensions, namely:

- **Batch size:** One sequence is one sample. A batch contains one or more samples.
- **Time steps:** One time step is a single observation in the sample.
- **Features:** One feature is one observation at a time step.

The following figure visualizes the shape of the input tensor:

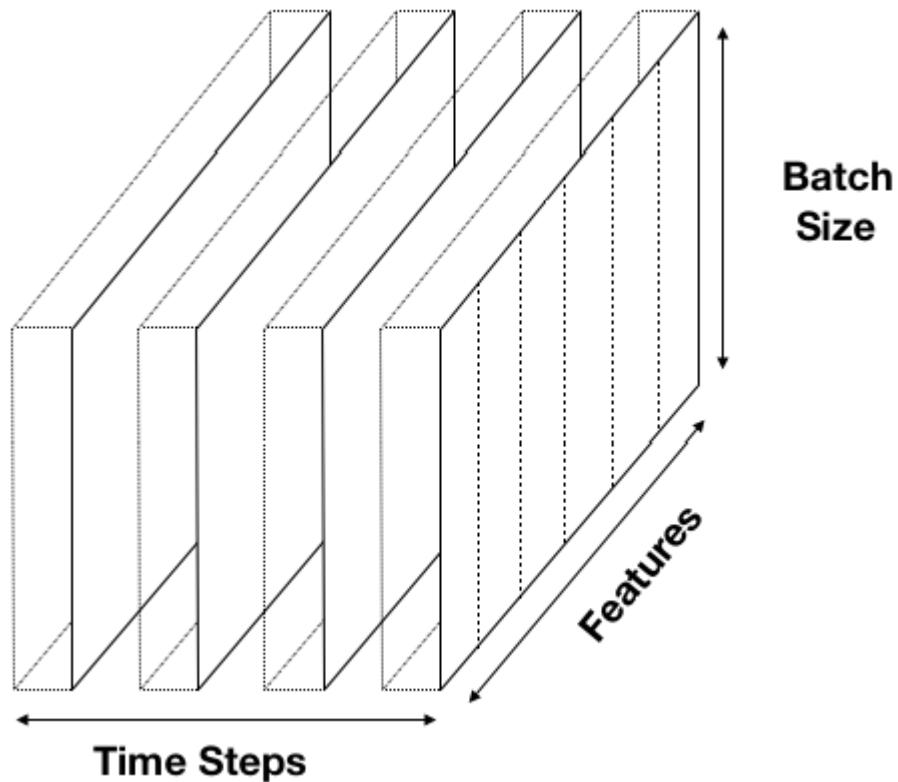


Figure 19.5: The three dimensions of an RNN input tensor

Our S&P 500 sample has 2,463 observations or time steps. We will create overlapping sequences using a window of 63 observations

each. Using a simpler window of size $T = 5$ to illustrate this autoregressive sequence pattern, we obtain input-output pairs where each output is associated with its first five lags, as shown in the following table:

Input	Output
$\langle x_1, x_2, x_3, x_4, x_5 \rangle$	x_6
$\langle x_2, x_3, x_4, x_5, x_6 \rangle$	x_7
\vdots	\vdots
$\langle x_{T-5}, x_{T-4}, x_{T-3}, x_{T-2}, x_{T-1} \rangle$	x_T

Figure 19.6: Input-output pairs with a $T=5$ size window

We can use the `create_univariate_rnn_data()` function to stack the overlapping sequences that we select using a rolling window:

```
def create_univariate_rnn_data(data, window_size):
    y = data[window_size:]
    data = data.values.reshape(-1, 1) # make 2D
    n = data.shape[0]
    X = np.hstack(tuple([data[i: n-j, :] for i, j in enumerate(range(
        window_size, 0, -1))]))
    return pd.DataFrame(X, index=y.index), y
```

We apply this function to the rescaled stock index using `window_size=63` to obtain a two-dimensional dataset with a shape of the number of samples x the number of time steps:

```
X, y = create_univariate_rnn_data(sp500_scaled, window_size=63)
X.shape
(2356, 63)
```

We will use data from 2019 as our test set and reshape the features to add a requisite third dimension:

```
X_train = X[:'2018'].values.reshape(-1, window_size, 1)
y_train = y[:'2018']
# keep the last year for testing
X_test = X['2019'].values.reshape(-1, window_size, 1)
y_test = y['2019']
```

How to define a two-layer RNN with a single LSTM layer

Now that we have created autoregressive input/output pairs from our time series and split the pairs into training and test sets, we can define our RNN architecture. The Keras interface of TensorFlow 2 makes it very straightforward to build an RNN with two hidden layers with the following specifications:

- **Layer 1:** An LSTM module with 10 hidden units (with `input_shape = (window_size,1)`; we will define `batch_size` in the omitted first dimension during training)
- **Layer 2:** A fully connected module with a single unit and linear activation
- **Loss:** `mean_squared_error` to match the regression objective

Just a few lines of code create the computational graph:

```
rnn = Sequential([
    LSTM(units=10,
          input_shape=(window_size, n_features), name='LSTM'),
    Dense(1, name='Output')
])
```

The summary shows that the model has 491 parameters:

```
rnn.summary()
Layer (type)          Output Shape         Param #
LSTM (LSTM)           (None, 10)           480
Output (Dense)        (None, 1)            11
Total params: 491
Trainable params: 491
```

Training and evaluating the model

We train the model using the RMSProp optimizer recommended for RNN with default settings and compile the model with `mean_squared_error` for this regression problem:

```
optimizer = keras.optimizers.RMSprop(lr=0.001,
                                      rho=0.9,
                                      epsilon=1e-08,
                                      decay=0.0)
rnn.compile(loss='mean_squared_error', optimizer=optimizer)
```

We define an `EarlyStopping` callback and train the model for 500 episodes:

```
early_stopping = EarlyStopping(monitor='val_loss',
                               patience=50,
                               restore_best_weights=True)
lstm_training = rnn.fit(X_train,
                        y_train,
                        epochs=500,
                        batch_size=20,
                        validation_data=(X_test, y_test),
                        callbacks=[checkpointer, early_stopping],
                        verbose=1)
```

Training stops after 138 epochs. The loss history in *Figure 19.7* shows the 5-epoch rolling average of the training and validation RMSE, highlights the best epoch, and shows that the loss is 0.998 percent:

```

loss_history = pd.DataFrame(lstm_training.history).pow(.5)
loss_history.index += 1
best_rmse = loss_history.val_loss.min()
best_epoch = loss_history.val_loss.idxmin()
loss_history.columns=['Training RMSE', 'Validation RMSE']
title = f'Best Validation RMSE: {best_rmse:.4%}'
loss_history.rolling(5).mean().plot(logy=True, lw=2, title=title, ax=

```

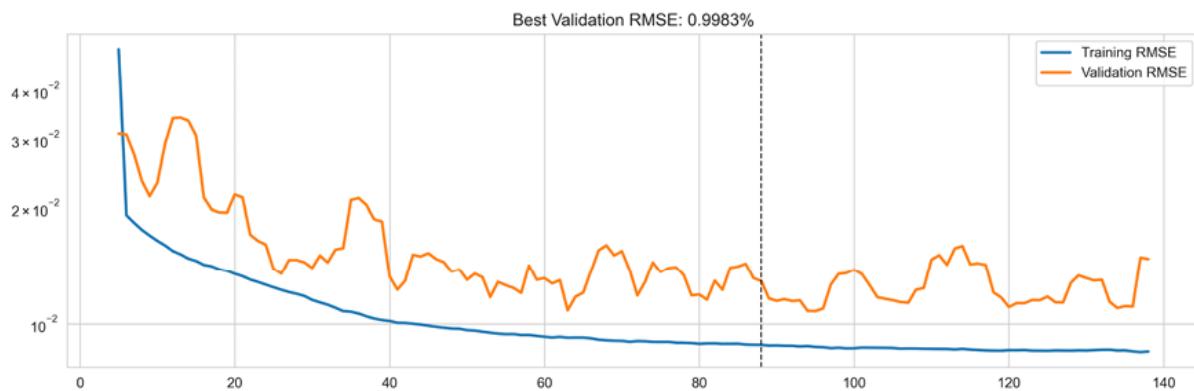


Figure 19.7: Cross-validation performance

Re-scaling the predictions

We use the `inverse_transform()` method of `MinMaxScaler()` to rescale the model predictions to the original S&P 500 range of values:

```

test_predict_scaled = rnn.predict(X_test)
test_predict = (pd.Series(scaler.inverse_transform(test_predict_scaled)
                         .squeeze(),
                         index=y_test.index))

```

The four plots in *Figure 19.8* illustrate the forecast performance based on the rescaled predictions that track the 2019 out-of-sample S&P 500 data with a test **information coefficient (IC)** of 0.9889:

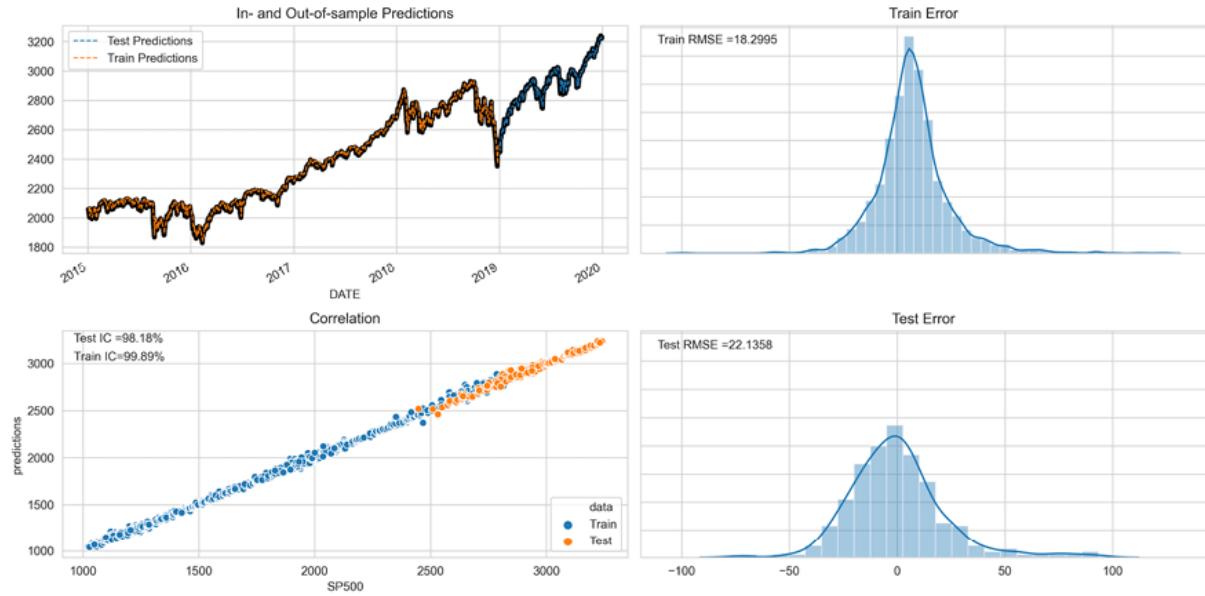


Figure 19.8: RNN performance on S&P 500 predictions

Stacked LSTM – predicting price moves and returns

We'll now build a deeper model by stacking two LSTM layers using the `Quandl` stock price data (see the `stacked_lstm_with_feature_embeddings.ipynb` notebook for implementation details). Furthermore, we will include features that are not sequential in nature, namely, indicator variables identifying the equity and the month.

Figure 19.9 outlines the architecture that illustrates how to combine different data sources in a single deep neural network. For example, instead of, or in addition to, one-hot encoded months, you could add technical or fundamental features:

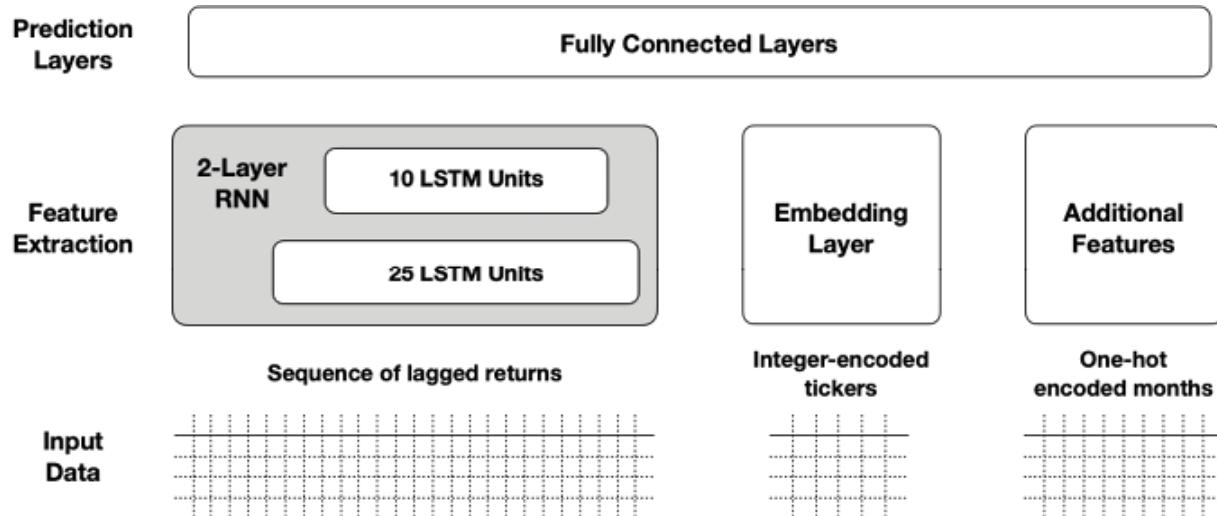


Figure 19.9: Stacked LSTM architecture with additional features

Preparing the data – how to create weekly stock returns

We load the Quandl adjusted stock price data (see instructions on GitHub on how to obtain the source data) as follows (refer to the `build_dataset.ipynb` notebook):

```
prices = (pd.read_hdf('../data/assets.h5', 'quandl/wiki/prices')
          .adj_close
          .unstack().loc['2007':])
prices.info()
DatetimeIndex: 2896 entries, 2007-01-01 to 2018-03-27
Columns: 3199 entries, A to ZUMZ
```

We start by generating weekly returns for close to 2,500 stocks with complete data for the 2008-17 period:

```
returns = (prices
           .resample('W')
           .last()
           .pct_change()
           .loc['2008': '2017']
           .dropna(axis=1)
           .sort_index(ascending=False))
returns.info()
```

```
DatetimeIndex: 2576 entries, 2017-12-29 to 2008-01-01
Columns: 2489 entries, A to ZUMZ
```

We create and stack rolling sequences of 52 weekly returns for each ticker and week as follows:

```
n = len(returns)
T = 52
tcols = list(range(T))
tickers = returns.columns
data = pd.DataFrame()
for i in range(n-T-1):
    df = returns.iloc[i:i+T+1]
    date = df.index.max()
    data = pd.concat([data, (df.reset_index(drop=True).T
                           .assign(date=date, ticker=tickers)
                           .set_index(['ticker', 'date']))])
```

We winsorize outliers at the 1 and 99 percentile level and create a binary label that indicates whether the weekly return was positive:

```
data[tcols] = (data[tcols].apply(lambda x: x.clip(lower=x.quantile(.01)
                                                upper=x.quantile(.99)))
data['label'] = (data['fwd_returns'] > 0).astype(int)
```

As a result, we obtain 1.16 million observations on over 2,400 stocks with 52 weeks of lagged returns each (plus the label):

```
data.shape
(1167341, 53)
```

Now we are ready to create the additional features, split the data into training and test sets, and bring them into the three-dimensional format required for the LSTM.

How to create multiple inputs in RNN format

This example illustrates how to combine several input data sources, namely:

- Rolling sequences of 52 weeks of lagged returns
- One-hot encoded indicator variables for each of the 12 months
- Integer-encoded values for the tickers

The following code generates the two additional features:

```
data['month'] = data.index.get_level_values('date').month
data = pd.get_dummies(data, columns=['month'], prefix='month')
data['ticker'] = pd.factorize(data.index.get_level_values('ticker'))[0]
```

Next, we create a training set covering the 2009-2016 period and a separate test set with data for 2017, the last full year with data:

```
train_data = data[:'2016']
test_data = data['2017']
```

For training and test datasets, we generate a list containing the three input arrays as shown in *Figure 19.9*:

- The lagged return series (using the format described in *Figure 19.5*)
- The integer-encoded stock ticker as a one-dimensional array
- The month dummies as a two-dimensional array with one column per month

```
window_size=52
sequence = list(range(1, window_size+1))
X_train = [
```

```

train_data.loc[:, sequence].values.reshape(-1, window_size, 1),
train_data.ticker,
train_data.filter(like='month')
]
y_train = train_data.label
[x.shape for x in X_train], y_train.shape
[(1035424, 52, 1), (1035424,), (1035424, 12)], (1035424,)

```

How to define the architecture using Keras' Functional API

Keras' Functional API makes it easy to design an architecture like the one outlined at the beginning of this section with multiple inputs (or several outputs, as in the SVHN example in *Chapter 18, CNNs for Financial Time Series and Satellite Images*). This example illustrates a network with three inputs:

1. **Two stacked LSTM layers** with 25 and 10 units, respectively
2. An **embedding layer** that learns a 10-dimensional real-valued representation of the equities
3. A **one-hot encoded** representation of the month

We begin by defining the three inputs with their respective shapes:

```

n_features = 1
returns = Input(shape=(window_size, n_features), name='Returns')
tickers = Input(shape=(1,), name='Tickers')
months = Input(shape=(12,), name='Months')

```

To define **stacked LSTM layers**, we set the `return_sequences` keyword for the first layer to `True`. This ensures that the first layer produces an output in the expected three-dimensional input format. Note that we also use dropout regularization and how the Functional API passes the tensor outputs from one layer to the subsequent layer's input:

```

lstm1 = LSTM(units=lstm1_units,
             input_shape=(window_size, n_features),
             name='LSTM1',
             dropout=.2,
             return_sequences=True)(returns)
lstm_model = LSTM(units=lstm2_units,
                  dropout=.2,
                  name='LSTM2')(lstm1)

```

The TensorFlow 2 guide for RNNs highlights the fact that GPU support is only available when using the default values for most LSTM settings

(<https://www.tensorflow.org/guide/keras/rnn>).

The **embedding layer** requires:

- The `input_dim` keyword, which defines how many embeddings the layer will learn
- The `output_dim` keyword, which defines the size of the embedding
- The `input_length` parameter, which sets the number of elements passed to the layer (here, only one ticker per sample)

The goal of the embedding layer is to learn vector representations that capture the relative locations of the feature values to one another with respect to the outcome. We'll choose a five-dimensional embedding for the roughly 2,500 ticker values to combine the embedding layer with the LSTM layer and the month dummies we need to reshape (or flatten) it:

```

ticker_embedding = Embedding(input_dim=n_tickers,
                             output_dim=5,
                             input_length=1)(tickers)
ticker_embedding = Reshape(target_shape=(5,))(ticker_embedding)

```

Now we can concatenate the three tensors, followed by

BatchNormalization :

```
merged = concatenate([lstm_model, ticker_embedding, months], name='Mer
bn = BatchNormalization()(merged)
```

The fully connected final layers learn a mapping from these stacked LSTM layers, ticker embeddings, and month indicators to the binary outcome that reflects a positive or negative return over the following week. We formulate the complete RNN by defining its inputs and outputs with the implicit data flow we just defined:

```
hidden_dense = Dense(10, name='FC1')(bn)
output = Dense(1, name='Output', activation='sigmoid')(hidden_dense)
rnn = Model(inputs=[returns, tickers, months], outputs=output)
```

The summary lays out this slightly more sophisticated architecture with 16,984 parameters:

Layer (type)	Output Shape	Param #	Connec
Returns (InputLayer)	[(None, 52, 1)]	0	
Tickers (InputLayer)	[(None, 1)]	0	
LSTM1 (LSTM)	(None, 52, 25)	2700	Retur
embedding (Embedding)	(None, 1, 5)	12445	Ticker
LSTM2 (LSTM)	(None, 10)	1440	LSTM1
reshape (Reshape)	(None, 5)	0	embeddi
Months (InputLayer)	[(None, 12)]	0	
Merged (Concatenate)	(None, 27)	0	LSTM2
			reshape
			Month
batch_normalization (BatchNorma	(None, 27)	108	Merge
FC1 (Dense)	(None, 10)	280	
atch_normalization[0][0]			
Output (Dense)	(None, 1)	11	FC1[0]
Total params: 16,984			
Trainable params: 16,930			
Non-trainable params: 54			

We compile the model using the recommended RMSProp optimizer with default settings and compute the AUC metric that we'll use for early stopping:

```
optimizer = tf.keras.optimizers.RMSprop(lr=0.001,
                                         rho=0.9,
                                         epsilon=1e-08,
                                         decay=0.0)
rnn.compile(loss='binary_crossentropy',
            optimizer=optimizer,
            metrics=['accuracy',
                      tf.keras.metrics.AUC(name='AUC')])
```

We train the model for 50 epochs by using early stopping:

```
result = rnn.fit(X_train,
                  y_train,
                  epochs=50,
                  batch_size=32,
                  validation_data=(X_test, y_test),
                  callbacks=[early_stopping])
```

The following plots show that training stops after 8 epochs, each of which takes around three minutes on a single GPU. It results in a test AUC of 0.6816 and a test accuracy of 0.6193 for the best model:

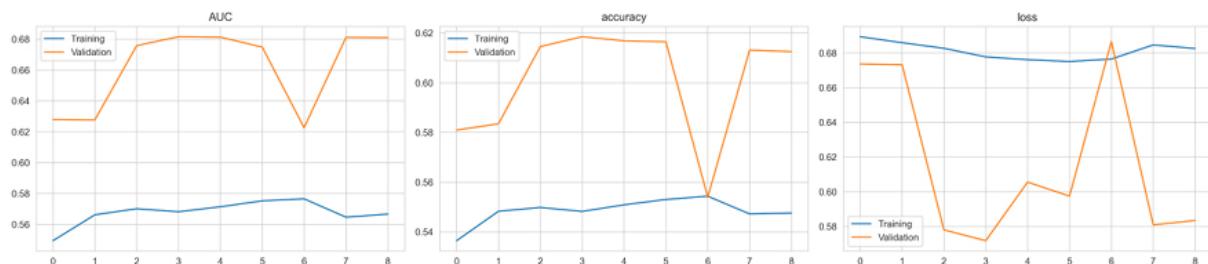


Figure 19.10: Stacked LSTM classification—cross-validation performance

The IC for the test prediction and actual weekly returns is 0.32.

Predicting returns instead of directional price moves

The `stacked_lstm_with_feature_embeddings_regression.ipynb` notebook illustrates how to adapt the model to the regression task of predicting returns rather than binary price changes.

The required changes are minor; just do the following:

1. Select the `fwd_returns` outcome instead of the binary `label`.
2. Convert the model output to linear (the default) instead of `sigmoid`.
3. Update the loss to mean squared error (and early stopping references).
4. Remove or update optional metrics to match the regression task.

Using otherwise the same training parameters (except that the Adam optimizer with default settings yields a better result in this case), the validation loss improves for nine epochs. The average weekly IC is 3.32, and 6.68 for the entire period while significant at the 1 percent level. The average weekly return differential between the equities in the top and bottom quintiles of predicted returns is slightly above 20 basis points:

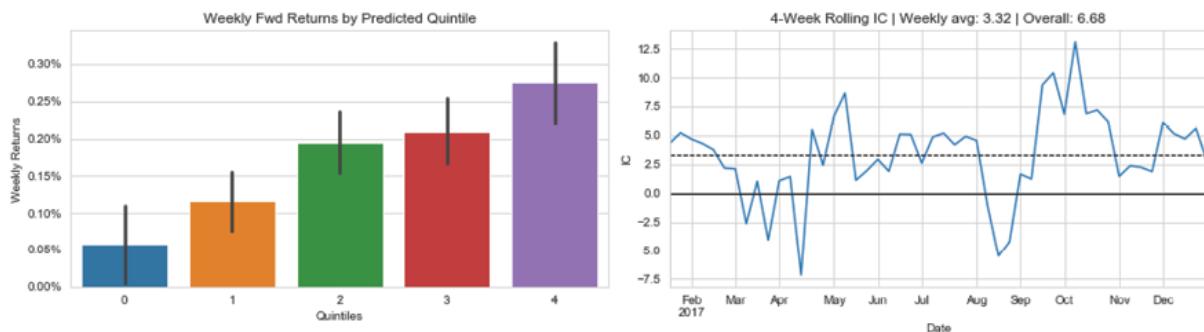


Figure 19.11: Stacked LSTM regression—out-of-sample performance

Multivariate time-series regression for macro data

So far, we have limited our modeling efforts to a single time series. RNNs are well-suited to multivariate time series and represent a nonlinear alternative to the **vector autoregressive (VAR)** models we covered in *Chapter 9, Time-Series Models for Volatility Forecasts and Statistical Arbitrage*. Refer to the `multivariate_timeseries` notebook for implementation details.

Loading sentiment and industrial production data

We'll show how to model and forecast multiple time series using RNNs with the same dataset we used for the VAR example. It has monthly observations over 40 years on consumer sentiment and industrial production from the Federal Reserve's FRED service:

```
df = web.DataReader(['UMCSENT', 'IPGMFN'], 'fred', '1980', '2019-12').  
df.columns = ['sentiment', 'ip']  
df.info()  
DatetimeIndex: 480 entries, 1980-01-01 to 2019-12-01  
Data columns (total 2 columns):  
sentiment    480 non-null float64  
ip          480 non-null float64
```

Making the data stationary and adjusting the scale

We apply the same transformation—annual difference for both series, prior log-transform for industrial production—to achieve stationarity (see *Chapter 9, Time-Series Models for Volatility Forecasts and Statistical Arbitrage* for details). We also rescale it to the [0, 1] range to ensure that the network gives both series equal weight during training:

```
df_transformed = (pd.DataFrame({'ip': np.log(df.ip).diff(12),
                               'sentiment': df.sentiment.diff(12)}).dr
df_transformed = df_transformed.apply(minmax_scale)
```

Figure 19.12 displays the original and transformed macro time series:

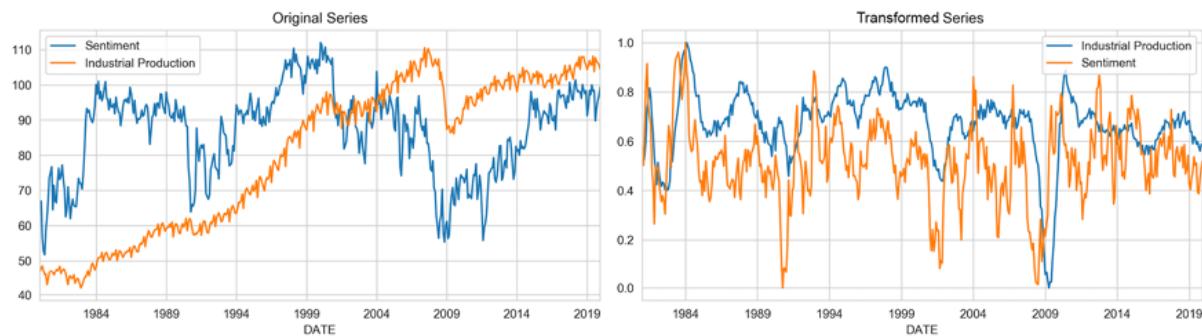


Figure 19.12: Original and transformed time series

Creating multivariate RNN inputs

The `create_multivariate_rnn_data()` function transforms a dataset of several time series into the three-dimensional shape required by TensorFlow's RNN layers, formed as `n_samples × window_size × n_series`:

```
def create_multivariate_rnn_data(data, window_size):
    y = data[window_size:]
    n = data.shape[0]
    X = np.stack([data[i: j] for i, j in enumerate(range(window_size,
                                                          axis=0))
    return X, y
```

A `window_size` value of 18 ensures that the entries in the second dimension are the lagged 18 months of the respective output

variable. We thus obtain the RNN model inputs for each of the two features as follows:

```
X, y = create_multivariate_rnn_data(df_transformed, window_size=window_size)
X.shape, y.shape
((450, 18, 2), (450, 2))
```

Finally, we split our data into a training and a test set, using the last 24 months to test the out-of-sample performance:

```
test_size = 24
train_size = X.shape[0]-test_size
X_train, y_train = X[:train_size], y[:train_size]
X_test, y_test = X[train_size:], y[train_size:]
X_train.shape, X_test.shape
((426, 18, 2), (24, 18, 2))
```

Defining and training the model

Given the relatively small dataset, we use a simpler RNN architecture than in the previous example. It has a single LSTM layer with 12 units, followed by a fully connected layer with 6 units. The output layer has two units, one for each time series.

We compile using mean absolute loss and the recommended RMSProp optimizer:

```
n_features = output_size = 2
lstm_units = 12
dense_units = 6
rnn = Sequential([
    LSTM(units=lstm_units,
        dropout=.1,
        recurrent_dropout=.1,
        input_shape=(window_size, n_features), name='LSTM',
        return_sequences=False),
```

```

        Dense(dense_units, name='FC'),
        Dense(output_size, name='Output')
    ])
rnn.compile(loss='mae', optimizer='RMSProp')

```

The model still has 812 parameters, compared to 10 for the `VAR(1, 1)` model from *Chapter 9, Time-Series Models for Volatility Forecasts and Statistical Arbitrage*:

Layer (type)	Output Shape	Param #
LSTM (LSTM)	(<code>None</code> , 12)	720
FC (Dense)	(<code>None</code> , 6)	78
Output (Dense)	(<code>None</code> , 2)	14
Total params:	812	
Trainable params:	812	

We train for 100 epochs with a `batch_size` of 20 using early stopping:

```

result = rnn.fit(X_train,
                  y_train,
                  epochs=100,
                  batch_size=20,
                  shuffle=False,
                  validation_data=(X_test, y_test),
                  callbacks=[checkpointer, early_stopping],
                  verbose=1)

```

Training stops early after 62 epochs, yielding a test MAE of 0.034, an almost 25 percent improvement over the test MAE for the VAR model of 0.043 on the same task.

However, the two results are not fully comparable because the RNN produces 18 1-step-ahead forecasts whereas the VAR model uses its own predictions as input for its out-of-sample forecast. You may want to tweak the VAR setup to obtain comparable forecasts and compare the performance.

Figure 19.13 highlights training and validation errors, and the out-of-sample predictions for both series:

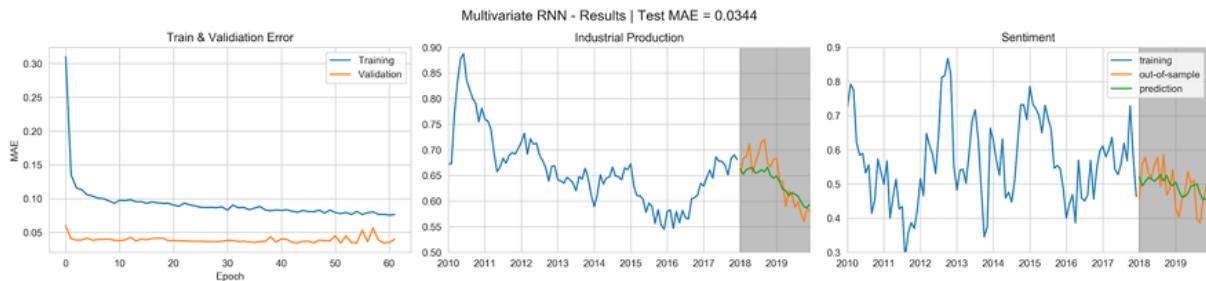


Figure 19.13: Cross-validation and test results for RNNs with multiple macro series

RNNs for text data

RNNs are commonly applied to various natural language processing tasks, from machine translation to sentiment analysis, that we already encountered in Part 3 of this book. In this section, we will illustrate how to apply an RNN to text data to detect positive or negative sentiment (easily extensible to a finer-grained sentiment scale) and to predict stock returns.

More specifically, we'll use word embeddings to represent the tokens in the documents. We covered word embeddings in *Chapter 16, Word Embeddings for Earnings Calls and SEC Filings*. They are an excellent technique for converting a token into a dense, real-value vector because the relative location of words in the embedding space encodes useful semantic aspects of how they are used in the training documents.

We saw in the previous stacked RNN example that TensorFlow has a built-in embedding layer that allows us to train vector representations specific to the task at hand. Alternatively, we can use

pretrained vectors. We'll demonstrate both approaches in the following three sections.

LSTM with embeddings for sentiment classification

This example shows how to learn custom embedding vectors while training an RNN on the classification task. This differs from the word2vec model that learns vectors while optimizing predictions of neighboring tokens, resulting in their ability to capture certain semantic relationships among words (see *Chapter 16, Word Embeddings for Earnings Calls and SEC Filings*). Learning word vectors with the goal of predicting sentiment implies that embeddings will reflect how a token relates to the outcomes it is associated with.

Loading the IMDB movie review data

To keep the data manageable, we will illustrate this use case with the IMDB reviews dataset, which contains 50,000 positive and negative movie reviews, evenly split into a training set and a test set, with balanced labels in each dataset. The vocabulary consists of 88,586 tokens. Alternatively, you could use the much larger Yelp review data (after converting the text into numerical sequences; see the next section on using pretrained embeddings or TensorFlow 2 docs).

The dataset is bundled into TensorFlow and can be loaded so that each review is represented as an integer-encoded sequence. We can limit the vocabulary to `num_words` while filtering out frequent and likely less informative words using `skip_top` as well as sentences longer than `maxlen`. We can also choose the `oov_char` value, which

represents tokens we chose to exclude from the vocabulary on frequency grounds:

```
from tensorflow.keras.datasets import imdb
vocab_size = 20000
(X_train, y_train), (X_test, y_test) = imdb.load_data(seed=42,
                                                       skip_top=0,
                                                       maxlen=None,
                                                       oov_char=2,
                                                       index_from=3,
                                                       num_words=vocab_
```

In the second step, convert the lists of integers into fixed-size arrays that we can stack and provide as an input to our RNN. The `pad_sequence` function produces arrays of equal length, truncated and padded to conform to `maxlen` :

```
maxlen = 100
X_train_padded = pad_sequences(X_train,
                                truncating='pre',
                                padding='pre',
                                maxlen=maxlen)
```

Defining embedding and the RNN architecture

Now we can set up our RNN architecture. The first layer learns the word embeddings. We define the embedding dimensions as before, using the following:

- The `input_dim` keyword, which sets the number of tokens that we need to embed
- The `output_dim` keyword, which defines the size of each embedding

- The `input_len` parameter, which specifies how long each input sequence is going to be

Note that we are using GRU units this time that train faster and perform better on smaller amounts of data. We are also using recurrent dropout for regularization:

```
embedding_size = 100
rnn = Sequential([
    Embedding(input_dim=vocab_size,
              output_dim= embedding_size,
              input_length=maxlen),
    GRU(units=32,
         dropout=0.2, # comment out to use optimized GPU implementation
         recurrent_dropout=0.2),
    Dense(1, activation='sigmoid')
])
```

The resulting model has over 2 million trainable parameters:

Layer (type)	Output Shape	Param #
embedding (Embedding)	(None, 100, 100)	2000000
gru (GRU)	(None, 32)	12864
dense (Dense)	(None, 1)	33
Total params: 2,012,897		
Trainable params: 2,012,897		

We compile the model to use the AUC metric and train with early stopping:

```
rnn.fit(X_train_padded,
        y_train,
        batch_size=32,
        epochs=25,
        validation_data=(X_test_padded, y_test),
        callbacks=[early_stopping],
        verbose=1)
```

Training stops after 12 epochs, and we recover the weights for the best models to find a high test AUC of 0.9393:

```
y_score = rnn.predict(X_test_padded)
roc_auc_score(y_score=y_score.squeeze(), y_true=y_test)
0.9393289376
```

Figure 19.14 displays the cross-validation performance in terms of accuracy and AUC:

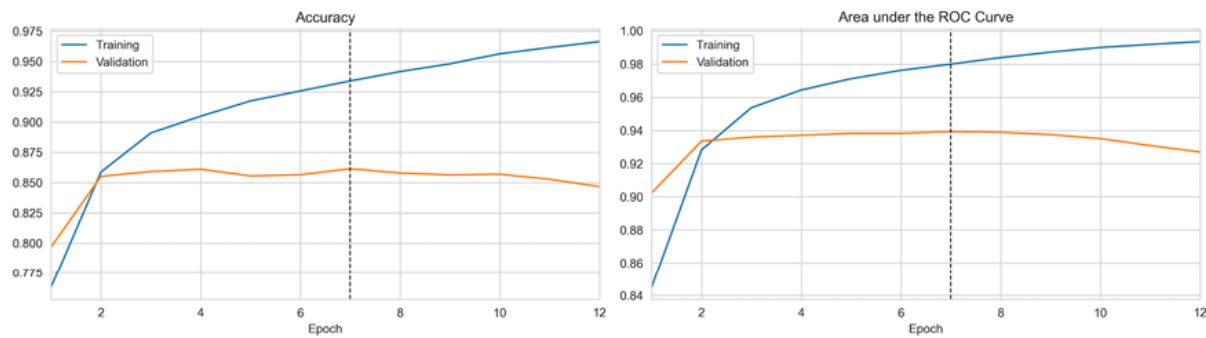


Figure 19.14: Cross-validation for RNN using IMDB data with custom embeddings

Sentiment analysis with pretrained word vectors

In *Chapter 16, Word Embeddings for Earnings Calls and SEC Filings*, we discussed how to learn domain-specific word embeddings. Word2vec and related learning algorithms produce high-quality word vectors but require large datasets. Hence, it is common that research groups share word vectors trained on large datasets, similar to the weights for pretrained deep learning models that we encountered in the section on transfer learning in the previous chapter.

We are now going to illustrate how to use pretrained **global vectors for word representation (GloVe)** provided by the Stanford NLP group with the IMDB review dataset (refer to GitHub for references and the `sentiment_analysis_pretrained_embeddings` notebook for implementation details).

Preprocessing the text data

We are going to load the IMDB dataset from the source to manually preprocess it (see the notebook). TensorFlow provides a `Tokenizer`, which we'll use to convert the text documents to integer-encoded sequences:

```
num_words = 10000
t = Tokenizer(num_words=num_words,
              lower=True,
              oov_token=2)
t.fit_on_texts(train_data.review)
vocab_size = len(t.word_index) + 1
train_data_encoded = t.texts_to_sequences(train_data.review)
test_data_encoded = t.texts_to_sequences(test_data.review)
```

We also use the `pad_sequences` function to convert the list of lists (of unequal length) to stacked sets of padded and truncated arrays for both the training and test data:

```
max_length = 100
X_train_padded = pad_sequences(train_data_encoded,
                                maxlen=max_length,
                                padding='post',
                                truncating='post')
y_train = train_data['label']
X_train_padded.shape
(25000, 100)
```

Loading the pretrained GloVe embeddings

We downloaded and unzipped the GloVe data to the location indicated in the code and will now create a dictionary that maps GloVe tokens to 100-dimensional, real-valued vectors:

```
glove_path = Path('data/glove/glove.6B.100d.txt')
embeddings_index = dict()
for line in glove_path.open(encoding='latin1'):
    values = line.split()
    word = values[0]
    coefs = np.asarray(values[1:], dtype='float32')
    embeddings_index[word] = coefs
```

There are around 340,000 word vectors that we use to create an embedding matrix that matches the vocabulary so that the RNN can access embeddings by the token index:

```
embedding_matrix = np.zeros((vocab_size, 100))
for word, i in t.word_index.items():
    embedding_vector = embeddings_index.get(word)
    if embedding_vector is not None:
        embedding_matrix[i] = embedding_vector
```

Defining the architecture with frozen weights

The difference with the RNN setup in the previous example is that we are going to pass the embedding matrix to the embedding layer and set it to *not trainable* so that the weights remain fixed during training:

```
rnn = Sequential([
    Embedding(input_dim=vocab_size,
              output_dim=embedding_size,
              input_length=max_length,
              weights=[embedding_matrix],
              trainable=False),
    GRU(units=32, dropout=0.2, recurrent_dropout=0.2),
    Dense(1, activation='sigmoid')])
```

From here on, we proceed as before. Training continues for 32 epochs, as shown in *Figure 19.15*, and we obtain a test AUC score of 0.9106. This is slightly worse than our result in the previous sections where we learned custom embedding for this domain, underscoring the value of training your own word embeddings:

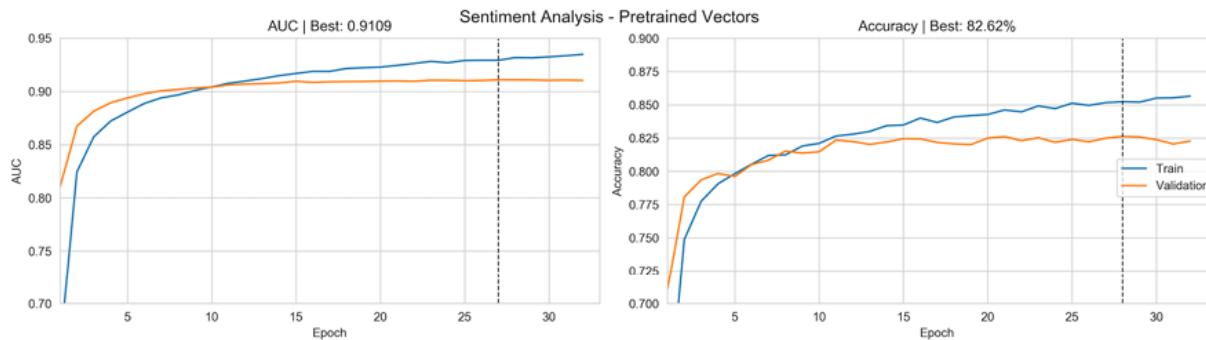


Figure 19.15: Cross-validation and test results for RNNs with multiple macro series

You may want to apply these techniques to the larger financial text datasets that we used in Part 3.

Predicting returns from SEC filing embeddings

In *Chapter 16, Word Embeddings for Earnings Calls and SEC Filings*, we discussed important differences between product reviews and financial text data. While the former was useful to illustrate important workflows, in this section, we will tackle more challenging but also more relevant financial documents. More specifically, we will use the SEC filings data introduced in *Chapter 16, Word Embeddings for Earnings Calls and SEC Filings*, to learn word embeddings tailored to predicting the return of the ticker associated with the disclosures from before publication to one week after.

The `sec_filings_return_prediction` notebook contains the code examples for this section. See the `sec_preprocessing` notebook in *Chapter 16, Word Embeddings for Earnings Calls and SEC Filings*, and instructions in the data folder on GitHub on how to obtain the data.

Source stock price data using yfinance

There are 22,631 filings for the period 2013-16. We use yfinance to obtain stock price data for the related 6,630 tickers because it achieves higher coverage than Quandl's WIKI Data. We use the ticker symbol and filing date from the filing index (see *Chapter 16, Word Embeddings for Earnings Calls and SEC Filings*) to download daily adjusted stock prices for three months before and one month after the filing data as follows, capturing both the price data and unsuccessful tickers in the process:

```
    yf_data, missing = [], []
    for i, (symbol, dates) in enumerate(filing_index.groupby('ticker').dat
                                         1):
        ticker = yf.Ticker(symbol)
        for idx, date in dates.to_dict().items():
            start = date - timedelta(days=93)
            end = date + timedelta(days=31)
            df = ticker.history(start=start, end=end)
            if df.empty:
                missing.append(symbol)
            else:
                yf_data.append(df.assign(ticker=symbol, filing=idx))
```

We obtain data on 3,954 tickers and source prices for a few hundred missing tickers using the Quandl Wiki data (see the notebook) and end up with 16,758 filings for 4,762 symbols.

Preprocessing SEC filing data

Compared to product reviews, financial text documents tend to be longer and have a more formal structure. In addition, in this case, we rely on data sourced from EDGAR that requires parsing of the XBRL source (see *Chapter 2, Market and Fundamental Data – Sources and Techniques*) and may have errors such as including material other than the desired sections. We take several steps during preprocessing to address outliers and format the text data as integer sequences of equal length, as required by the model that we will build in the next section:

1. Remove all sentences that contain fewer than 5 or more than 50 tokens; this affects approximately 5 percent of sentences.
2. Create 28,599 bigrams, 10,032 trigrams, and 2,372 n-grams with 4 elements.
3. Convert filings to a sequence of integers that represent the token frequency rank, removing filings with fewer than 100 tokens and truncating sequences at 20,000 elements.

Figure 19.16 highlights some corpus statistics for the remaining 16,538 filings with 179,214,369 tokens, around 204,206 of which are unique. The left panel shows the token frequency distribution on a log-log scale; the most frequent terms, "million," "business," "company," and "products" occur more than 1 million times each. As usual, there is a very long tail, with 60 percent of tokens occurring fewer than 25 times.

The central panel shows the distribution of the sentence lengths with a mode of around 10 tokens. Finally, the right panel shows the distribution of the filing length with a peak at 20,000 due to truncation:

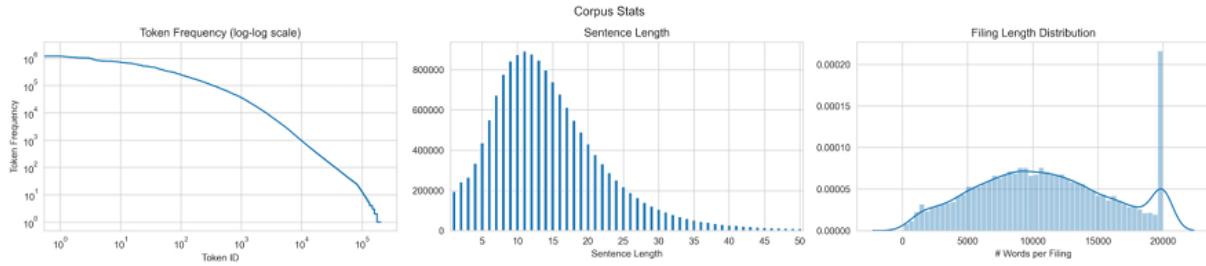


Figure 19.16: Cross-validation and test results for RNNs with multiple macro series

Preparing data for the RNN model

Now we need an outcome for our model to predict. We'll compute (somewhat arbitrarily) five-day forward returns for the day of filing (or the day before if there are no prices for that date), assuming that filing occurred after market hours. Clearly, this assumption could be wrong, underscoring the need for **point-in-time data** emphasized in *Chapter 2, Market and Fundamental Data – Sources and Techniques*, and *Chapter 3, Alternative Data for Finance – Categories and Use Cases*. We'll ignore this issue as the hidden cost of using free data.

We compute the forward returns as follows, removing outliers with weekly returns below 50 or above 100 percent:

```

fwd_return = []
for filing in filings:
    date_filed = filing_index.at[filing, 'date_filed']
    price_data = prices[prices.filing==filing].close.sort_index()

    try:
        r = (price_data
              .pct_change(periods=5)
              .shift(-5)
              .loc[:date_filed]
              .iloc[-1])
    except:
        continue
    if not np.isnan(r) and -.5 < r < 1:
        fwd_return[filing] = r

```

This leaves us with 16,355 data points. Now we combine these outcomes with their matching filing sequences and convert the list of returns to a NumPy array:

```
y, X = [], []
for filing_id, fwd_ret in fwd_return.items():
    X.append(np.load(vector_path / f'{filing_id}.npy') + 2)
    y.append(fwd_ret)
y = np.array(y)
```

Finally, we create a 90:10 training/test split and use the `pad_sequences` function introduced in the first example in this section to generate fixed-length sequences of 20,000 elements each:

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.1)
X_train = pad_sequences(X_train,
                        truncating='pre',
                        padding='pre',
                        maxlen=maxlen)
X_test = pad_sequences(X_test,
                        truncating='pre',
                        padding='pre',
                        maxlen=maxlen)
X_train.shape, X_test.shape
((14719, 20000), (1636, 20000))
```

Building, training, and evaluating the RNN model

Now we can define our RNN architecture. The first layer learns the word embeddings. We define the embedding dimensions as previously, setting the following:

- The `input_dim` keyword to the size of the vocabulary
- The `output_dim` keyword to the size of each embedding

- The `input_length` parameter to how long each input sequence is going to be

For the recurrent layer, we use a bidirectional GRU unit that scans the text both forward and backward and concatenates the resulting output. We also add batch normalization and dropout for regularization with a five-unit dense layer before the linear output:

```
embedding_size = 100
input_dim = X_train.max() + 1
rnn = Sequential([
    Embedding(input_dim=input_dim,
              output_dim=embedding_size,
              input_length=maxlen,
              name='EMB'),
    BatchNormalization(name='BN1'),
    Bidirectional(GRU(32), name='BD1'),
    BatchNormalization(name='BN2'),
    Dropout(.1, name='DO1'),
    Dense(5, name='D'),
    Dense(1, activation='linear', name='OUT'))
```

The resulting model has over 2.5 million trainable parameters:

```
rnn.summary()
Layer (type)          Output Shape       Param #
EMB (Embedding)      (None, 20000, 100)    2500000
BN1 (BatchNormalization) (None, 20000, 100)    400
BD1 (Bidirectional)   (None, 64)          25728
BN2 (BatchNormalization) (None, 64)          256
DO1 (Dropout)         (None, 64)          0
D (Dense)             (None, 5)           325
OUT (Dense)           (None, 1)           6
Total params: 2,526,715
Trainable params: 2,526,387
Non-trainable params: 328
```

We compile using the Adam optimizer, targeting the mean squared loss for this regression task while also tracking the square root of the

loss and the mean absolute error as optional metrics:

```
rnn.compile(loss='mse',
            optimizer='Adam',
            metrics=[RootMeanSquaredError(name='RMSE'),
                     MeanAbsoluteError(name='MAE')])
```

With early stopping, we train for up to 100 epochs on batches of 32 observations each:

```
early_stopping = EarlyStopping(monitor='val_MAE',
                               patience=5,
                               restore_best_weights=True)
training = rnn.fit(X_train,
                    y_train,
                    batch_size=32,
                    epochs=100,
                    validation_data=(X_test, y_test),
                    callbacks=[early_stopping],
                    verbose=1)
```

The mean absolute error improves for only 4 epochs, as shown in the left panel of *Figure 19.17*:

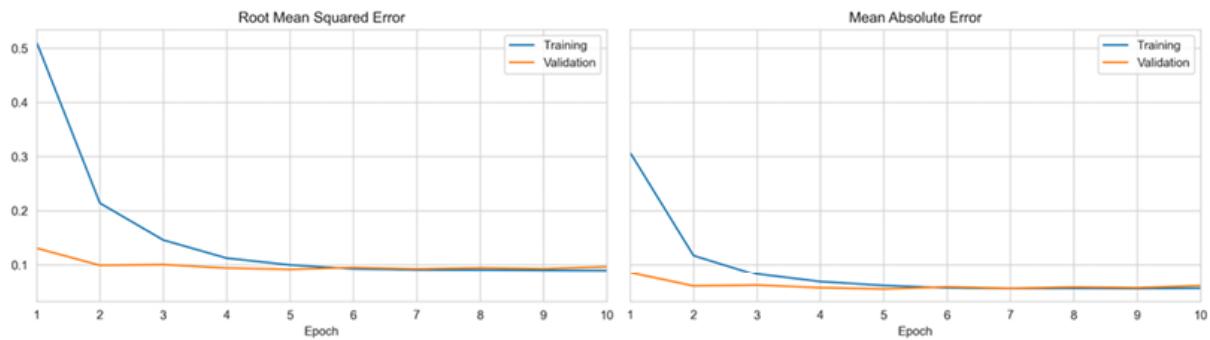


Figure 19.17: Cross-validation test results for RNNs using SEC filings to predict weekly returns

On the test set, the best model achieves a highly significant IC of 6.02:

```
y_score = rnn.predict(X_test)
rho, p = spearmanr(y_score.squeeze(), y_test)
print(f'{rho*100:.2f} ({p:.2%})')
6.02 (1.48%)
```

Lessons learned and next steps

The model is capable of generating return predictions that are significantly better than chance using only text data. There are both caveats that suggest taking the results with a grain of salt and reasons to believe we could improve on the result of this experiment.

On the one hand, the quality of both the stock price data and the parsed SEC filings is far from perfect. It's unclear whether price data issues bias the results positively or negatively, but they certainly increase the margin of error. More careful parsing and cleaning of the SEC filings would most likely improve the results by removing noise.

On the other hand, there are numerous optimizations that may well improve the result. Starting with the text input, we did not attempt to parse the filing content beyond selecting certain sections; there may be value in removing boilerplate language or otherwise trying to pick the most meaningful statements. We also made somewhat arbitrary choices about the maximum length of filings and the size of the vocabulary that we could revisit. We could also shorten or lengthen the weekly prediction horizon. Furthermore, there are multiple aspects of the model architecture that we could refine, from the size of the embeddings to the number and size of layers and the degree of regularization.

Most fundamentally, we could combine the text input with a richer set of complementary features, as demonstrated in the previous

section, using stacked LSTM with multiple inputs. Finally, we would certainly want a larger set of filings.

Summary

In this chapter, we presented the specialized RNN architecture that is tailored to sequential data. We covered how RNNs work, analyzed the computational graph, and saw how RNNs enable parameter-sharing over numerous steps to capture long-range dependencies that FFNNs and CNNs are not well suited for.

We also reviewed the challenges of vanishing and exploding gradients and saw how gated units like long short-term memory cells enable RNNs to learn dependencies over hundreds of time steps. Finally, we applied RNNs to challenges common in algorithmic trading, such as predicting univariate and multivariate time series and sentiment analysis using SEC filings.

In the next chapter, we will introduce unsupervised deep learning techniques like autoencoders and generative adversarial networks and their applications to investment and trading strategies.

Autoencoders for Conditional Risk Factors and Asset Pricing

This chapter shows how unsupervised learning can leverage deep learning for trading. More specifically, we'll discuss **autoencoders** that have been around for decades but have recently attracted fresh interest.

Unsupervised learning addresses practical ML challenges such as the limited availability of labeled data and the curse of dimensionality, which requires exponentially more samples for successful learning from complex, real-life data with many features. At a conceptual level, unsupervised learning resembles human learning and the development of common sense much more closely than supervised and reinforcement learning, which we'll cover in the next chapter. It is also called **predictive learning** because it aims to discover structure and regularities from data so that it can predict missing inputs, that is, fill in the blanks from the observed parts.

An **autoencoder** is a **neural network** (NN) trained to reproduce the input while learning a new representation of the data, encoded by the parameters of a hidden layer. Autoencoders have long been used for nonlinear dimensionality reduction and manifold learning (see *Chapter 13, Data-Driven Risk Factors and Asset Allocation with Unsupervised Learning*). A variety of designs leverage the feedforward, convolutional, and recurrent network architectures we

covered in the last three chapters. We will see how autoencoders can underpin a **trading strategy**: we will build a deep neural network that uses an autoencoder to extract risk factors and predict equity returns, conditioned on a range of equity attributes (Gu, Kelly, and Xiu 2020).

More specifically, in this chapter you will learn about:

- Which types of autoencoders are of practical use and how they work
- Building and training autoencoders using Python
- Using autoencoders to extract data-driven risk factors that take into account asset characteristics to predict returns



You can find the code samples for this chapter and links to additional resources in the corresponding directory of the GitHub repository. The notebooks include color versions of the images.

Autoencoders for nonlinear feature extraction

In *Chapter 17, Deep Learning for Trading*, we saw how neural networks succeed at supervised learning by extracting a hierarchical feature representation useful for the given task. **Convolutional neural networks (CNNs)**, for example, learn and synthesize increasingly complex patterns from grid-like data, for example, to identify or detect objects in an image or to classify time series.

An autoencoder, in contrast, is a neural network designed exclusively to learn a **new representation** that encodes the input in a way that helps solve another task. To this end, the training forces the network to reproduce the input. Since autoencoders typically use the same data as input and output, they are also considered an instance of **self-supervised learning**. In the process, the parameters of a hidden layer h become the code that represents the input, similar to the word2vec model covered in *Chapter 16, Word Embeddings for Earnings Calls and SEC Filings*.

More specifically, the network can be viewed as consisting of an encoder function $h=f(x)$ that learns the hidden layer's parameters from input x , and a decoder function g that learns to reconstruct the input from the encoding h . Rather than learning the identity function:

$$x = g(f(x))$$

which simply copies the input, autoencoders use **constraints** that force the hidden layer to **prioritize which aspects of the data to encode**. The goal is to obtain a representation of practical value.

Autoencoders can also be viewed as a **special case of a feedforward neural network** (see *Chapter 17, Deep Learning for Trading*) and can be trained using the same techniques. Just as with other models, excess capacity will lead to overfitting, preventing the autoencoder from producing an informative encoding that generalizes beyond the training samples. See *Chapters 14 and 15* of Goodfellow, Bengio, and Courville (2016) for additional background.

Generalizing linear dimensionality reduction

A traditional use case includes dimensionality reduction, achieved by limiting the size of the hidden layer and thus creating a "bottleneck" so that it performs lossy compression. Such an autoencoder is called **undercomplete**, and the purpose is to learn the most salient properties of the data by minimizing a loss function L of the form:

$$L(x, g(f(x)))$$

An example loss function that we will explore in the next section is simply the mean squared error evaluated on the pixel values of the input images and their reconstruction. We will also use this loss function to extract risk factors from time series of financial features when we build a conditional autoencoder for trading.

Undercomplete autoencoders differ from linear dimensionality reduction methods like **principal component analysis (PCA)**; see *Chapter 13, Data-Driven Risk Factors and Asset Allocation with Unsupervised Learning*) when they use **nonlinear activation functions**; otherwise, they learn the same subspace as PCA. They can thus be viewed as a nonlinear generalization of PCA capable of learning a wider range of encodings.

Figure 20.1 illustrates the encoder-decoder logic of an undercomplete feedforward autoencoder with three hidden layers: the encoder and decoder have one hidden layer each plus a shared encoder output/decoder input layer containing the encoding. The three hidden layers use nonlinear activation functions, like **rectified linear units (ReLU)**, *sigmoid*, or *tanh* (see *Chapter 17, Deep Learning for Trading*) and have fewer units than the input that the network aims to reconstruct.

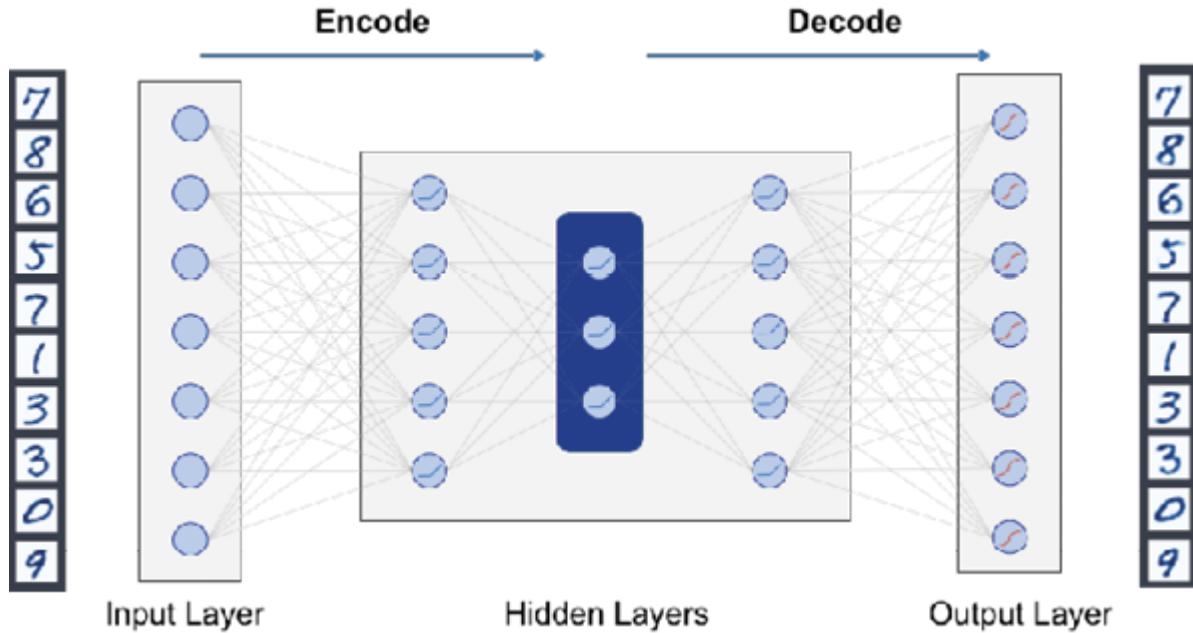


Figure 20.1: Undercomplete encoder-decoder architecture

Depending on the task, a simple autoencoder with a single encoder and decoder layer may be adequate. However, **deeper autoencoders** with additional layers can have several advantages, just as for other neural networks. These advantages include the ability to learn more complex encodings, achieve better compression, and do so with less computational effort and fewer training samples, subject to the perennial risk of overfitting.

Convolutional autoencoders for image compression

As discussed in *Chapter 18, CNNs for Financial Time Series and Satellite Images*, fully connected feedforward architectures are not well suited to capture local correlations typical to data with a grid-like structure. Instead, autoencoders can also use convolutional layers to learn a hierarchical feature representation. Convolutional autoencoders

leverage convolutions and parameter sharing to learn hierarchical patterns and features irrespective of their location, translation, or changes in size.

We will illustrate different implementations of convolutional autoencoders for image data below. Alternatively, convolutional autoencoders could be applied to multivariate time series data arranged in a grid-like format as illustrated in *Chapter 18, CNNs for Financial Time Series and Satellite Images*.

Managing overfitting with regularized autoencoders

The powerful capabilities of neural networks to represent complex functions require tight controls of the capacity of encoders and decoders to extract signals rather than noise so that the encoding is more useful for a downstream task. In other words, when it is too easy for the network to recreate the input, it fails to learn only the most interesting aspects of the data and improve the performance of a machine learning model that uses the encoding as inputs.

Just as for other models with excessive capacity for the given task, **regularization** can help to address the **overfitting** challenge by constraining the autoencoder's learning process and forcing it to produce a useful representation (see, for instance, *Chapter 7, Linear Models – From Risk Factors to Return Forecasts*, on regularization for linear models, and *Chapter 17, Deep Learning for Trading*, for neural networks). Ideally, we could precisely match the model's capacity to the complexity of the distribution of the data. In practice, the optimal model often combines (limited) excess capacity with appropriate regularization. To this end, we add a sparsity penalty $\Omega(h)$ that

depends on the weights of the encoding layer h to the training objective:

$$L(x, g(f(x))) + \Omega(h)$$

A common approach that we explore later in this chapter is the use of **L1 regularization**, which adds a penalty to the loss function in the form of the sum of the absolute values of the weights. The L1 norm results in sparse encodings because it forces the values of parameters to zero if they do not capture independent variation in the data (see *Chapter 7, Linear Models – From Risk Factors to Return Forecasts*). As a result, even overcomplete autoencoders with hidden layers of a higher dimension than the input may be able to learn signal content.

Fixing corrupted data with denoising autoencoders

The autoencoders we have discussed so far are designed to reproduce the input despite capacity constraints. An alternative approach trains autoencoders with corrupted inputs \tilde{x} to output the desired, original data points. In this case, the autoencoder minimizes a loss L :

$$L(x, g(f(\tilde{x})))$$

Corrupted inputs are a different way of preventing the network from learning the identity function and rather extracting the signal or salient features from the data. Denoising autoencoders have been shown to learn the data generating process of the original data and have become popular in generative modeling where the goal is to

learn the probability distribution that gives rise to the input (Vincent et al., 2008).

Seq2seq autoencoders for time series features

Recurrent neural networks (RNNs) have been developed for sequential data characterized by longitudinal dependencies between data points, potentially over long ranges (*Chapter 19, RNNs for Multivariate Time Series and Sentiment Analysis*). Similarly, sequence-to-sequence (seq2seq) autoencoders aim to learn representations attuned to the nature of data generated in sequence (Srivastava, Mansimov, and Salakhutdinov, 2016).

Seq2seq autoencoders are based on RNN components like **long short-term memory (LSTM)** or gated recurrent unit. They learn a representation of sequential data and have been successfully applied to video, text, audio, and time series data.

As mentioned in the last chapter, encoder-decoder architectures allow RNNs to process input and output sequences with variable length. These architectures underpin many advances in complex sequence prediction tasks, like speech recognition and text translation, and are being increasingly applied to (financial) time series. At a high level, they work as follows:

1. The LSTM encoder processes the input sequence step by step to learn a hidden state.
2. This state becomes a learned representation of the sequence in the form of a fixed-length vector.

3. The LSTM decoder receives this state as input and uses it to generate the output sequence.

See references linked on GitHub for examples on building sequence-to-sequence autoencoders to **compress time series data** and **detect anomalies** in time series to allow, for example, regulators to uncover potentially illegal trading activity.

Generative modeling with variational autoencoders

Variational autoencoders (VAE) were developed more recently (Kingma and Welling, 2014) and focus on generative modeling. In contrast to a discriminative model that learns a predictor given data, a generative model aims to solve the more general problem of learning a joint probability distribution over all variables. If successful, it could simulate how the data is produced in the first place. Learning the data-generating process is very valuable: it reveals underlying causal relationships and supports semi-supervised learning to effectively generalize from a small labeled dataset to a large unlabeled one.

More specifically, VAEs are designed to learn the latent (meaning *unobserved*) variables of the model responsible for the input data. Note that we encountered latent variables in *Chapter 15, Topic Modeling – Summarizing Financial News*, and *Chapter 16, Word Embeddings for Earnings Calls and SEC Filings*.

Just like the autoencoders discussed so far, VAEs do not let the network learn arbitrary functions as long as it faithfully reproduces the input. Instead, they aim to learn the parameters of a probability distribution that generates the input data.

In other words, VAEs are generative models because, if successful, you can generate new data points by sampling from the distribution learned by the VAE.

The operation of a VAE is more complex than the autoencoders discussed so far because it involves stochastic backpropagation, that is, taking derivatives of stochastic variables, and the details are beyond the scope of this book. They are able to learn high-capacity input encodings without regularization that are useful because the models aim to maximize the probability of the training data rather than to reproduce the input. For a detailed introduction, see Kingma and Welling (2019).

The `variational_autoencoder.ipynb` notebook includes a sample VAE implementation applied to the Fashion MNIST data, adapted from a Keras tutorial by Francois Chollet to work with TensorFlow 2. The resources linked on GitHub contain a VAE tutorial with references to PyTorch and TensorFlow 2 implementations and many additional references. See Wang et al. (2019) for an application that combines a VAE with an RNN using LSTM and outperforms various benchmark models in futures markets.

Implementing autoencoders with TensorFlow 2

In this section, we'll illustrate how to implement several of the autoencoder models introduced in the previous section using the Keras interface of TensorFlow 2. We'll first load and prepare an image dataset that we'll use throughout this section. We will use images instead of financial time series because it makes it easier to

visualize the results of the encoding process. The next section shows how to use an autoencoder with financial data as part of a more complex architecture that can serve as the basis for a trading strategy.

After preparing the data, we'll proceed to build autoencoders using deep feedforward nets, sparsity constraints, and convolutions and apply the latter to denoise images.

How to prepare the data

For illustration, we'll use the Fashion MNIST dataset, a modern drop-in replacement for the classic MNIST handwritten digit dataset popularized by Lecun et al. (1998) with LeNet. We also relied on this dataset in *Chapter 13, Data-Driven Risk Factors and Asset Allocation with Unsupervised Learning*, on unsupervised learning.

Keras makes it easy to access the 60,000 training and 10,000 test grayscale samples with a resolution of 28×28 pixels:

```
from tensorflow.keras.datasets import fashion_mnist
(x_train, y_train), (X_test, y_test) = fashion_mnist.load_data()
X_train.shape, X_test.shape
((60000, 28, 28), (10000, 28, 28))
```

The data contains clothing items from 10 classes. *Figure 20.2* plots a sample image for each class:

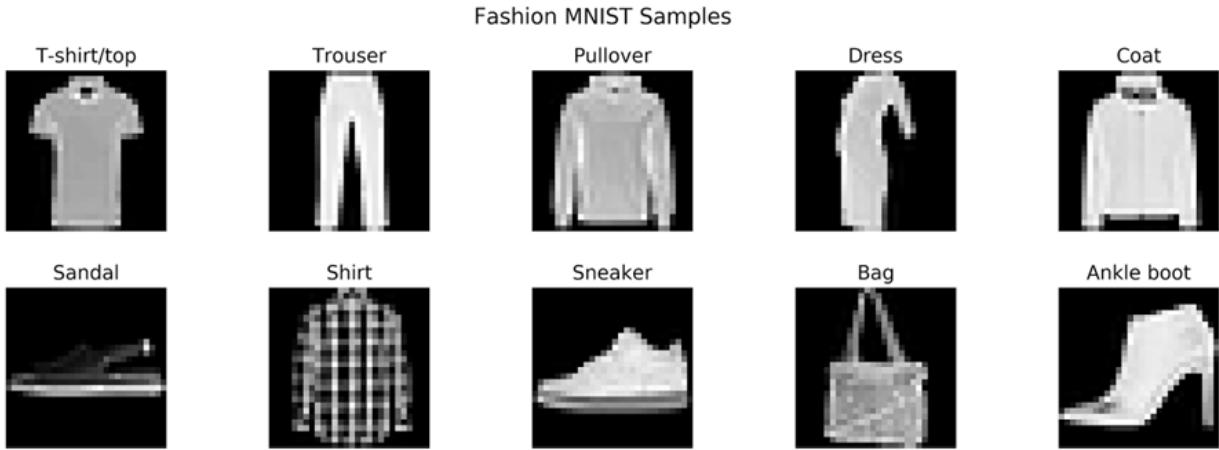


Figure 20.2: Fashion MNIST sample images

We reshape the data so that each image is represented by a flat one-dimensional pixel vector with $28 \times 28 = 784$ elements normalized to the range $[0, 1]$:

```
image_size = 28          # pixels per side
input_size = image_size ** 2 # 784
def data_prep(x, size=input_size):
    return x.reshape(-1, size).astype('float32')/255
X_train_scaled = data_prep(X_train)
X_test_scaled = data_prep(X_test)
X_train_scaled.shape, X_test_scaled.shape
((60000, 784), (10000, 784))
```

One-layer feedforward autoencoder

We start with a vanilla feedforward autoencoder with a single hidden layer to illustrate the general design approach using the Functional Keras API and establish a performance baseline.

The first step is a placeholder for the flattened image vectors with 784 elements:

```
input_ = Input(shape=(input_size,), name='Input')
```

The encoder part of the model consists of a fully connected layer that learns the new, compressed representation of the input. We use 32 units for a compression ratio of 24.5:

```
encoding_size = 32 # compression factor: 784 / 32 = 24.5
encoding = Dense(units=encoding_size,
                  activation='relu',
                  name='Encoder')(input_)
```

The decoding part reconstructs the compressed data to its original size in a single step:

```
decoding = Dense(units=input_size,
                  activation='sigmoid',
                  name='Decoder')(encoding)
```

We instantiate the `Model` class with the chained input and output elements that implicitly define the computational graph as follows:

```
autoencoder = Model(inputs=input_,
                     outputs=decoding,
                     name='Autoencoder')
```

The encoder-decoder computation thus defined uses almost 51,000 parameters:

Layer (type)	Output Shape	Param #
Input (InputLayer)	(None, 784)	0
Encoder (Dense)	(None, 32)	25120
Decoder (Dense)	(None, 784)	25872
Total params: 50,992		
Trainable params: 50,992		
Non-trainable params: 0		

The Functional API allows us to use parts of the model's chain as separate encoder and decoder models that use the autoencoder's parameters learned during training.

Defining the encoder

The encoder just uses the input and hidden layer with about half the total parameters:

```
encoder = Model(inputs=input_, outputs=encoding, name='Encoder')
encoder.summary()
Layer (type)          Output Shape         Param #
Input (InputLayer)    (None, 784)           0
Encoder (Dense)       (None, 32)            25120
Total params: 25,120
Trainable params: 25,120
Non-trainable params: 0
```

We will see shortly that, once we train the autoencoder, we can use the encoder to compress the data.

Defining the decoder

The decoder consists of the last autoencoder layer, fed by a placeholder for the encoded data:

```
encoded_input = Input(shape=(encoding_size,), name='Decoder_Input')
decoder_layer = autoencoder.layers[-1](encoded_input)
decoder = Model(inputs=encoded_input, outputs=decoder_layer)
decoder.summary()
Layer (type)          Output Shape         Param #
Decoder_Input (InputLayer) (None, 32)           0
Decoder (Dense)       (None, 784)            25872
Total params: 25,872
Trainable params: 25,872
Non-trainable params: 0
```

Training the model

We compile the model to use the Adam optimizer (see *Chapter 17, Deep Learning for Trading*) to minimize the mean squared error between the input data and the reproduction achieved by the autoencoder. To ensure that the autoencoder learns to reproduce the input, we train the model using the same input and output data:

```
autoencoder.compile(optimizer='adam', loss='mse')
autoencoder.fit(x=X_train_scaled, y=X_train_scaled,
                 epochs=100, batch_size=32,
                 shuffle=True, validation_split=.1,
                 callbacks=[tb_callback, early_stopping, checkpointer])
```

Evaluating the results

Training stops after some 20 epochs with a test RMSE of 0.1121:

```
mse = autoencoder.evaluate(x=X_test_scaled, y=X_test_scaled)
f'MSE: {mse:.4f} | RMSE {mse**.5:.4f}'
'MSE: 0.0126 | RMSE 0.1121'
```

To encode data, we use the encoder we just defined like so:

```
encoded_test_img = encoder.predict(X_test_scaled)
Encoded_test_img.shape
(10000, 32)
```

The decoder takes the compressed data and reproduces the output according to the autoencoder training results:

```
decoded_test_img = decoder.predict(encoded_test_img)
decoded_test_img.shape
(10000, 784)
```

Figure 20.3 shows 10 original images and their reconstruction by the autoencoder and illustrates the loss after compression:

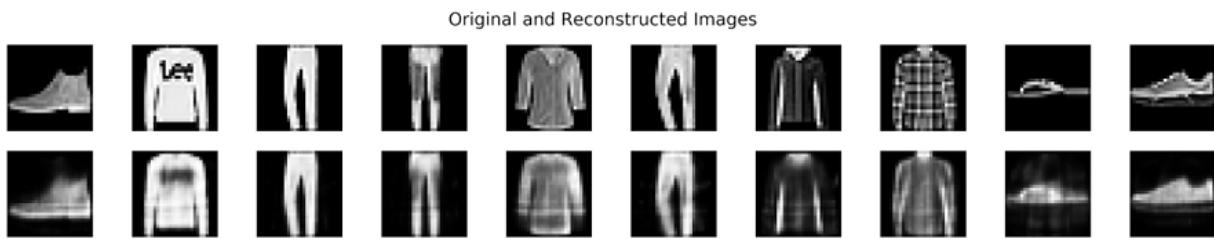


Figure 20.3: Sample Fashion MNIST images, original and reconstructed

Feedforward autoencoder with sparsity constraints

The addition of regularization is fairly straightforward. We can apply it to the dense encoder layer using Keras' `activity_regularizer` as follows:

```
encoding_l1 = Dense(units=encoding_size,
                     activation='relu',
                     activity_regularizer=regularizers.l1(10e-5),
                     name='Encoder_L1')(input_)
```

The input and decoding layers remain unchanged. In this example with compression of factor 24.5, regularization negatively affects performance with a test RMSE of 0.1229.

Deep feedforward autoencoder

To illustrate the benefit of adding depth to the autoencoder, we will build a three-layer feedforward model that successively compresses the input from 784 to 128, 64, and 32 units, respectively:

```

input_ = Input(shape=(input_size,))
x = Dense(128, activation='relu', name='Encoding1')(input_)
x = Dense(64, activation='relu', name='Encoding2')(x)
encoding_deep = Dense(32, activation='relu', name='Encoding3')(x)
x = Dense(64, activation='relu', name='Decoding1')(encoding_deep)
x = Dense(128, activation='relu', name='Decoding2')(x)
decoding_deep = Dense(input_size, activation='sigmoid', name='Decoding')
autoencoder_deep = Model(input_, decoding_deep)

```

The resulting model has over 222,000 parameters, more than four times the capacity of the previous single-layer model:

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 784)	0
Encoding1 (Dense)	(None, 128)	100480
Encoding2 (Dense)	(None, 64)	8256
Encoding3 (Dense)	(None, 32)	2080
Decoding1 (Dense)	(None, 64)	2112
Decoding2 (Dense)	(None, 128)	8320
Decoding3 (Dense)	(None, 784)	101136
<hr/>		
Total params: 222,384		
Trainable params: 222,384		
Non-trainable params: 0		

Training stops after 45 epochs and results in a 14 percent reduction of the test RMSE to 0.097. Due to the low resolution, it is difficult to visually note the better reconstruction.

Visualizing the encoding

We can use the manifold learning technique **t-distributed Stochastic Neighbor Embedding (t-SNE)**; see *Chapter 13, Data-Driven Risk Factors and Asset Allocation with Unsupervised Learning*) to visualize and assess the quality of the encoding learned by the autoencoder's hidden layer.

If the encoding is successful in capturing the salient features of the data, then the compressed representation of the data should still reveal a structure aligned with the 10 classes that differentiate the observations. We use the output of the deep encoder we just trained to obtain the 32-dimensional representation of the test set:

```
tsne = TSNE(perplexity=25, n_iter=5000)
train_embed = tsne.fit_transform(encoder_deep.predict(X_train_scaled))
```

Figure 20.4 shows that the 10 classes are well separated, suggesting that the encoding is useful as a lower-dimensional representation that preserves the key characteristics of the data (see the `variational_autoencoder.ipynb` notebook for a color version):

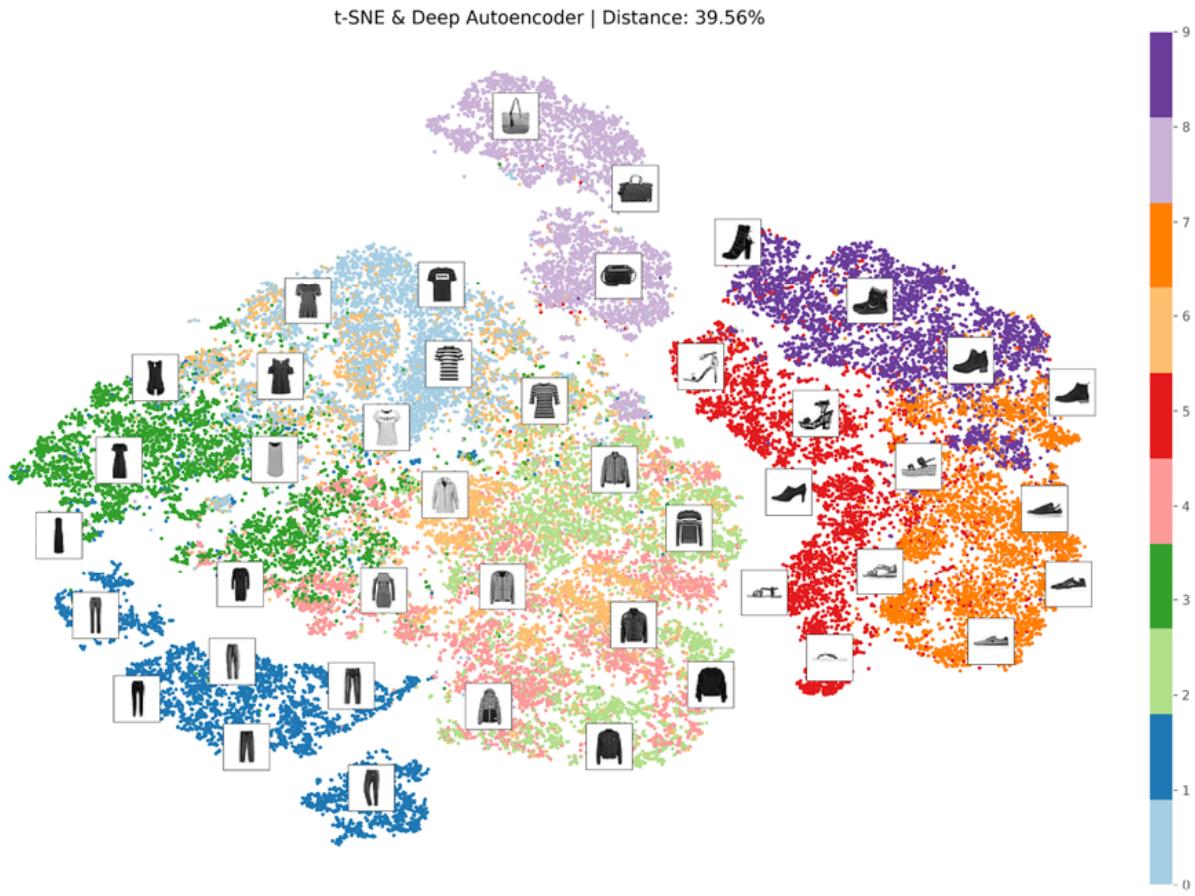


Figure 20.4: t-SNE visualization of the Fashion MNIST autoencoder embedding

Convolutional autoencoders

The insights from *Chapter 18, CNNs for Financial Time Series and Satellite Images*, on CNNs suggest we incorporate convolutional layers into the autoencoder to extract information characteristic of the grid-like structure of image data.

We define a three-layer encoder that uses 2D convolutions with 32, 16, and 8 filters, respectively, ReLU activations, and `'same'` padding to maintain the input size. The resulting encoding size at the third layer is $4 \times 4 \times 8 = 128$, higher than for the previous examples:

```

x = Conv2D(filters=32,
           kernel_size=(3, 3),
           activation='relu',
           padding='same',
           name='Encoding_Conv_1')(input_)
x = MaxPooling2D(pool_size=(2, 2), padding='same', name='Encoding_Max_1')(x)
x = Conv2D(filters=16,
           kernel_size=(3, 3),
           activation='relu',
           padding='same',
           name='Encoding_Conv_2')(x)
x = MaxPooling2D(pool_size=(2, 2), padding='same', name='Encoding_Max_2')(x)
x = Conv2D(filters=8,
           kernel_size=(3, 3),
           activation='relu',
           padding='same',
           name='Encoding_Conv_3')(x)
encoded_conv = MaxPooling2D(pool_size=(2, 2),
                            padding='same',
                            name='Encoding_Max_3')(x)

```

We also define a matching decoder that reverses the number of filters and uses 2D upsampling instead of max pooling to reverse the reduction of the filter sizes. The three-layer autoencoder has 12,785 parameters, a little more than 5 percent of the capacity of the deep autoencoder.

Training stops after 67 epochs and results in a further 9 percent reduction in the test RMSE, due to a combination of the ability of convolutional filters to learn more efficiently from image data and the larger encoding size.

Denoising autoencoders

The application of an autoencoder to a denoising task only affects the training stage. In this example, we add noise from a standard

normal distribution to the Fashion MNIST data while maintaining the pixel values in the range [0, 1] as follows:

```
def add_noise(x, noise_factor=.3):
    return np.clip(x + noise_factor * np.random.normal(size=x.shape),
X_train_noisy = add_noise(X_train_scaled)
X_test_noisy = add_noise(X_test_scaled)
```

We then proceed to train the convolutional autoencoder on noisy inputs, the objective being to learn how to generate the uncorrupted originals:

```
autoencoder_denoise.fit(x=X_train_noisy,
                        y=X_train_scaled,
                        ...)
```

The test RMSE after 60 epochs is 0.0931, unsurprisingly higher than before. *Figure 20.5* shows, from top to bottom, the original images as well as the noisy and denoised versions. It illustrates that the autoencoder is successful in producing compressed encodings from the noisy images that are quite similar to those produced from the original images:

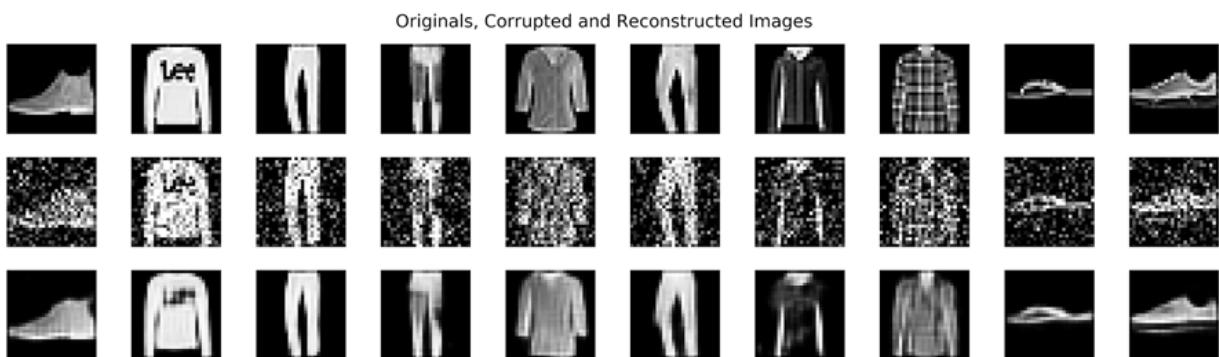


Figure 20.5: Denoising input and output examples

A conditional autoencoder for trading

Recent research by Gu, Kelly, and Xiu (GKX, 2019) developed an asset pricing model based on the exposure of securities to risk factors. It builds on the concept of **data-driven risk factors** that we discussed in *Chapter 13, Data-Driven Risk Factors and Asset Allocation with Unsupervised Learning*, when introducing PCA as well as the risk factor models covered in *Chapter 4, Financial Feature Engineering – How to Research Alpha Factors*. They aim to show that the asset characteristics used by factor models to capture the systematic drivers of "anomalies" are just proxies for the time-varying exposure to risk factors that cannot be directly measured. In this context, anomalies are returns in excess of those explained by the exposure to aggregate market risk (see the discussion of the capital asset pricing model in *Chapter 5, Portfolio Optimization and Performance Evaluation*).

The **Fama-French factor models** discussed in *Chapter 4* and *Chapter 7* explain returns by specifying risk factors like firm size based on empirical observations of differences in average stock returns beyond those due to aggregate market risk. Given such **specific risk factors**, these models are able to measure the reward an investor receives for taking on factor risk using portfolios designed accordingly: sort stocks by size, buy the smallest quintile, sell the largest quintile, and compute the return. The observed risk factor return then allows linear regression to estimate the sensitivity of assets to these factors (called **factor loadings**), which in turn helps to predict the returns of (many) assets based on forecasts of (far fewer) factor returns.

In contrast, GKX treat **risk factors as latent, or non-observable**, drivers of covariance among a number of assets large enough to prevent investors from avoiding exposure through diversification. Therefore, investors require a reward that adjusts like any price to achieve equilibrium, providing in turn an economic rationale for return differences that are no longer anomalous. In this view, risk factors are purely statistical in nature while the underlying economic forces can be of arbitrary and varying origin.

In another recent paper (Kelly, Pruitt, and Su, 2019), Kelly—who teaches finance at Yale, works with AQR, and is one of the pioneers in applying ML to trading—and his coauthors developed a linear model dubbed **Instrumented Principal Component Analysis** (IPCA) to **estimate latent risk factors and the assets' factor loadings from data**. IPCA extends PCA to include asset characteristics as covariates and produce time-varying factor loadings. (See *Chapter 13, Data-Driven Risk Factors and Asset Allocation with Unsupervised Learning*, for coverage of PCA.) By conditioning asset exposure to factors on observable asset characteristics, IPCA aims to answer whether there is a set of common latent risk factors that explain an observed anomaly rather than whether there is a specific observable factor that can do so.

GKX creates a **conditional autoencoder architecture** to reflect the nonlinear nature of return dynamics ignored by the linear Fama-French models and the IPCA approach. The result is a deep neural network that simultaneously learns the premia on a given number of unobservable factors using an autoencoder, and the factor loadings for a large universe of equities based on a broad range of time-varying asset characteristics using a feedforward network. The model succeeds in explaining and predicting asset returns. It demonstrates a relationship that is both statistically and

economically significant, yielding an attractive Sharpe ratio when translated into a long-short decile spread strategy similar to the examples we have used throughout this book.

In this section, we'll create a simplified version of this model to demonstrate how you can **leverage autoencoders to generate tradeable signals**. To this end, we'll build a new dataset of close to 4,000 US stocks over the 1990-2019 period using `yfinance`, because it provides some additional information that facilitates the computation of the asset characteristics. We'll take a few shortcuts, such as using fewer assets and only the most important characteristics. We'll also omit some implementation details to simplify the exposition. We'll highlight the most important differences so that you can enhance the model accordingly.

We'll first show how to prepare the data before we explain, build, and train the model and evaluate its predictive performance. Please see the above references for additional background on the theory and implementation.

Sourcing stock prices and metadata information

The GKX reference implementation uses stock price and firm characteristic data for over 30,000 US equities from the Center for Research in Security Prices (CRSP) from 1957-2016 at a monthly frequency. It computes 94 metrics that include a broad range of asset attributes suggested as predictive of returns in previous academic research and listed in Green, Hand, and Zhang (2017), who set out to verify these claims.

Since we do not have access to the high-quality but costly CRSP data, we leverage yfinance (see *Chapter 2, Market and Fundamental Data – Sources and Techniques*) to download price and metadata from Yahoo Finance. There are downsides to choosing free data, including:

- The lack of quality control regarding adjustments
- Survivorship bias because we cannot get data for stocks that are no longer listed
- A smaller scope in terms of both the number of equities and the length of their history

The `build_us_stock_dataset.ipynb` notebook contains the relevant code examples for this section.

To obtain the data, we get a list of the 8,882 currently traded symbols from NASDAQ using pandas-datareader (see *Chapter 2, Market and Fundamental Data – Sources and Techniques*):

```
from pandas_datareader.nasdaq_trader import get_nasdaq_symbols
traded_symbols = get_nasdaq_symbols()
```

We remove ETFs and create yfinance `Ticker()` objects for the remainder:

```
import yfinance as yf
tickers = yf.Tickers(traded_symbols[~traded_symbols.ETF].index.to_list)
```

Each ticker's `.info` attribute contains data points scraped from Yahoo Finance, ranging from the outstanding number of shares and other fundamentals to the latest market capitalization; coverage varies by security:

```

info = []
for ticker in tickers.tickers:
    info.append(pd.Series(ticker.info).to_frame(ticker.ticker))
info = pd.concat(info, axis=1).dropna(how='all').T
info = info.apply(pd.to_numeric, errors='ignore')

```

For the tickers with metadata, we download both adjusted and unadjusted prices, the latter including corporate actions like stock splits and dividend payments that we could use to create a Zipline bundle for strategy backtesting (see *Chapter 8, The ML4T Workflow – From Model to Strategy Backtesting*).

We get adjusted OHLCV data on 4,314 stocks as follows:

```

prices_adj = []
with pd.HDFStore('chunks.h5') as store:
    for i, chunk in enumerate(chunks(tickers, 100)):
        print(i, end=' ', flush=True)
        prices_adj.append(yf.download(chunk,
                                      period='max',
                                      auto_adjust=True).stack(-1))
prices_adj = (pd.concat(prices_adj)
              .dropna(how='all', axis=1)
              .rename(columns=str.lower)
              .swaplevel())
prices_adj.index.names = ['ticker', 'date']

```

Absent any quality control regarding the underlying price data and the adjustments for stock splits, we remove equities with suspicious values such as daily returns above 100 percent or below -100 percent:

```

df = prices_adj.close.unstack('ticker')
pmax = df.pct_change().max()
pmin = df.pct_change().min()
to_drop = pmax[pmax > 1].index.union(pmin[pmin < -1].index)

```

This removes around 10 percent of the tickers, leaving us with close to 3,900 assets for the 1990-2019 period.

Computing predictive asset characteristics

GKX tested 94 asset attributes based on Green et al. (2017) and identified the 20 most influential metrics while asserting that feature importance drops off quickly thereafter. The top 20 stock characteristics fall into three categories, namely:

- **Price trend**, including (industry) momentum, short- and long-term reversal, or the recent maximum return
 - **Liquidity**, such as turnover, dollar volume, or market capitalization
 - **Risk measures**, for instance, total and idiosyncratic return volatility or market beta

Of these 20, we limit the analysis to 16 for which we have or can approximate the relevant inputs. The

`conditional_autoencoder_for_trading_data.ipynb` notebook demonstrates how to calculate the relevant metrics. We highlight a few examples in this section; see also the *Appendix, Alpha Factor Library*.

Some metrics require information like sector, market cap, and outstanding shares, so we limit our stock price dataset to the securities with relevant metadata:

We run our analysis at a weekly instead of monthly return frequency to compensate for the 50 percent shorter time period and around 80 percent lower number of stocks. We obtain weekly returns as follows:

```
returns = (prices.close
           .unstack('ticker')
           .resample('W-FRI').last()
           .sort_index().pct_change().iloc[1:])
```

Most metrics are fairly straightforward to compute. **Stock momentum**, the 11-month cumulative stock returns ending 1 month before the current date, can be derived as follows:

```
MONTH = 21
mom12m = (close
           .pct_change(periods=11 * MONTH)
           .shift(MONTH)
           .resample('W-FRI')
           .last()
           .stack()
           .to_frame('mom12m'))
```

The **Amihud Illiquidity** measure is the ratio of a stock's absolute returns relative to its dollar volume, measured as a rolling 21-day average:

```
dv = close.mul(volume)
ill = (close.pct_change().abs()
       .div(dv)
       .rolling(21)
       .mean()
       .resample('W-FRI').last()
       .stack()
       .to_frame('ill'))
```

Idiosyncratic volatility is measured as the standard deviation of a regression of residuals of weekly returns on the returns of equally weighted market index returns for the prior three years. We compute this computationally intensive metric using `statsmodels`:

```
index = close.resample('W-FRI').last().pct_change().mean(1).to_frame()
def get_ols_residuals(y, x=index):
    df = x.join(y.to_frame('y')).dropna()
    model = sm.OLS(endog=df.y, exog=sm.add_constant(df[['x']])))
    result = model.fit()
    return result.resid.std()
idiovol = (returns.apply(lambda x: x.rolling(3 * 52)
                        .apply(get_ols_residuals)))
```

◀ ▶

For the **market beta**, we can use `statsmodels`' `RollingOLS` class with the weekly asset returns as outcome and the equal-weighted index as input:

```
def get_market_beta(y, x=index):
    df = x.join(y.to_frame('y')).dropna()
    model = RollingOLS(endog=df.y,
                        exog=sm.add_constant(df[['x']])),
                        window=3*52)
    return model.fit(params_only=True).params['x']
beta = (returns.dropna(thresh=3*52, axis=1)
        .apply(get_market_beta).stack().to_frame('beta'))
```

We end up with around 3 million observations on 16 metrics for some 3,800 securities over the 1990-2019 period. *Figure 20.6* displays a histogram of the number of stock returns per week (the left panel) and boxplots outlining the distribution of the number of observations for each characteristic:

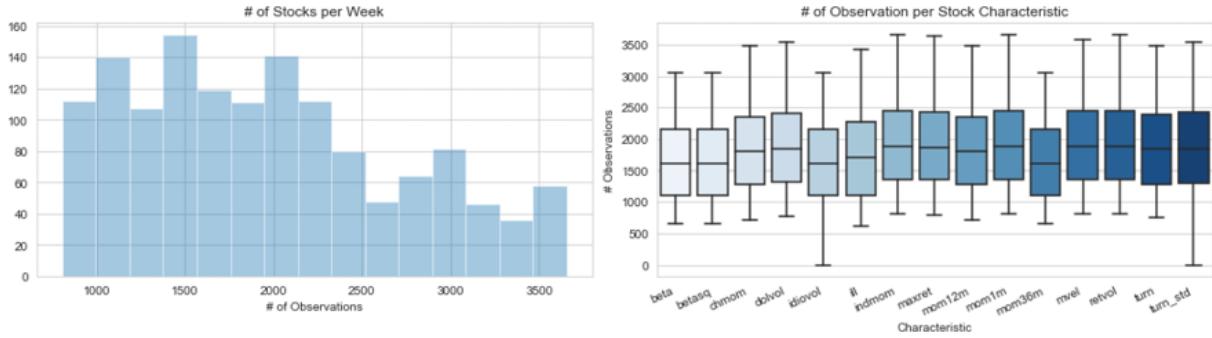


Figure 20.6: Number of tickers over time and per - stock characteristic

To limit the influence of outliers, we follow GKX and rank-normalize the characteristics to the $[-1, 1]$ interval:

```
data.loc[:, characteristics] = (data.loc[:, characteristics]
    .groupby(level='date')
    .apply(lambda x:
        pd.DataFrame(quantile_transform(
            x,
            copy=True,
            n_quantiles=x.shape[0]),
        columns=characteristics,
        index=x.index.get_level_values
    )
    .mul(2).sub(1))
```

Since the neural network cannot handle missing data, we set missing values to -2, which lies outside the range for both weekly returns and the characteristics.

The authors apply additional methods to avoid overweighting microcap stocks like market-value-weighted least-squares regression. They also adjust for data-snooping biases by factoring in conservative reporting lags for the characteristics.

Creating the conditional autoencoder architecture

The conditional autoencoder proposed by GKX allows for time-varying return distributions that take into account changing asset characteristics. To this end, the authors extend standard autoencoder architectures that we discussed in the first section of this chapter to allow for features to shape the encoding.

Figure 20.7 illustrates the architecture that models the outcome (asset returns, top) as a function of both asset characteristics (left input) and, again, individual asset returns (right input). The authors allow for asset returns to be individual stock returns or portfolios that are formed from the stocks in the sample based on the asset characteristics, similar to the Fama-French factor portfolios we discussed in *Chapter 4, Financial Feature Engineering – How to Research Alpha Factors*, and summarized in the introduction to this section (hence the dotted lines from stocks to portfolios in the lower-right box). We will use individual stock returns; see GKX for details on how and why to use portfolios instead.

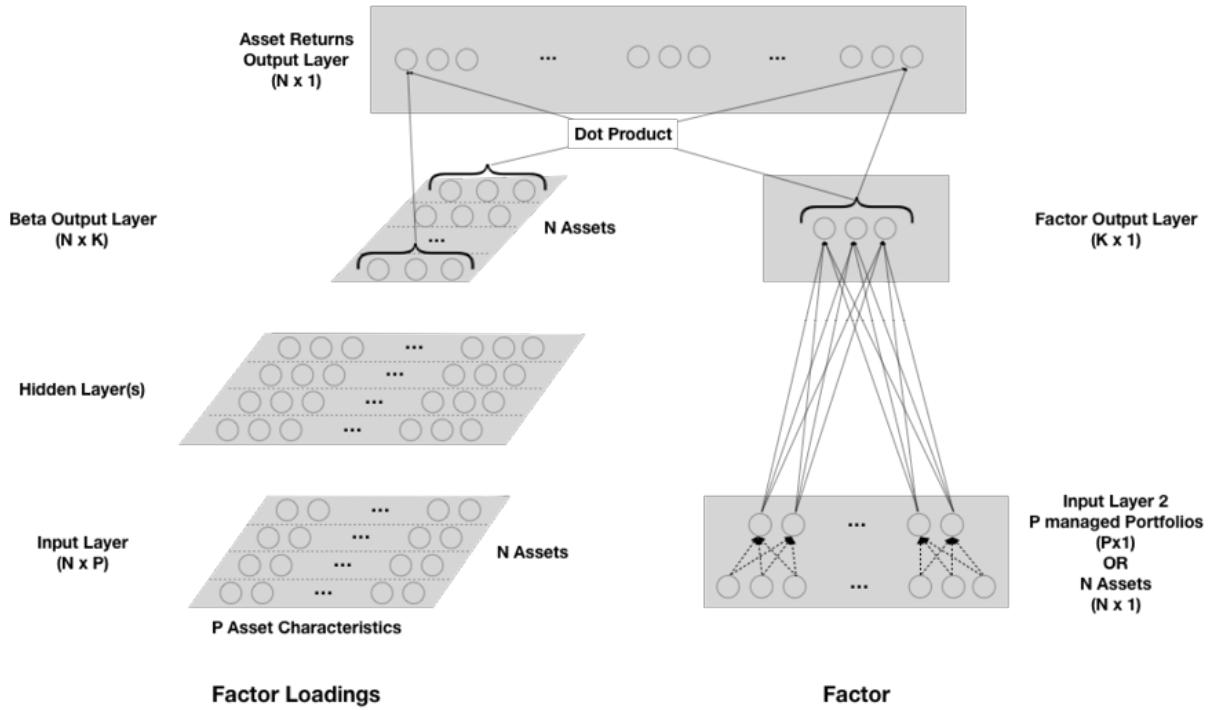


Figure 20.7: Conditional autoencoder architecture designed by GKX

The **feedforward neural network** on the left side of the conditional autoencoder models the K factor loadings (beta output) of N individual stocks as a function of their P characteristics (input). In our case, N is around 3,800 and P equals 16. The authors experiment with up to three hidden layers with 32, 16, and 8 units, respectively, and find two layers to perform best. Due to the smaller number of characteristics, we only use a similar layer and find 8 units most effective.

The right side of this architecture is a traditional autoencoder when used with individual asset returns as inputs because it maps N asset returns onto themselves. The authors use it in this way to measure how well the derived factors explain contemporaneous returns. In addition, they use the autoencoder to predict future returns by using input returns from period $t-1$ with output returns from period t . We will focus on the use of the architecture for prediction, underlining

that autoencoders are a special case of a feedforward neural network as mentioned in the first section of this chapter.

The model output is the dot product of the $N \times K$ factor loadings on the left with the $K \times 1$ factor premia on the right. The authors experiment with values of K in the range 2-6, similar to established factor models.

To create this architecture using TensorFlow 2, we use the Functional Keras API and define a `make_model()` function that automates the model compilation process as follows:

```
def make_model(hidden_units=8, n_factors=3):
    input_beta = Input((n_tickers, n_characteristics), name='input_beta')
    input_factor = Input((n_tickers,), name='input_factor')
    hidden_layer = Dense(units=hidden_units,
                          activation='relu',
                          name='hidden_layer')(input_beta)
    batch_norm = BatchNormalization(name='batch_norm')(hidden_layer)

    output_beta = Dense(units=n_factors, name='output_beta')(batch_norm)
    output_factor = Dense(units=n_factors,
                          name='output_factor')(input_factor)
    output = Dot(axes=(2,1),
                 name='output_layer')([output_beta, output_factor])
    model = Model(inputs=[input_beta, input_factor], outputs=output)
    model.compile(loss='mse', optimizer='adam')
    return model
```

We follow the authors in using batch normalization and compile the model to use mean squared error for this regression task and the Adam optimizer. This model has 12,418 parameters (see the notebook).

The authors use additional regularization techniques such as L1 penalties on network weights and combine the results of various

networks with the same architecture but using different random seeds. They also use early stopping.

We cross-validate using 20 years for training and predict the following year of weekly returns with five folds corresponding to the years 2015-2019. We evaluate combinations of numbers of factors K from 2 to 6 and 8, 16, or 32 hidden layer units by computing the **information coefficient (IC)** for the validation set as follows:

```
factor_opts = [2, 3, 4, 5, 6]
unit_opts = [8, 16, 32]
param_grid = list(product(unit_opts, factor_opts))
for units, n_factors in param_grid:
    scores = []
    model = make_model(hidden_units=units, n_factors=n_factors)
    for fold, (train_idx, val_idx) in enumerate(cv.split(data)):
        X1_train, X2_train, y_train, X1_val, X2_val, y_val = \
            get_train_valid_data(data, train_idx, val_idx)
        for epoch in range(250):
            model.fit([X1_train, X2_train], y_train,
                      batch_size=batch_size,
                      validation_data=([X1_val, X2_val], y_val),
                      epochs=epoch + 1,
                      initial_epoch=epoch,
                      verbose=0, shuffle=True)
        result = (pd.DataFrame({'y_pred': model.predict([X1_val,
                                                          X2_val]))
                               .reshape(-1),
                               'y_true': y_val.stack().values},
                               index=y_val.stack().index)
        .replace(-2, np.nan).dropna()
    r0 = spearmanr(result.y_true, result.y_pred)[0]
    r1 = result.groupby(level='date').apply(lambda x:
                                              spearmanr(x.y_pred,
                                                         x.y_true))
    scores.append([units, n_factors, fold, epoch, r0, r1.mean(
        r1.std(), r1.median())])

```

Figure 20.8 plots the validation IC averaged over the five annual folds by epoch for the five-factor count and three hidden-layer size

combinations. The upper panel shows the IC across the 52 weeks and the lower panel shows the average weekly IC (see the notebook for the color version):

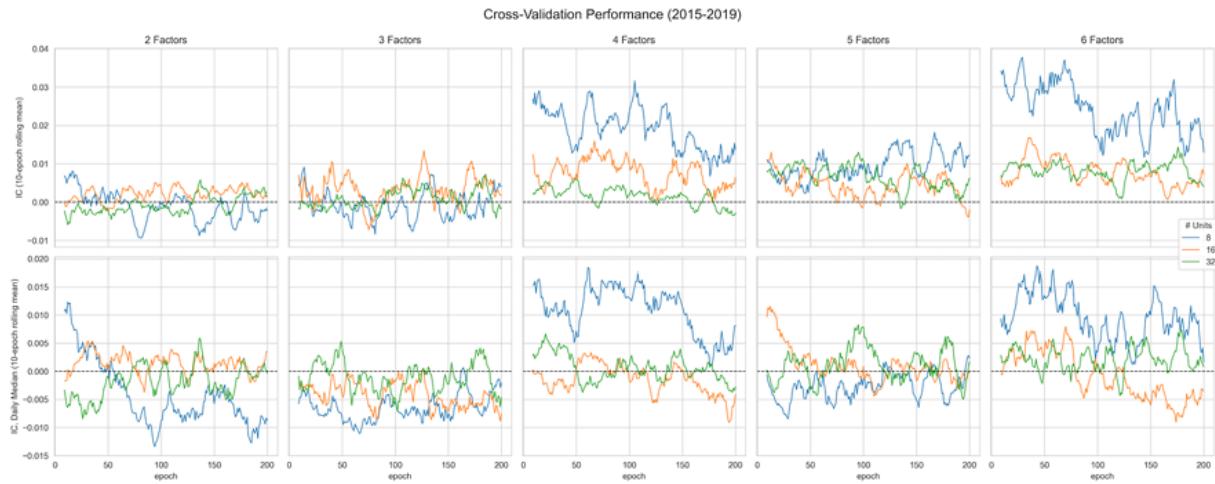


Figure 20.8: Cross-validation performance for all factor and hidden-layer size combinations

The results suggest that more factors and fewer hidden layer units work better; in particular, four and six factors with eight units perform best with overall IC values in the range of 0.02-0.03.

To evaluate the economic significance of the model's predictive performance, we generate predictions for a four-factor model with eight units trained for 15 epochs. Then we use Alphalens to compute the spreads between equal-weighted portfolios invested by a quintile of the predictions for each point in time, while ignoring transaction costs (see the `alphalens_analysis.ipynb` notebook).

Figure 20.9 shows the mean spread for holding periods from 5 to 21 days. For the shorter end that also reflects the prediction horizon, the spread between the bottom and the top decile is around 10 basis points:

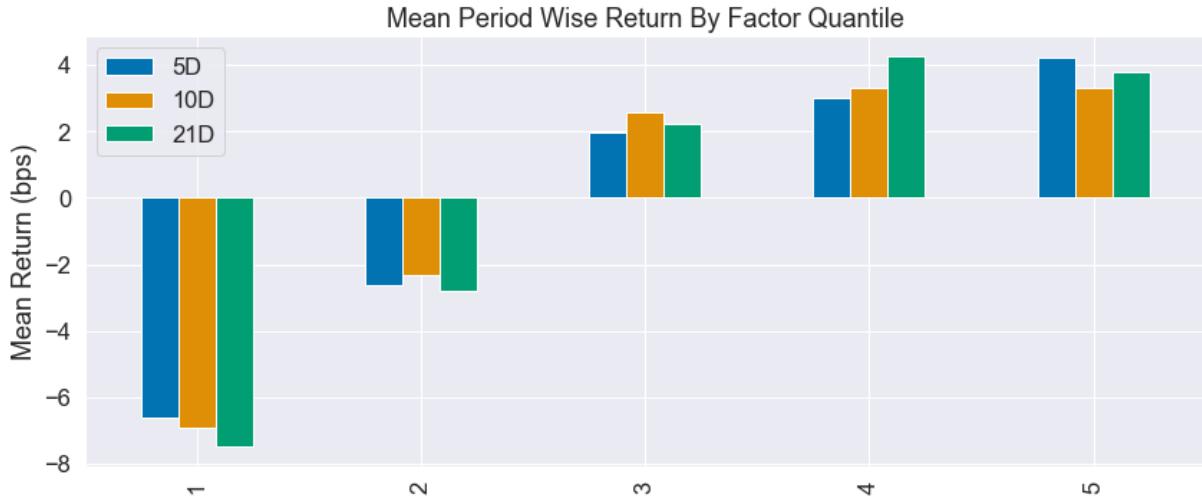


Figure 20.9: Mean period-wise spread by prediction quintile

To evaluate how the predictive performance might translate into returns over time, we lot the cumulative returns of similarly invested portfolios, as well as the cumulative return for a long-short portfolio invested in the top and bottom half, respectively:

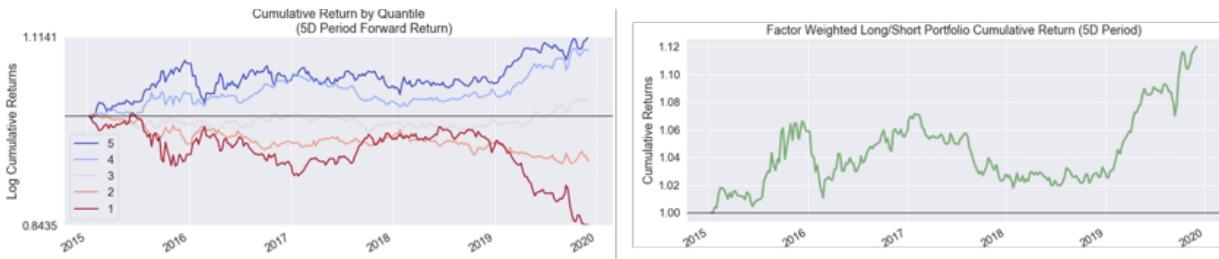


Figure 20.10: Cumulative returns of quintile-based and long-short portfolios

The results show significant spreads between quintile portfolios and positive cumulative returns for the broader-based long-short portfolio over time. This supports the hypothesis that the conditional autoencoder model could contribute to a profitable trading strategy.

Lessons learned and next steps

The conditional autoencoder combines a nonlinear version of the data-driven risk factors we explored using PCA in *Chapter 13, Data-Driven Risk Factors and Asset Allocation with Unsupervised Learning*, with the risk factor approach to modeling returns discussed in *Chapter 4* and *Chapter 7*. It illustrates how deep neural network architectures can be flexibly adapted to various tasks as well as the fluid boundary between autoencoders and feedforward neural networks.

The numerous simplifications from the data source to the architecture point to several avenues for improvements. Besides sourcing more data of better quality that also allows the computation of additional characteristics, the following modifications are a starting point—there are certainly many more:

- Experiment with **data frequencies** other than weekly and forecast horizons other than annual, where shorter periods will also increase the amount of training data
- Modify the **model architecture**, especially if using more data, which might reverse the finding that an even smaller hidden layer would estimate better factor loadings

Summary

In this chapter, we introduced how unsupervised learning leverages deep learning. Autoencoders learn sophisticated, nonlinear feature representations that are capable of significantly compressing complex data while losing little information. As a result, they are very useful to counter the curse of dimensionality associated with rich datasets that have many features, especially common datasets

with alternative data. We also saw how to implement various types of autoencoders using TensorFlow 2.

Most importantly, we implemented recent academic research that extracts data-driven risk factors from data to predict returns.

Different from our linear approach to this challenge in *Chapter 13, Data-Driven Risk Factors and Asset Allocation with Unsupervised Learning*, autoencoders capture nonlinear relationships. Moreover, the flexibility of deep learning allowed us to incorporate numerous key asset characteristics to model more sensitive factors that helped predict returns.

In the next chapter, we focus on generative adversarial networks, which have often been called one of the most exciting recent developments in artificial intelligence, and see how they are capable of creating synthetic training data.

Generative Adversarial Networks for Synthetic Time-Series Data

Following the coverage of autoencoders in the previous chapter, this chapter introduces a second unsupervised deep learning technique: **generative adversarial networks (GANs)**. As with autoencoders, GANs complement the methods for dimensionality reduction and clustering introduced in *Chapter 13, Data-Driven Risk Factors and Asset Allocation with Unsupervised Learning*.

GANs were invented by Goodfellow et al. in 2014. Yann LeCun has called GANs the "most exciting idea in AI in the last ten years." A **GAN** trains two neural networks, called the **generator** and **discriminator**, in a competitive setting. The generator aims to produce samples that the discriminator is unable to distinguish from a given class of training data. The result is a generative model capable of producing synthetic samples representative of a certain target distribution but artificially and, thus, inexpensively created.

GANs have produced an avalanche of research and successful applications in many domains. While originally applied to images, Esteban, Hyland, and Rätsch (2017) applied GANs to the medical domain to generate **synthetic time-series data**. Experiments with financial data ensued (Koshiyama, Firoozye, and Treleaven 2019;

Wiese et al. 2019; Zhou et al. 2018; Fu et al. 2019) to explore whether GANs can generate data that simulates alternative asset price trajectories to train supervised or reinforcement algorithms, or to backtest trading strategies. We will replicate the Time-Series GAN presented at the 2019 NeurIPS by Yoon, Jarrett, and van der Schaar (2019) to illustrate the approach and demonstrate the results.

More specifically, in this chapter you will learn about the following:

- How GANs work, why they are useful, and how they can be applied to trading
- Designing and training GANs using TensorFlow 2
- Generating synthetic financial data to expand the inputs available for training ML models and backtesting



You can find the code samples for this chapter and links to additional resources in the corresponding directory of the GitHub repository. The notebooks include color versions of the images.

Creating synthetic data with GANs

This book mostly focuses on supervised learning algorithms that receive input data and predict an outcome, which we can compare to the ground truth to evaluate their performance. Such algorithms are also called **discriminative models** because they learn to differentiate between different output values.

GANs are an instance of **generative models** like the variational autoencoder we encountered in the previous chapter. As described there, a generative model takes a training set with samples drawn from some distribution p_{data} and learns to represent an estimate p_{model} of that data-generating distribution.

As mentioned in the introduction, GANs are considered one of the most exciting recent machine learning innovations because they appear capable of generating high-quality samples that faithfully mimic a range of input data. This is very attractive given the absence or high cost of labeled data required for supervised learning.

GANs have triggered a wave of research that initially focused on the generation of surprisingly realistic images. More recently, GAN instances have emerged that produce synthetic time series with significant potential for trading since the limited availability of historical market data is a key driver of the risk of backtest overfitting.

In this section, we explain in more detail how generative models and adversarial training work and review various GAN architectures. In the next section, we will demonstrate how to design and train a GAN using TensorFlow 2. In the last section, we will describe how to adapt a GAN so that it creates synthetic time-series data.

Comparing generative and discriminative models

Discriminative models learn how to differentiate among outcomes y , given input data X . In other words, they learn the probability of the outcome given the data: $p(y \mid X)$. Generative models, on the other

hand, learn the joint distribution of inputs and outcome $p(y, X)$. While generative models can be used as discriminative models using Bayes' rule to compute which class is most likely (see *Chapter 10, Bayesian ML – Dynamic Sharpe Ratios and Pairs Trading*), it often seems preferable to solve the prediction problem directly rather than by solving the more general generative challenge first (Ng and Jordan 2002).

GANs have a generative objective: they produce complex outputs, such as realistic images, given simple inputs that can even be random numbers. They achieve this by modeling a probability distribution over the possible outputs. This probability distribution can have many dimensions, for example, one for each pixel in an image, each character or token in a document, or each value in a time series. As a result, the model can generate outputs that are very likely representative of the class of outputs.

Richard Feynman's quote "**What I cannot create, I do not understand**" emphasizes that modeling generative distributions is an important step towards more general AI and resembles human learning, which succeeds using much fewer samples.

Generative models have several **use cases** beyond their ability to generate additional samples from a given distribution. For example, they can be incorporated into model-based **reinforcement learning (RL)** algorithms (see the next chapter). Generative models can also be applied to time-series data to simulate alternative past or possible future trajectories that can be used for planning in RL or supervised learning more generally, including for the design of trading algorithms. Other use cases include semi-supervised learning where GANs facilitate feature matching to assign missing labels with much fewer training samples than current approaches.

Adversarial training – a zero-sum game of trickery

The key innovation of GANs is a new way of learning the data-generating probability distribution. The algorithm sets up a competitive, or adversarial game between two neural networks called the **generator** and the **discriminator**.

The generator's goal is to convert random noise input into fake instances of a specific class of objects, such as images of faces or stock price time series. The discriminator, in turn, aims to differentiate the generator's deceptive output from a set of training data containing true samples of the target objects. The overall GAN objective is for both networks to get better at their respective tasks so that the generator produces outputs that a machine can no longer distinguish from the originals (at which point we don't need the discriminator, which is no longer necessary, and can discard it).

Figure 21.1 illustrates adversarial training using a generic GAN architecture designed to generate images. We assume the generator uses a deep CNN architecture (such as the VGG16 example from *Chapter 18, CNNs for Financial Time Series and Satellite Images*) that is reversed just like the decoder part of the convolutional autoencoder we discussed in the previous chapter. The generator receives an input image with random pixel values and produces a *fake* output image that is passed on to the discriminator network, which uses a mirrored CNN architecture. The discriminator network also receives *real* samples that represent the target distribution and predicts the probability that the input is *real*, as opposed to *fake*. Learning takes place by backpropagating the gradients of the discriminator and generator losses to the respective network's parameters:

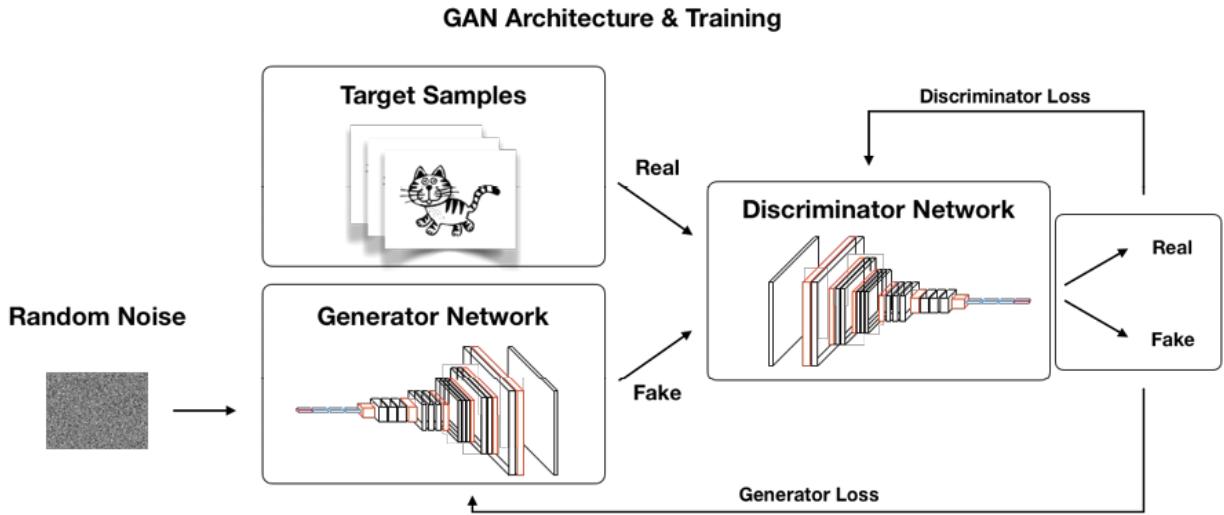


Figure 21.1: GAN architecture

The recent GAN Lab is a great interactive tool inspired by TensorFlow Playground, which allows the user to design GANs and visualize various aspects of the learning process and performance over time (see resource links on GitHub).

The rapid evolution of the GAN architecture zoo

Since the publication of the paper by Goodfellow et al. in 2014, GANs have attracted an enormous amount of interest and triggered a corresponding flurry of research.

The bulk of this work has refined the original architecture to adapt it to different domains and tasks, as well as expanding it to include additional information and create conditional GANs. Additional research has focused on improving methods for the challenging training process, which requires achieving a stable game-theoretic equilibrium between two networks, each of which can be tricky to train on its own.

The GAN landscape has become more diverse than we can cover here; see Creswell et al. (2018) and Pan et al. (2019) for recent surveys, and Odena (2019) for a list of open questions.

Deep convolutional GANs for representation learning

Deep convolutional GANs (DCGANs) were motivated by the successful application of CNNs to supervised learning for grid-like data (Radford, Metz, and Chintala 2016). The architecture pioneered the use of GANs for unsupervised learning by developing a feature extractor based on adversarial training. It is also easier to train and generates higher-quality images. It is now considered a baseline implementation, with numerous open source examples available (see references on GitHub).

A DCGAN network takes uniformly distributed random numbers as input and outputs a color image with a resolution of 64×64 pixels. As the input changes incrementally, so do the generated images. The network consists of standard CNN components, including deconvolutional layers that reverse convolutional layers as in the convolutional autoencoder example in the previous chapter, or fully connected layers.

The authors experimented exhaustively and made several recommendations, such as the use of batch normalization and ReLU activations in both networks. We will explore a TensorFlow implementation later in this chapter.

Conditional GANs for image-to-image translation

Conditional GANs (cGANs) introduce additional label information into the training process, resulting in better quality and some control

over the output.

cGANs alter the baseline architecture displayed previously in *Figure 21.1* by adding a third input to the discriminator that contains class labels. These labels, for example, could convey gender or hair color information when generating images.

Extensions include the **generative adversarial what-where network (GAWWN; Reed et al. 2016)**, which uses bounding box information not only to generate synthetic images but also to place objects at a given location.

GAN applications to images and time-series data

Alongside a large variety of extensions and modifications of the original architecture, numerous applications to images, as well as sequential data like speech and music, have emerged. Image applications are particularly diverse, ranging from image blending and super-resolution to video generation and human pose identification. Furthermore, GANs have been used to improve supervised learning performance.

We will look at a few salient examples and then take a closer look at applications to time-series data that may become particularly relevant to algorithmic trading and investment. See Alqahtani, Kavakli-Thorne, and Kumar (2019) for a recent survey and GitHub references for additional resources.

CycleGAN – unpaired image-to-image translation

Supervised image-to-image translation aims to learn a mapping between aligned input and output images. CycleGAN solves this task when paired images are not available and transforms images from one domain to match another.

Popular examples include the synthetic "painting" of horses as zebras and vice versa. It also includes the transfer of styles, by generating a realistic sample of an impressionistic print from an arbitrary landscape photo (Zhu et al. 2018).

StackGAN – text-to-photo image synthesis

One of the earlier applications of GANs to domain-transfer is the generation of images based on text. **Stacked GAN**, often shortened to **StackGAN**, uses a sentence as input and generates multiple images that match the description.

The architecture operates in two stages, where the first stage yields a low-resolution sketch of shape and colors, and the second stage enhances the result to a high-resolution image with photorealistic details (Zhang et al. 2017).

SRGAN – photorealistic single image super-resolution

Super-resolution aims at producing higher-resolution photorealistic images from low-resolution input. GANs applied to this task have deep CNN architectures that use batch normalization, ReLU, and skip connection as encountered in ResNet (see *Chapter 18, CNNs for Financial Time Series and Satellite Images*) to produce impressive results that are already finding commercial applications (Ledig et al. 2017).

Synthetic time series with recurrent conditional GANs

Recurrent GANs (RGANs) and recurrent conditional GANs (RCGANs) are two model architectures that aim to synthesize realistic real-valued multivariate time series (Esteban, Hyland, and Rätsch 2017). The authors target applications in the medical domain, but the approach could be highly valuable to overcome the limitations of historical market data.

RGANs rely on recurrent neural networks (RNNs) for the generator and the discriminator. RCGANs add auxiliary information in the spirit of cGANs (see the previous *Conditional GANs for image-to-image translation* section).

The authors succeed in generating visually and quantitatively compelling realistic samples. Furthermore, they evaluate the quality of the synthetic data, including synthetic labels, by using it to train a model with only minor degradation of the predictive performance on a real test set. The authors also demonstrate the successful application of RCGANs to an early warning system using a medical dataset of 17,000 patients from an intensive care unit. Hence, the authors illustrate that RCGANs are capable of generating time-series data useful for supervised training. We will apply this approach to financial market data this chapter in the *TimeGAN – adversarial training for synthetic financial data* section.

How to build a GAN using TensorFlow 2

To illustrate the implementation of a GAN using Python, we will use the DCGAN example discussed earlier in this section to synthesize images from the Fashion-MNIST dataset that we first encountered in *Chapter 13, Data-Driven Risk Factors and Asset Allocation with Unsupervised Learning*.

See the notebook `deep_convolutional_generative_adversarial_network` for implementation details and references.

Building the generator network

Both generator and discriminator use a deep CNN architecture along the lines illustrated in *Figure 20.1*, but with fewer layers. The generator uses a fully connected input layer, followed by three convolutional layers, as defined in the following `build_generator()` function, which returns a Keras model instance:

```
def build_generator():
    return Sequential([
        Dense(7 * 7 * 256,
              use_bias=False,
              input_shape=(100, ),
              name='IN'),
        BatchNormalization(name='BN1'),
        LeakyReLU(name='RELU1'),
        Reshape((7, 7, 256), name='SHAPE1'),
        Conv2DTranspose(128, (5, 5),
                       strides=(1, 1),
                       padding='same',
                       use_bias=False,
                       name='CONV1'),
        BatchNormalization(name='BN2'),
        LeakyReLU(name='RELU2'),
        Conv2DTranspose(64, (5, 5),
                       strides=(2, 2),
                       padding='same',
                       use_bias=False,
                       name='CONV2'),
        BatchNormalization(name='BN3'),
```

```
LeakyReLU(name='RELU3'),
Conv2DTranspose(1, (5, 5),
               strides=(2, 2),
               padding='same',
               use_bias=False,
               activation='tanh',
               name='CONV3')],
name='Generator')
```

The generator accepts 100 one-dimensional random values as input, and it produces images that are 28 pixels wide and high and, thus, contain 784 data points.

A call to the `.summary()` method of the model returned by this function shows that this network has over 2.3 million parameters (see the notebook for details, including a visualization of the generator output prior to training).

Creating the discriminator network

The discriminator network uses two convolutional layers that translate the input received from the generator into a single output value. The model has around 212,000 parameters:

```
def build_discriminator():
    return Sequential([Conv2D(64, (5, 5),
                           strides=(2, 2),
                           padding='same',
                           input_shape=[28, 28, 1],
                           name='CONV1'),
                      LeakyReLU(name='RELU1'),
                      Dropout(0.3, name='D01'),
                      Conv2D(128, (5, 5),
                             strides=(2, 2),
                             padding='same',
                             name='CONV2'),
                      LeakyReLU(name='RELU2'),
                      Dropout(0.3, name='D02'),
```

```
Flatten(name='FLAT'),
Dense(1, name='OUT')],  
name='Discriminator')
```

Figure 21.2 depicts how the random input flows from the generator to the discriminator, as well as the input and output shapes of the various network components:

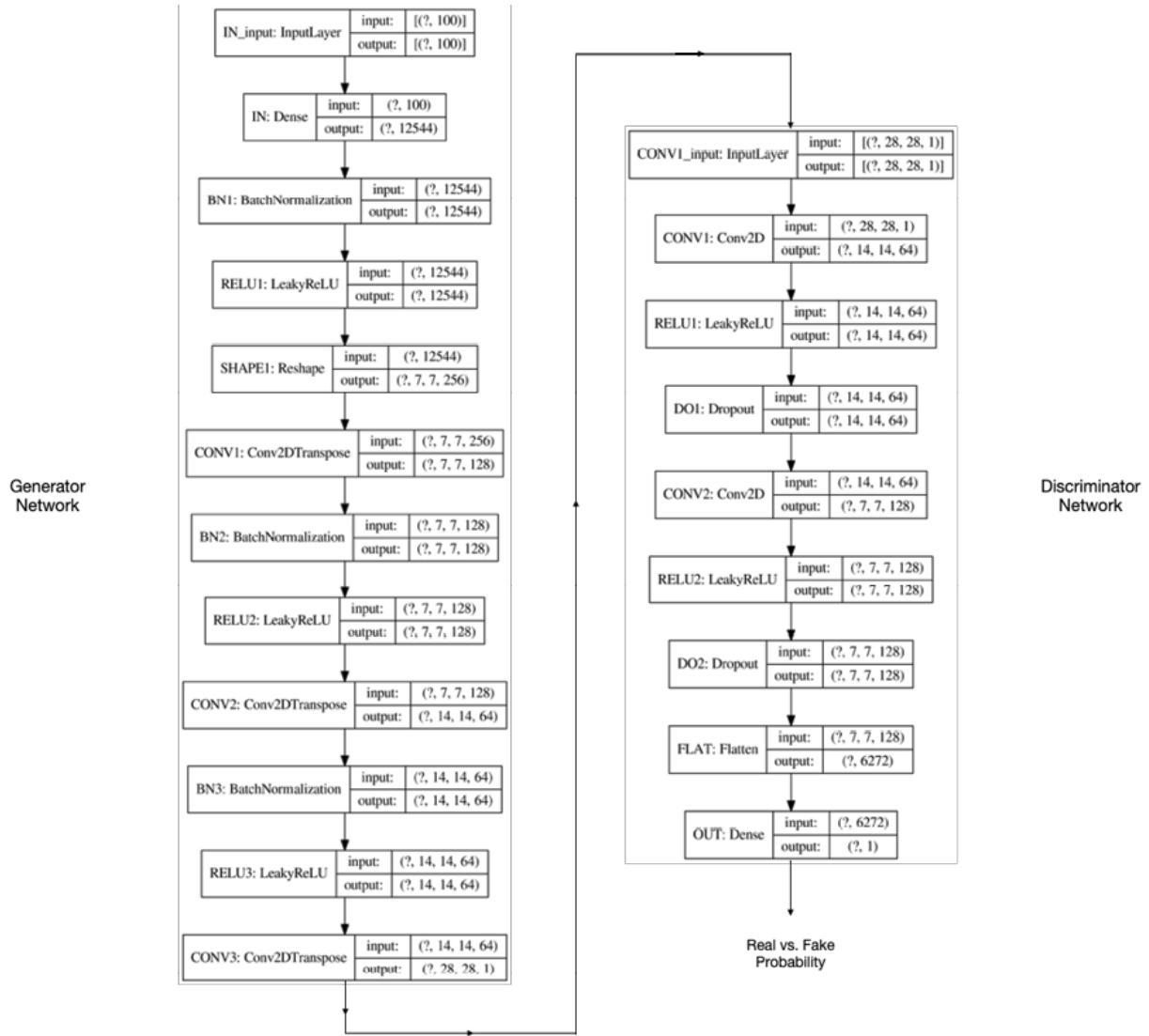


Figure 21.2: DCGAN TensorFlow 2 model architecture

Setting up the adversarial training process

Now that we have built the generator and the discriminator models, we will design and execute the adversarial training process. To this end, we will define the following:

- The loss functions for both models that reflect their competitive interaction
- A single training step that runs the backpropagation algorithm
- The training loop that repeats the training step until the model performance meets our expectations

Defining the generator and discriminator loss functions

The generator loss reflects the discriminator's decision regarding the fake input. It will be low if the discriminator mistakes an image produced by the generator for a real image, and high otherwise; we will define the interaction between both models when we create the training step.

The generator loss is measured by the binary cross-entropy loss function as follows:

```
cross_entropy = BinaryCrossentropy(from_logits=True)
def generator_loss(fake_output):
    return cross_entropy(tf.ones_like(fake_output), fake_output)
```

The discriminator receives both real and fake images as input. It computes a loss for each and attempts to minimize the sum with the goal of accurately recognizing both types of inputs:

```
def discriminator_loss(true_output, fake_output):
    true_loss = cross_entropy(tf.ones_like(true_output), true_output)
    fake_loss = cross_entropy(tf.zeros_like(fake_output), fake_output)
    return true_loss + fake_loss
```

To train both models, we assign each an Adam optimizer with a learning rate lower than the default:

```
gen_optimizer = Adam(1e-4)
dis_optimizer = Adam(1e-4)
```

The core – designing the training step

Each training step implements one round of stochastic gradient descent using the Adam optimizer. It consists of five steps:

1. Providing the minibatch inputs to each model
2. Getting the models' outputs for the current weights
3. Computing the loss given the models' objective and output
4. Obtaining the gradients for the loss with respect to each model's weights
5. Applying the gradients according to the optimizer's algorithm

The function `train_step()` carries out these five steps. We use the `@tf.function` decorator to speed up execution by compiling it to a TensorFlow operation rather than relying on eager execution (see the TensorFlow documentation for details):

```
@tf.function
def train_step(images):
    # generate the random input for the generator
    noise = tf.random.normal([BATCH_SIZE, noise_dim])
    with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
        # get the generator output
        generated_img = generator(noise, training=True)
        # collect discriminator decisions regarding real and fake input
        true_output = discriminator(images, training=True)
        fake_output = discriminator(generated_img, training=True)
        # compute the loss for each model
```

```

        gen_loss = generator_loss(fake_output)
        disc_loss = discriminator_loss(true_output, fake_output)
    # compute the gradients for each loss with respect to the model variables
    grad_generator = gen_tape.gradient(gen_loss,
                                        generator.trainable_variables)
    grad_discriminator = disc_tape.gradient(disc_loss,
                                              discriminator.trainable_variables)
    # apply the gradient to complete the backpropagation step
    gen_optimizer.apply_gradients(zip(grad_generator,
                                      generator.trainable_variables))
    dis_optimizer.apply_gradients(zip(grad_discriminator,
                                      discriminator.trainable_variables))

```

Putting it together – the training loop

The training loop is very straightforward to implement once we have the training step properly defined. It consists of a simple `for` loop, and during each iteration, we pass a new batch of real images to the training step. We also will sample some synthetic images and occasionally save the model weights.

Note that we track progress using the `tqdm` package, which shows the percentage complete during training:

```

def train(dataset, epochs, save_every=10):
    for epoch in tqdm(range(epochs)):
        for img_batch in dataset:
            train_step(img_batch)
        # produce images for the GIF as we go
        display.clear_output(wait=True)
        generate_and_save_images(generator, epoch + 1, seed)
        # Save the model every 10 EPOCHS
        if (epoch + 1) % save_every == 0:
            checkpoint.save(file_prefix=checkpoint_prefix)
        # Generator after final epoch
        display.clear_output(wait=True)
        generate_and_save_images(generator, epochs, seed)
    train(train_set, EPOCHS)

```

Evaluating the results

After 100 epochs that only take a few minutes, the synthetic images created from random noise clearly begin to resemble the originals, as you can see in *Figure 21.3* (see the notebook for the best visual quality):

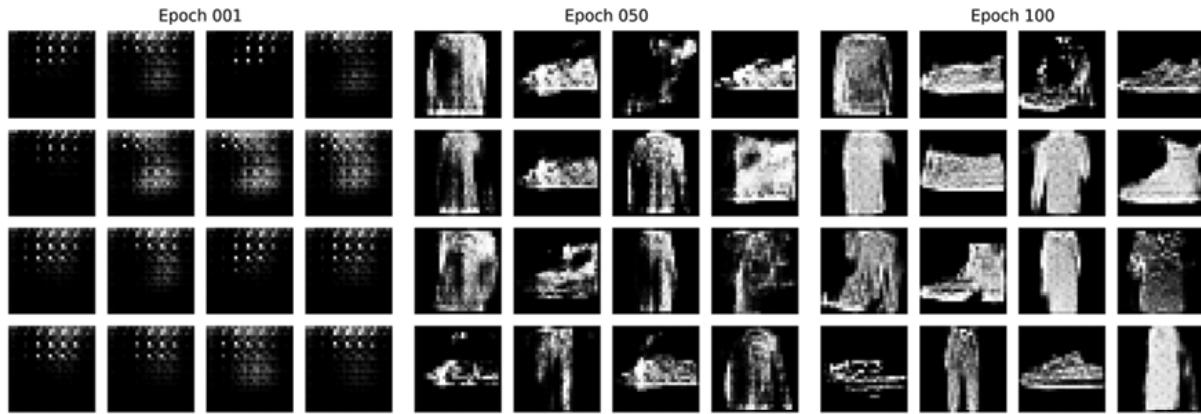


Figure 21.3: A sample of synthetic Fashion-MNIST images

The notebook also creates a dynamic GIF image that visualizes how the quality of the synthetic images improves during training.

Now that we understand how to build and train a GAN using TensorFlow 2, we will move on to a more complex example that produces synthetic time series from stock price data.

TimeGAN for synthetic financial data

Generating synthetic time-series data poses specific challenges above and beyond those encountered when designing GANs for images. In

addition to the distribution over variables at any given point, such as pixel values or the prices of numerous stocks, a generative model for time-series data should also learn the temporal dynamics that shape how one sequence of observations follows another. (Refer also to the discussion in *Chapter 9, Time-Series Models for Volatility Forecasts and Statistical Arbitrage*).

Very recent and promising research by Yoon, Jarrett, and van der Schaar, presented at NeurIPS in December 2019, introduces a novel **time-series generative adversarial network (TimeGAN)** framework that aims to account for temporal correlations by combining supervised and unsupervised training. The model learns a time-series embedding space while optimizing both supervised and adversarial objectives, which encourage it to adhere to the dynamics observed while sampling from historical data during training. The authors test the model on various time series, including historical stock prices, and find that the quality of the synthetic data significantly outperforms that of available alternatives.

In this section, we will outline how this sophisticated model works, highlight key implementation steps that build on the previous DCGAN example, and show how to evaluate the quality of the resulting time series. Please see the paper for additional information.

Learning to generate data across features and time

A successful generative model for time-series data needs to capture both the cross-sectional distribution of features at each point in time and the longitudinal relationships among these features over time. Expressed in the image context we just discussed, the model needs

to learn not only what a realistic image looks like, but also how one image evolves from the previous as in a video.

Combining adversarial and supervised training

As mentioned in the first section, prior attempts at generating time-series data, like RGANs and RCGANs, relied on RNNs (see *Chapter 19, RNNs for Multivariate Time Series and Sentiment Analysis*) in the roles of generator and discriminator. TimeGAN explicitly incorporates the autoregressive nature of time series by combining the **unsupervised adversarial loss** on both real and synthetic sequences familiar from the DCGAN example with a **stepwise supervised loss** with respect to the original data. The goal is to reward the model for learning the **distribution over transitions** from one point in time to the next that are present in the historical data.

Furthermore, TimeGAN includes an embedding network that maps the time-series features to a lower-dimensional latent space to reduce the complexity of the adversarial space. The motivation is to capture the drivers of temporal dynamics that often have lower dimensionality. (Refer also to the discussions of manifold learning in *Chapter 13, Data-Driven Risk Factors and Asset Allocation with Unsupervised Learning* and nonlinear dimensionality reduction in *Chapter 20, Autoencoders for Conditional Risk Factors and Asset Pricing*).

A key element of the TimeGAN architecture is that both the generator and the embedding (or autoencoder) network are responsible for minimizing the supervised loss that measures how well the model learns the dynamic relationship. As a result, the model learns a latent space conditioned on facilitating the generator's task to faithfully reproduce the temporal relationships observed in the historical data. In addition to time-series data, the

model can also process static data that does not change or changes less frequently over time.

The four components of the TimeGAN architecture

The TimeGAN architecture combines an adversarial network with an autoencoder and thus has four network components, as depicted in *Figure 21.4*:

1. **Autoencoder**: embedding and recovery networks
2. **Adversarial network**: sequence generator and sequence discriminator components

The authors emphasize the **joint training** of the autoencoder and the adversarial networks by means of **three different loss functions**. The **reconstruction loss** optimizes the autoencoder, the **unsupervised loss** trains the adversarial net, and the **supervised loss** enforces the temporal dynamics. As a result of this key insight, the TimeGAN simultaneously learns to encode features, generate representations, and iterate across time. More specifically, the embedding network creates the latent space, the adversarial network operates within this space, and supervised loss synchronizes the latent dynamics of both real and synthetic data.

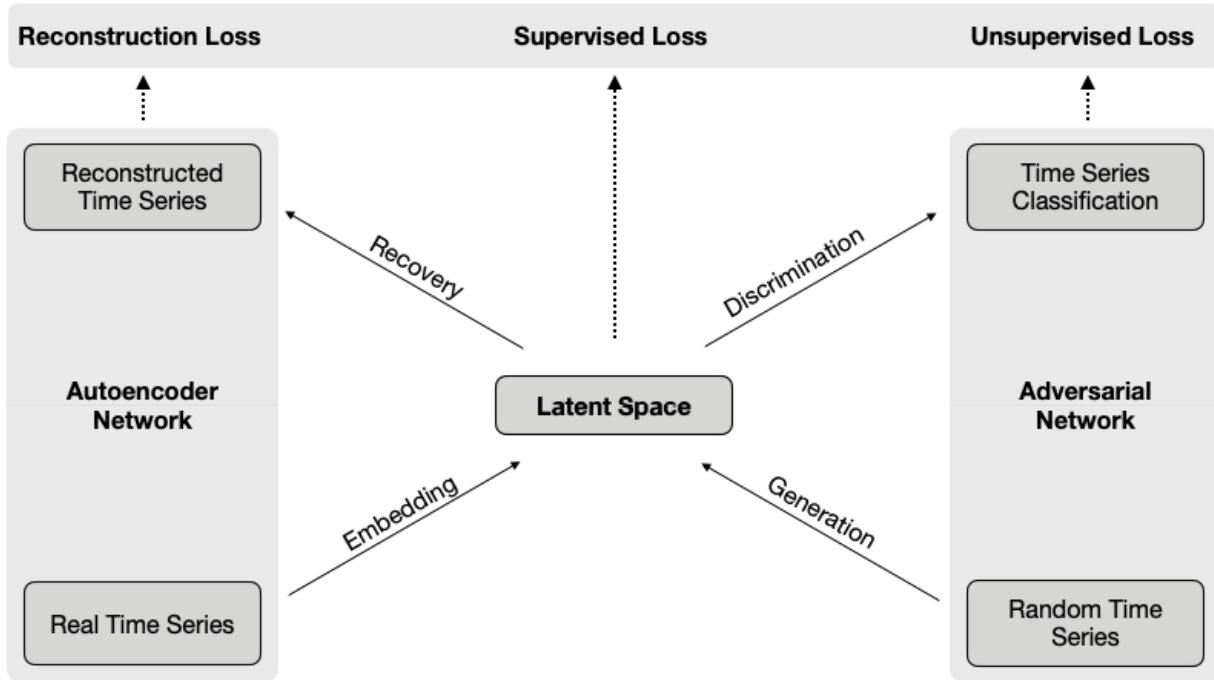


Figure 21.4: The components of the TimeGAN architecture

The **embedding and recovery** components of the autoencoder map the feature space into the latent space and vice versa. This facilitates the learning of the temporal dynamics by the adversarial network, which learns in a lower-dimensional space. The authors implement the embedding and recovery network using a stacked RNN and a feedforward network. However, these choices can be flexibly adapted to the task at hand as long as they are autoregressive and respect the temporal order of the data.

The **generator and the discriminator** elements of the adversarial network differ from the DCGAN not only because they operate on sequential data but also because the synthetic features are generated in the latent space that the model learns simultaneously. The authors chose an RNN as the generator and a bidirectional RNN with a feedforward output layer for the discriminator.

Joint training of an autoencoder and adversarial network

The three loss functions displayed in *Figure 21.4* drive the joint optimization of the network elements just described while training on real and randomly generated time series. In more detail, they aim to accomplish the following:

- The **reconstruction loss** is familiar from our discussion of autoencoders in *Chapter 20, Autoencoders for Conditional Risk Factors and Asset Pricing*; it compares how well the reconstruction of the encoded data resembles the original.
- The **unsupervised loss** reflects the competitive interaction between the generator and the discriminator described in the DCGAN example; while the generator aims to minimize the probability that the discriminator classifies its output as fake, the discriminator aims to optimize the correct classification of real and fake inputs.
- The **supervised loss** captures how well the generator approximates the actual next time step in latent space when receiving encoded real data for the prior sequence.

Training takes place in **three phases**:

1. Training the autoencoder on real time series to optimize reconstruction
2. Optimizing the supervised loss using real time series to capture the temporal dynamics of the historical data
3. Jointly training the four components while minimizing all three loss functions

TimeGAN includes several **hyperparameters** used to weigh the components of composite loss functions; however, the authors find the network to be less sensitive to these settings than one might expect given the notorious difficulties of GAN training. In fact, they

do not discover significant challenges during training and suggest that the embedding task serves to regularize adversarial learning because it reduces its dimensionality while the supervised loss constrains the stepwise dynamics of the generator.

We now turn to the TimeGAN implementation using TensorFlow 2; see the paper for an in-depth explanation of the math and methodology of the approach.

Implementing TimeGAN using TensorFlow 2

In this section, we will implement the TimeGAN architecture just described. The authors provide sample code using TensorFlow 1 that we will port to TensorFlow 2. Building and training TimeGAN requires several steps:

1. Selecting and preparing real and random time series inputs
2. Creating the key TimeGAN model components
3. Defining the various loss functions and training steps used during the three training phases
4. Running the training loops and logging the results
5. Generating synthetic time series and evaluating the results

We'll walk through the key items for each of these steps; please refer to the notebook `TimeGAN_TF2` for the code examples in this section (unless otherwise noted), as well as additional implementation details.

Preparing the real and random input series

The authors demonstrate the applicability of TimeGAN to financial data using 15 years of daily Google stock prices downloaded from Yahoo Finance with six features, namely open, high, low, close and adjusted close price, and volume. We'll instead use close to 20 years of adjusted close prices for six different tickers because it introduces somewhat higher variability. We will follow the original paper in targeting synthetic series with 24 time steps.

Among the stocks with the longest history in the Quandl Wiki dataset are those displayed in normalized format, that is, starting at 1.0, in *Figure 21.5*. We retrieve the adjusted close from 2000-2017 and obtain over 4,000 observations. The correlation coefficient among the series ranges from 0.01 for GE and CAT to 0.94 for DIS and KO.



Figure 21.5: The TimeGAN input—six real stock prices series

We scale each series to the range $[0, 1]$ using scikit-learn's `MinMaxScaler` class, which we will later use to rescale the synthetic data:

```
df = pd.read_hdf(hdf_store, 'data/real')
scaler = MinMaxScaler()
scaled_data = scaler.fit_transform(df).astype(np.float32)
```

In the next step, we create rolling windows containing overlapping sequences of 24 consecutive data points for the six series:

```
data = []
for i in range(len(df) - seq_len):
    data.append(scaled_data[i:i + seq_len])
n_series = len(data)
```

We then create a `tf.data.Dataset` instance from the list of `NumPy` arrays, ensure the data gets shuffled while training, and set a batch size of 128:

```
real_series = (tf.data.Dataset
               .from_tensor_slices(data)
               .shuffle(buffer_size=n_windows)
               .batch(batch_size))
real_series_iter = iter(real_series.repeat())
```

We also need a random time-series generator that produces simulated data with 24 observations on the six series for as long as the training continues.

To this end, we will create a generator that draws the requisite data uniform at random and feeds the result into a second `tf.data.Dataset` instance. We set this dataset to produce batches of the desired size and to repeat the process for as long as necessary:

```
def make_random_data():
    while True:
        yield np.random.uniform(low=0, high=1, size=(seq_len, n_seq))
random_series = iter(tf.data.Dataset
                     .from_generator(make_random_data,
                                    output_types=tf.float32)
                     .batch(batch_size)
                     .repeat())
```

We'll now proceed to define and instantiate the TimeGAN model components.

Creating the TimeGAN model components

We'll now create the two autoencoder components and the two adversarial network elements, as well as the supervisor that encourages the generator to learn the temporal dynamic of the historical price series.

We will follow the authors' sample code in creating RNNs with three hidden layers, each with 24 GRU units, except for the supervisor, which uses only two hidden layers. The following `make_rnn` function automates the network creation:

```
def make_rnn(n_layers, hidden_units, output_units, name):
    return Sequential([GRU(units=hidden_units,
                           return_sequences=True,
                           name=f'GRU_{i + 1}') for i in range(n_layers - 1),
                       Dense(units=output_units,
                             activation='sigmoid',
                             name='OUT')], name=name)
```

The `autoencoder` consists of the `embedder` and the recovery networks that we instantiate here:

```
embedder = make_rnn(n_layers=3,
                     hidden_units=hidden_dim,
                     output_units=hidden_dim,
                     name='Embedder')
recovery = make_rnn(n_layers=3,
                     hidden_units=hidden_dim,
                     output_units=n_seq,
                     name='Recovery')
```

We then create the generator, the discriminator, and the supervisor like so:

```
generator = make_rnn(n_layers=3,
                      hidden_units=hidden_dim,
                      output_units=hidden_dim,
                      name='Generator')
discriminator = make_rnn(n_layers=3,
                         hidden_units=hidden_dim,
                         output_units=1,
                         name='Discriminator')
supervisor = make_rnn(n_layers=2,
                      hidden_units=hidden_dim,
                      output_units=hidden_dim,
                      name='Supervisor')
```

We also define two generic loss functions, namely `MeanSquaredError` and `BinaryCrossEntropy`, which we will use later to create the various specific loss functions during the three phases:

```
mse = MeanSquaredError()
bce = BinaryCrossentropy()
```

Now it's time to start the training process.

Training phase 1 – autoencoder with real data

The autoencoder integrates the embedder and the recovery functions, as we saw in the previous chapter:

```
H = embedder(X)
X_tilde = recovery(H)
autoencoder = Model(inputs=X,
                     outputs=X_tilde,
                     name='Autoencoder')
autoencoder.summary()
Model: "Autoencoder"
```

Layer (type)	Output Shape	Param #
<hr/>		
RealData (InputLayer)	[<code>(None, 24, 6)</code>]	0
Embedder (Sequential)	(<code>None, 24, 24</code>)	10104
Recovery (Sequential)	(<code>None, 24, 6</code>)	10950
<hr/>		
Trainable params:	<code>21,054</code>	

It has 21,054 parameters. We will now instantiate the optimizer for this training phase and define the training step. It follows the pattern introduced with the DCGAN example, using `tf.GradientTape` to record the operations that generate the reconstruction loss. This allows us to rely on the automatic differentiation engine to obtain the gradients with respect to the trainable embedder and recovery network weights that drive `backpropagation`:

```
autoencoder_optimizer = Adam()
@tf.function
def train_autoencoder_init(x):
    with tf.GradientTape() as tape:
        x_tilde = autoencoder(x)
        embedding_loss_t0 = mse(x, x_tilde)
        e_loss_0 = 10 * tf.sqrt(embedding_loss_t0)
    var_list = embedder.trainable_variables + recovery.trainable_variables
    gradients = tape.gradient(e_loss_0, var_list)
    autoencoder_optimizer.apply_gradients(zip(gradients, var_list))
    return tf.sqrt(embedding_loss_t0)
```

The reconstruction loss simply compares the autoencoder outputs with its inputs. We train for 10,000 steps in a little over one minute using this training loop that records the step loss for monitoring with TensorBoard:

```
for step in tqdm(range(train_steps)):
    X_ = next(real_series_iter)
    step_e_loss_t0 = train_autoencoder_init(X_)
```

```
    with writer.as_default():
        tf.summary.scalar('Loss Autoencoder Init', step_e_loss_t0, step_e_loss_t1)
```

Training phase 2 – supervised learning with real data

We already created the supervisor model so we just need to instantiate the optimizer and define the train step as follows:

```
supervisor_optimizer = Adam()
@tf.function
def train_supervisor(x):
    with tf.GradientTape() as tape:
        h = embedder(x)
        h_hat_supervised = supervisor(h)
        g_loss_s = mse(h[:, 1:, :], h_hat_supervised[:, 1:, :])
    var_list = supervisor.trainable_variables
    gradients = tape.gradient(g_loss_s, var_list)
    supervisor_optimizer.apply_gradients(zip(gradients, var_list))
    return g_loss_s
```

In this case, the loss compares the output of the supervisor with the next timestep for the embedded sequence so that it learns the temporal dynamics of the historical price sequences; the training loop works similarly to the autoencoder example in the previous chapter.

Training phase 3 – joint training with real and random data

The joint training involves all four network components, as well as the supervisor. It uses multiple loss functions and combinations of the base components to achieve the simultaneous learning of latent space embeddings, transition dynamics, and synthetic data generation.

We will highlight a few salient examples; please see the notebook for the full implementation that includes some repetitive steps that we will omit here.

To ensure that the generator faithfully reproduces the time series, TimeGAN includes a moment loss that penalizes when the mean and variance of the synthetic data deviate from the real version:

```
def get_generator_moment_loss(y_true, y_pred):
    y_true_mean, y_true_var = tf.nn.moments(x=y_true, axes=[0])
    y_pred_mean, y_pred_var = tf.nn.moments(x=y_pred, axes=[0])
    g_loss_mean = tf.reduce_mean(tf.abs(y_true_mean - y_pred_mean))
    g_loss_var = tf.reduce_mean(tf.abs(tf.sqrt(y_true_var + 1e-6) -
                                      tf.sqrt(y_pred_var + 1e-6)))
    return g_loss_mean + g_loss_var
```

The end-to-end model that produces synthetic data involves the generator, supervisor, and recovery components. It is defined as follows and has close to 30,000 trainable parameters:

```
E_hat = generator(Z)
H_hat = supervisor(E_hat)
X_hat = recovery(H_hat)
synthetic_data = Model(inputs=Z,
                       outputs=X_hat,
                       name='SyntheticData')
Model: "SyntheticData"
```

Layer (type)	Output Shape	Param #
<hr/>		
RandomData (InputLayer)	[(None, 24, 6)]	0
Generator (Sequential)	(None, 24, 24)	10104
Supervisor (Sequential)	(None, 24, 24)	7800
Recovery (Sequential)	(None, 24, 6)	10950
<hr/>		
Trainable params: 28,854		

The joint training involves three optimizers for the autoencoder, the generator, and the discriminator:

```
generator_optimizer = Adam()
discriminator_optimizer = Adam()
embedding_optimizer = Adam()
```

The train step for the generator illustrates the use of four loss functions and corresponding combinations of network components to achieve the desired learning outlined at the beginning of this section:

```
@tf.function
def train_generator(x, z):
    with tf.GradientTape() as tape:
        y_fake = adversarial_supervised(z)
        generator_loss_unsupervised = bce(y_true=tf.ones_like(y_fake),
                                           y_pred=y_fake)
        y_fake_e = adversarial_emb(z)
        generator_loss_unsupervised_e = bce(y_true=tf.ones_like(y_fake_e),
                                            y_pred=y_fake_e)
        h = embedder(x)
        h_hat_supervised = supervisor(h)
        generator_loss_supervised = mse(h[:, 1:, :],
                                         h_hat_supervised[:, 1:, :])
        x_hat = synthetic_data(z)
        generator_moment_loss = get_generator_moment_loss(x, x_hat)
        generator_loss = (generator_loss_unsupervised +
                          generator_loss_unsupervised_e +
                          100 * tf.sqrt(generator_loss_supervised) +
                          100 * generator_moment_loss)
    var_list = generator.trainable_variables + supervisor.trainable_variables
    gradients = tape.gradient(generator_loss, var_list)
    generator_optimizer.apply_gradients(zip(gradients, var_list))
    return (generator_loss_unsupervised, generator_loss_supervised,
            generator_moment_loss)
```

Finally, the joint training loop pulls the various training steps together and builds on the learning from phase 1 and 2 to train the

TimeGAN components on both real and random data. We run the loop for 10,000 iterations in under 40 minutes:

```
for step in range(train_steps):
    # Train generator (twice as often as discriminator)
    for kk in range(2):
        X_ = next(real_series_iter)
        Z_ = next(random_series)
        # Train generator
        step_g_loss_u, step_g_loss_s, step_g_loss_v = train_generator(
            # Train embedder
            step_e_loss_t0 = train_embedder(X_)
        X_ = next(real_series_iter)
        Z_ = next(random_series)
        step_d_loss = get_discriminator_loss(X_, Z_)
        if step_d_loss > 0.15:
            step_d_loss = train_discriminator(X_, Z_)
        if step % 1000 == 0:
            print(f'{step:6,.0f} | d_loss: {step_d_loss:6.4f} | '
                  f'g_loss_u: {step_g_loss_u:6.4f} | '
                  f'g_loss_s: {step_g_loss_s:6.4f} | '
                  f'g_loss_v: {step_g_loss_v:6.4f} | '
                  f'e_loss_t0: {step_e_loss_t0:6.4f}')
        with writer.as_default():
            tf.summary.scalar('G Loss S', step_g_loss_s, step=step)
            tf.summary.scalar('G Loss U', step_g_loss_u, step=step)
            tf.summary.scalar('G Loss V', step_g_loss_v, step=step)
            tf.summary.scalar('E Loss T0', step_e_loss_t0, step=step)
            tf.summary.scalar('D Loss', step_d_loss, step=step)
```

Now we can finally generate synthetic time series!

Generating synthetic time series

To evaluate the `TimeGAN` results, we will generate synthetic time series by drawing random inputs and feeding them to the `synthetic_data` network just described in the preceding section. More specifically, we'll create roughly as many artificial series with 24

observations on the six tickers as there are overlapping windows in the real dataset:

```
generated_data = []
for i in range(int(n_windows / batch_size)):
    Z_ = next(random_series)
    d = synthetic_data(Z_)
    generated_data.append(d)
len(generated_data)
35
```

The result is 35 batches containing 128 samples, each with the dimensions 24×6 , that we stack like so:

```
generated_data = np.array(np.vstack(generated_data))
generated_data.shape
(4480, 24, 6)
```

We can use the trained `MinMaxScaler` to revert the synthetic output to the scale of the input series:

```
generated_data = (scaler.inverse_transform(generated_data
                                             .reshape(-1, n_seq))
                                             .reshape(-1, seq_len, n_seq))
```

Figure 21.6 displays samples of the six synthetic series and the corresponding real series. The synthetic data generally reflects a variation of behavior not unlike its real counterparts and, after rescaling, roughly (due to the random input) matches its range:

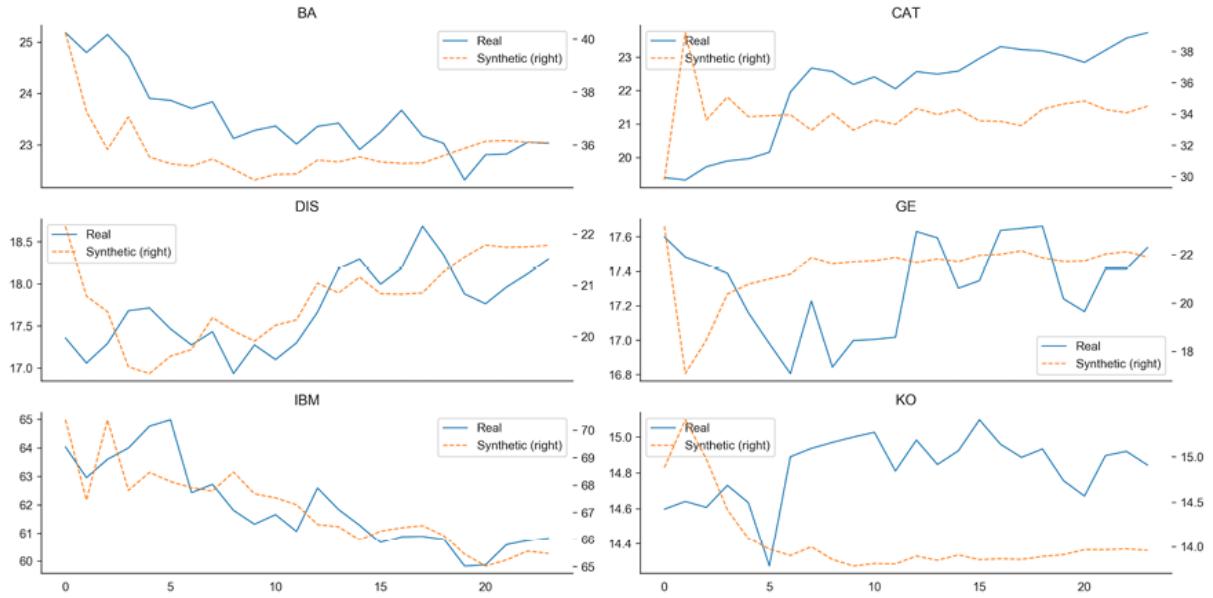


Figure 21.6: TimeGAN output—six synthetic prices series and their real counterparts

Now it's time to take a closer look at how to more thoroughly evaluate the quality of the synthetic data.

Evaluating the quality of synthetic time-series data

The TimeGAN authors assess the quality of the generated data with respect to three practical criteria:

- **Diversity:** The distribution of the synthetic samples should roughly match that of the real data.
- **Fidelity:** The sample series should be indistinguishable from the real data.
- **Usefulness:** The synthetic data should be as useful as its real counterparts for solving a predictive task.

They apply three methods to evaluate whether the synthetic data actually exhibits these characteristics:

- **Visualization:** For a qualitative diversity assessment of diversity, we use dimensionality reduction—**principal component analysis (PCA)** and **t-SNE** (see *Chapter 13, Data-Driven Risk Factors and Asset Allocation with Unsupervised Learning*)—to visually inspect how closely the distribution of the synthetic samples resembles that of the original data.
- **Discriminative score:** For a quantitative assessment of fidelity, the test error of a time-series classifier, such as a two-layer LSTM (see *Chapter 18, CNNs for Financial Time Series and Satellite Images*), lets us evaluate whether real and synthetic time series can be differentiated or are, in fact, indistinguishable.
- **Predictive score:** For a quantitative measure of usefulness, we can compare the test errors of a sequence prediction model trained on, alternatively, real or synthetic data to predict the next time step for the real data.

We'll apply and discuss the results of each method in the following sections. See the notebook `evaluating_synthetic_data` for the code samples and additional details.

Assessing diversity – visualization using PCA and t-SNE

To visualize the real and synthetic series with 24 time steps and six features, we will reduce their dimensionality so that we can plot them in two dimensions. To this end, we will sample 250 normalized sequences with six features each and reshape them to obtain data with the dimensionality $1,500 \times 24$ (showing only the steps for real data; see the notebook for the synthetic data):

```

# same steps to create real sequences for training
real_data = get_real_data()
# reload synthetic data
synthetic_data = np.load('generated_data.npy')
synthetic_data.shape
(4480, 24, 6)
# ensure same number of sequences
real_data = real_data[:synthetic_data.shape[0]]
sample_size = 250
idx = np.random.permutation(len(real_data))[:sample_size]
real_sample = np.asarray(real_data)[idx]
real_sample_2d = real_sample.reshape(-1, seq_len)
real_sample_2d.shape
(1500, 24)

```

PCA is a linear method that identifies a new basis with mutually orthogonal vectors that, successively, capture the directions of maximum variance in the data. We will compute the first two components using the real data and then project both real and synthetic samples onto the new coordinate system:

```

pca = PCA(n_components=2)
pca.fit(real_sample_2d)
pca_real = (pd.DataFrame(pca.transform(real_sample_2d))
            .assign(Data='Real'))
pca_synthetic = (pd.DataFrame(pca.transform(synthetic_sample_2d))
                 .assign(Data='Synthetic'))

```

t-SNE is a nonlinear manifold learning method for the visualization of high-dimensional data. It converts similarities between data points to joint probabilities and aims to minimize the Kullback-Leibler divergence between the joint probabilities of the low-dimensional embedding and the high-dimensional data (see *Chapter 13, Data-Driven Risk Factors and Asset Allocation with Unsupervised Learning*). We compute t-SNE for the combined real and synthetic data as follows:

```

tsne_data = np.concatenate((real_sample_2d,
                           synthetic_sample_2d), axis=0)
tsne = TSNE(n_components=2, perplexity=40)
tsne_result = tsne.fit_transform(tsne_data)

```

Figure 21.7 displays the PCA and t-SNE results for a qualitative assessment of the similarity of the real and synthetic data distributions. Both methods reveal strikingly similar patterns and significant overlap, suggesting that the synthetic data captures important aspects of the real data characteristics.

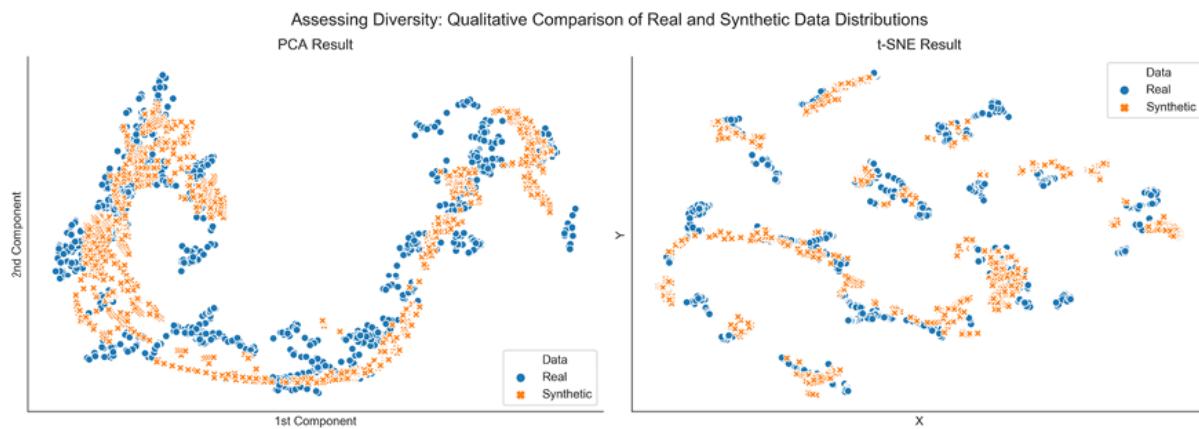


Figure 21.7: 250 samples of real and synthetic data in two dimensions

Assessing fidelity – time-series classification performance

The visualization only provides a qualitative impression. For a quantitative assessment of the fidelity of the synthetic data, we will train a time-series classifier to distinguish between real and fake data and evaluate its performance on a held-out test set.

More specifically, we will select the first 80 percent of the rolling sequences for training and the last 20 percent as a test set, as follows:

```

synthetic_data.shape
(4480, 24, 6)
n_series = synthetic_data.shape[0]
idx = np.arange(n_series)
n_train = int(.8*n_series)
train_idx, test_idx = idx[:n_train], idx[n_train:]
train_data = np.vstack((real_data[train_idx],
                        synthetic_data[train_idx]))
test_data = np.vstack((real_data[test_idx],
                        synthetic_data[test_idx]))
n_train, n_test = len(train_idx), len(test_idx)
train_labels = np.concatenate((np.ones(n_train),
                             np.zeros(n_train)))
test_labels = np.concatenate((np.ones(n_test),
                             np.zeros(n_test)))

```

Then we will create a simple RNN with six units that receives mini batches of real and synthetic series with the shape 24×6 and uses a sigmoid activation. We will optimize it using binary cross-entropy loss and the Adam optimizer, while tracking the AUC and accuracy metrics:

```

ts_classifier = Sequential([GRU(6, input_shape=(24, 6), name='GRU'),
                            Dense(1, activation='sigmoid', name='OUT')]
                           ts_classifier.compile(loss='binary_crossentropy',
                                                 optimizer='adam',
                                                 metrics=[AUC(name='AUC'), 'accuracy'])
Model: "Time Series Classifier"

```

Layer (type)	Output Shape	Param #
<hr/>		
GRU (GRU)	(None, 6)	252
OUT (Dense)	(None, 1)	7
<hr/>		
Total params: 259		
Trainable params: 259		

The model has 259 trainable parameters. We will train it for 250 epochs on batches of 128 randomly selected samples and track the

validation performance:

```
result = ts_classifier.fit(x=train_data,  
                           y=train_labels,  
                           validation_data=(test_data, test_labels),  
                           epochs=250, batch_size=128)
```

Once the training completes, evaluation of the test set yields a classification error of almost 56 percent on the balanced test set and a very low AUC of 0.15:

```
ts_classifier.evaluate(x=test_data, y=test_labels)  
56/56 [=====] - 0s 2ms/step - loss: 3.7510 -
```

Figure 21.8 plots the accuracy and AUC performance metrics for both train and test data over the 250 training epochs:

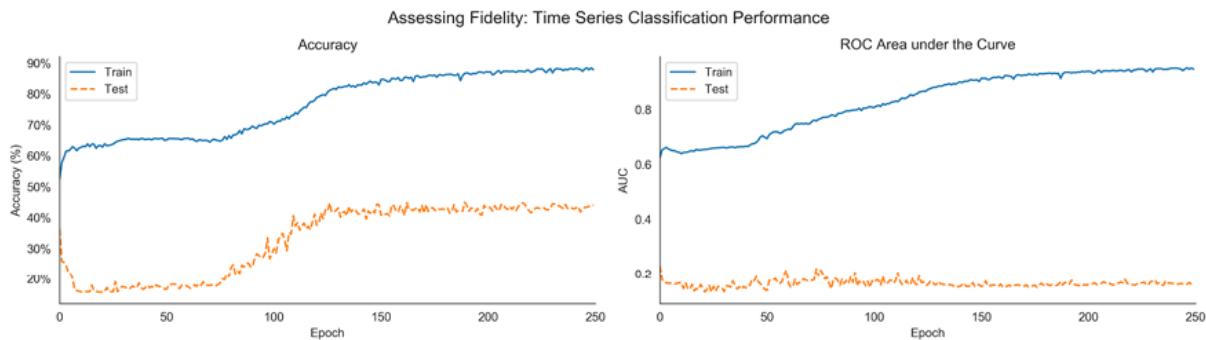


Figure 21.8: Train and test performance of the time-series classifier over 250 epochs

The plot shows that that model is not able to learn the difference between the real and synthetic data in a way that generalizes to the test set. This result suggests that the quality of the synthetic data meets the fidelity standard.

Assessing usefulness – train on synthetic, test on real

Finally, we want to know how useful synthetic data is when it comes to solving a prediction problem. To this end, we will train a time-series prediction model alternatively on the synthetic and the real data to predict the next time step and compare the performance on a test set created from the real data.

More specifically, we will select the first 23 time steps of each sequence as input, and the final time step as output. At the same time, we will split the real data into train and test sets using the same temporal split as in the previous classification example:

```
real_data.shape, synthetic_data.shape
((4480, 24, 6), (4480, 24, 6))
real_train_data = real_data[train_idx, :23, :]
real_train_label = real_data[train_idx, -1, :]
real_test_data = real_data[test_idx, :23, :]
real_test_label = real_data[test_idx, -1, :]
real_train_data.shape, real_train_label.shape
((3584, 23, 6), (3584, 6))
```

We will select the complete synthetic data for training since abundance is one of the reasons we generated it in the first place:

```
synthetic_train = synthetic_data[:, :23, :]
synthetic_label = synthetic_data[:, -1, :]
synthetic_train.shape, synthetic_label.shape
((4480, 23, 6), (4480, 6))
```

We will create a one-layer RNN with 12 GRU units that predicts the last time steps for the six stock price series and, thus, has six linear output units. The model uses the Adam optimizer to minimize the mean absolute error (MAE):

```
def get_model():
    model = Sequential([GRU(12, input_shape=(seq_len-1, n_seq)),
                        Dense(6)])
```

```

model.compile(optimizer=Adam(),
              loss=MeanAbsoluteError(name='MAE'))
return model

```

We will train the model twice using the synthetic and real data for training, respectively, and the real test set to evaluate the out-of-sample performance. Training on synthetic data works as follows; training on real data works analogously (see the notebook):

```

ts_regression = get_model()
synthetic_result = ts_regression.fit(x=synthetic_train,
                                      y=synthetic_label,
                                      validation_data=(real_test_data,
                                                       real_test_label),
                                      epochs=100,
                                      batch_size=128)

```

Figure 21.9 plots the MAE on the train and test sets (on a log scale so we can spot the differences) for both models. It turns out that the MAE is slightly lower after training on the synthetic dataset:

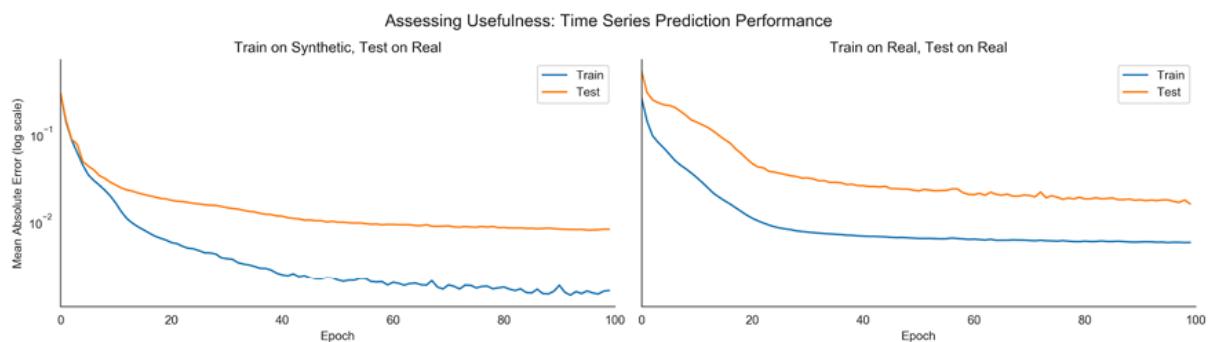


Figure 21.9: Train and test performance of the time-series prediction model over 100 epochs

The result shows that synthetic training data may indeed be useful. On the specific predictive task of predicting the next daily stock price for six tickers, a simple model trained on synthetic TimeGAN data delivers equal or better performance than training on real data.

Lessons learned and next steps

The perennial problem of overfitting that we encountered throughout this book implies that the ability to generate useful synthetic data would be quite valuable. The TimeGAN example justifies cautious optimism in this regard. At the same time, there are some **caveats**: we generated price data for a small number of assets at a daily frequency. In reality, we are probably interested in returns for a much larger number of assets, possibly at a higher frequency. The **cross-sectional and temporal dynamics** will certainly become more complex and may require adjustments to the TimeGAN architecture and training process.

These limitations of the experiment, however promising, imply natural next steps: we need to expand the scope to higher-dimensional time series containing information other than prices and also need to test their usefulness in the context of more complex models, including for feature engineering. These are very early days for synthetic training data, but this example should equip you to pursue your own research agenda towards more realistic solutions.

Summary

In this chapter, we introduced GANs that learn a probability distribution over the input data and are thus capable of generating synthetic samples that are representative of the target data.

While there are many practical applications for this very recent innovation, they could be particularly valuable for algorithmic trading if the success in generating time-series training data in the

medical domain can be transferred to financial market data. We learned how to set up adversarial training using TensorFlow. We also explored TimeGAN, a recent example of such a model, tailored to generating synthetic time-series data.

In the next chapter, we focus on reinforcement learning where we will build agents that interactively learn from their (market) environment.

Deep Reinforcement Learning – Building a Trading Agent

In this chapter, we'll introduce **reinforcement learning (RL)**, which takes a different approach to **machine learning (ML)** than the supervised and unsupervised algorithms we have covered so far. RL has attracted enormous attention as it has been the main driver behind some of the most exciting AI breakthroughs, like AlphaGo. David Silver, AlphaGo's creator and the lead RL researcher at Google-owned DeepMind, recently won the prestigious 2019 ACM Prize in Computing "for breakthrough advances in computer game-playing." We will see that the interactive and online nature of RL makes it particularly well-suited to the trading and investment domain.

RL models **goal-directed learning by an agent** that interacts with a typically stochastic environment that the agent has incomplete information about. RL aims to automate how the agent makes decisions to achieve a long-term objective by learning the value of states and actions from a reward signal. The ultimate goal is to derive a policy that encodes behavioral rules and maps states to actions.

RL is considered **most similar to human learning** that results from taking actions in the real world and observing the consequences. It differs from supervised learning because it optimizes the agent's

behavior one trial-and-error experience at a time based on a scalar reward signal, rather than by generalizing from correctly labeled, representative samples of the target concept. Moreover, RL does not stop at making predictions. Instead, it takes an end-to-end perspective on goal-oriented decision-making by including actions and their consequences.

In this chapter, you will learn how to formulate an RL problem and apply various solution methods. We will cover model-based and model-free methods, introduce the OpenAI Gym environment, and combine deep learning with RL to train an agent that navigates a complex environment. Finally, we'll show you how to adapt RL to algorithmic trading by modeling an agent that interacts with the financial market to optimize its profit objective.

More specifically, after reading this chapter, you will be able to:

- Define a **Markov decision problem (MDP)**
- Use value and policy iteration to solve an MDP
- Apply Q-learning in an environment with discrete states and actions
- Build and train a deep Q-learning agent in a continuous environment
- Use OpenAI Gym to train an RL trading agent



You can find the code samples for this chapter and links to additional resources in the corresponding directory of the GitHub repository. The notebooks include color versions of the images.

Elements of a reinforcement learning system

RL problems feature several elements that set them apart from the ML settings we have covered so far. The following two sections outline the key features required for defining and solving an RL problem by learning a policy that automates decisions. We'll use the notation and generally follow *Reinforcement Learning: An Introduction* (Sutton and Barto 2018) and David Silver's UCL Courses on RL (<https://www.davidsilver.uk/teaching/>), which are recommended for further study beyond the brief summary that the scope of this chapter permits.

RL problems aim to solve for actions that **optimize the agent's objective, given some observations about the environment**. The environment presents information about its state to the agent, assigns rewards for actions, and transitions the agent to new states, subject to probability distributions the agent may or may not know. It may be fully or partially observable, and it may also contain other agents. The structure of the environment has a strong impact on the agent's ability to learn a given task, and typically requires significant up-front design effort to facilitate the training process.

RL problems differ based on the complexity of the environment's state and agent's action spaces, which can be either discrete or continuous. Continuous actions and states, unless discretized, require machine learning to approximate a functional relationship between states, actions, and their values. They also require generalization because the agent almost certainly experiences only a subset of the potentially infinite number of states and actions during training.

Solving complex decision problems usually requires a simplified model that isolates the key aspects. *Figure 22.1* highlights the **salient features of an RL problem**. These typically include:

- Observations by the agent on the state of the environment
- A set of actions available to the agent
- A policy that governs the agent's decisions

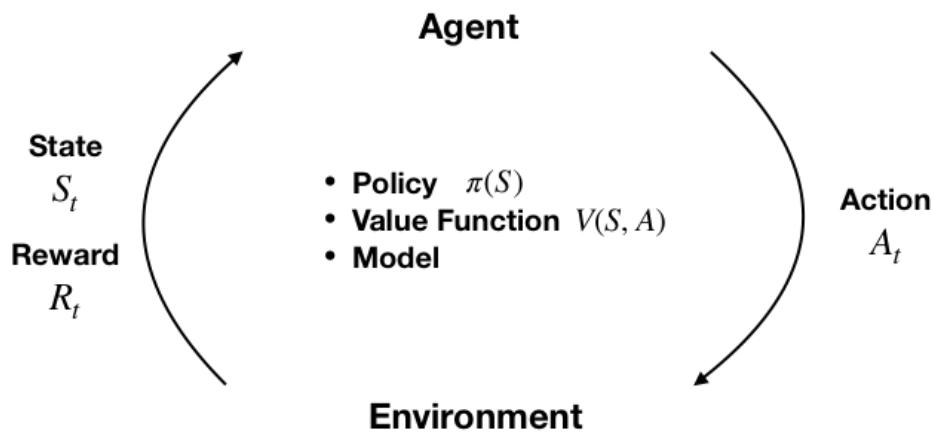


Figure 22.1: Components of an RL system

In addition, the environment emits a **reward signal** (that may be negative) as the agent's action leads to a transition to a new state. At its core, the agent usually learns a **value function** that informs its judgment of the available actions. The agent's objective function processes the reward signal and translates the value judgments into an optimal policy.

The policy – translating states into actions

At any point in time, **the policy defines the agent's behavior**. It maps any state the agent may encounter to one or several actions. In

an environment with a limited number of states and actions, the policy can be a simple lookup table that's filled in during training.

With continuous states and actions, the policy takes the form of a function that machine learning can help to approximate. The policy may also involve significant computation, as in the case of AlphaZero, which uses tree search to decide on the best action for a given game state. The policy may also be stochastic and assign probabilities to actions, given a state.

Rewards – learning from actions

The reward signal is a single value that the environment sends to the agent at each time step. The agent's objective is typically to **maximize the total reward received over time**. Rewards can also be a stochastic function of the state and the actions. They are typically discounted to facilitate convergence and reflect the time decay of value.

Rewards are the **only way for the agent to learn** about the value of its decisions in a given state and to modify the policy accordingly. Due to its critical impact on the agent's learning, the reward signal is often the most challenging part of designing an RL system.

Rewards need to clearly communicate what the agent should accomplish (as opposed to how it should do so) and may require domain knowledge to properly encode this information. For example, the development of a trading agent may need to define rewards for buy, hold, and sell decisions. These may be limited to profit and loss, but also may need to include volatility and risk considerations, such as drawdown.

The value function – optimal choice for the long run

The reward provides immediate feedback on actions. However, solving an RL problem requires decisions that create value in the long run. This is where the value function comes in: it summarizes the utility of states or of actions in a given state in terms of their long-term reward.

In other words, the value of a state is the total reward an agent can expect to obtain in the future when starting in that state. The immediate reward may be a good proxy of future rewards, but the agent also needs to account for cases where low rewards are followed by much better outcomes that are likely to follow (or the reverse).

Hence, **value estimates aim to predict future rewards**. Rewards are the key inputs, and the goal of making value estimates is to achieve more rewards. However, RL methods focus on learning accurate values that enable good decisions while efficiently leveraging the (often limited) experience.

There are also RL approaches that do not rely on value functions, such as randomized optimization methods like genetic algorithms or simulated annealing, which aim to find optimal behaviors by efficiently exploring the policy space. The current interest in RL, however, is mostly driven by methods that directly or indirectly estimate the value of states and actions.

Policy gradient methods are a new development that relies on a parameterized, differentiable policy that can be directly optimized with respect to the objective using gradient descent (Sutton et al.

2000). See the resources on GitHub that include abstracts of key papers and algorithms beyond the scope of this chapter.

With or without a model – look before you leap?

Model-based RL approaches learn a model of the environment to allow the agent to plan ahead by predicting the consequences of its actions. Such a model may be used, for example, to predict the next state and reward based on the current state and action. This is the **basis for planning**, that is, deciding on the best course of action by considering possible futures before they materialize.

Simpler **model-free methods**, in contrast, learn from **trial and error**. Modern RL methods span the gamut from low-level trial-and-error methods to high-level, deliberative planning. The right approach depends on the complexity and learnability of the environment.

How to solve reinforcement learning problems

RL methods aim to learn from experience how to take actions that achieve a long-term goal. To this end, the agent and the environment interact over a sequence of discrete time steps via the interface of actions, state observations, and rewards described in the previous section.

Key challenges in solving RL problems

Solving RL problems requires addressing two unique challenges: the credit-assignment problem and the exploration-exploitation trade-off.

Credit assignment

In RL, reward signals can occur significantly later than actions that contributed to the result, complicating the association of actions with their consequences. For example, when an agent takes 100 different positions and trades repeatedly, how does it realize that certain holdings performed much better than others if it only learns about the portfolio return?

The **credit-assignment problem** is the challenge of accurately estimating the benefits and costs of actions in a given state, despite these delays. RL algorithms need to find a way to distribute the credit for positive and negative outcomes among the many decisions that may have been involved in producing it.

Exploration versus exploitation

The dynamic and interactive nature of RL implies that the agent needs to estimate the value of the states and actions before it has experienced all relevant trajectories. While it is able to select an action at any stage, these decisions are based on incomplete learning, yet generate the agent's first insights into the optimal choices of its behavior.

Partial visibility into the value of actions creates the risk of decisions that only exploit past (successful) experience rather than exploring uncharted territory. Such choices limit the agent's exposure and prevent it from learning an optimal policy.

An RL algorithm needs to balance this **exploration-exploitation trade-off**—too little exploration will likely produce biased value estimates and suboptimal policies, whereas too little exploitation prevents learning from taking place in the first place.

Fundamental approaches to solving RL problems

There are numerous approaches to solving RL problems, all of which involve finding rules for the agent's optimal behavior:

- **Dynamic programming (DP)** methods make the often unrealistic assumption of complete knowledge of the environment, but they are the conceptual foundation for most other approaches.
- **Monte Carlo (MC)** methods learn about the environment and the costs and benefits of different decisions by sampling entire state-action-reward sequences.
- **Temporal difference (TD)** learning significantly improves sample efficiency by learning from shorter sequences. To this end, it relies on **bootstrapping**, which is defined as refining its estimates based on its own prior estimates.

When an RL problem includes well-defined transition probabilities and a limited number of states and actions, it can be framed as a finite **Markov decision process (MDP)** for which DP can compute an exact solution. Much of the current RL theory focuses on finite MDPs, but practical applications are used for (and require) more general settings. Unknown transition probabilities require efficient sampling to learn about their distribution.

Approaches to continuous state and/or action spaces often leverage **machine learning** to approximate a value or policy function. They integrate supervised learning and, in particular, deep learning methods like those discussed in the previous four chapters.

However, these methods face **distinct challenges** in the RL context:

- The **reward signal** does not directly reflect the target concept, like a labeled training sample.
- The **distribution of the observations** depends on the agent's actions and the policy, which is itself the subject of the learning process.

The following sections will introduce and demonstrate various solution methods. We'll start with the DP methods value iteration and policy iteration, which are limited to finite MDP with known transition probabilities. As we will see in the following section, they are the foundation for Q-learning, which is based on TD learning and does not require information about transition probabilities. It aims for similar outcomes as DP but with less computation and without assuming a perfect model of the environment. Finally, we'll expand the scope to continuous states and introduce deep Q-learning.

Solving dynamic programming problems

Finite MDPs are a simple yet fundamental framework. We will introduce the trajectories of rewards that the agent aims to optimize, define the policy and value functions used to formulate the

optimization problem, and the Bellman equations that form the basis for the solution methods.

Finite Markov decision problems

MDPs frame the agent-environment interaction as a sequential decision problem over a series of time steps $t = 1, \dots, T$ that constitute an episode. Time steps are assumed as discrete, but the framework can be extended to continuous time.

The abstraction afforded by MDPs makes its application easily adaptable to many contexts. The time steps can be at arbitrary intervals, and actions and states can take any form that can be expressed numerically.

The Markov property implies that the current state completely describes the process, that is, the process has no memory.

Information from past states adds no value when trying to predict the process's future. Due to these properties, the framework has been used to model asset prices subject to the efficient market hypothesis discussed in *Chapter 5, Portfolio Optimization and Performance Evaluation*.

Sequences of states, actions, and rewards

MDPs proceed in the following fashion: at each step t , the agent observes the environment's state $S_t \in S$ and selects an action $A_t \in A$, where S and A are the sets of states and actions, respectively. At the next time step $t+1$, the agent receives a reward $R_{t+1} \in R$ and transitions to state S_{t+1} . Over time, the MDP gives rise to a trajectory

$S_0, A_0, R_1, S_1, A_1, R_1, \dots$ that continues until the agent reaches a terminal state and the episode ends.

Finite MDPs with a limited number of actions A , states S , and rewards R include well-defined discrete probability distributions over these elements. Due to the Markov property, these distributions only depend on the previous state and action.

The probabilistic nature of trajectories implies that the agent maximizes the expected sum of future rewards. Furthermore, rewards are typically discounted using a factor $0 \leq \gamma \leq 1$ to reflect their time value. In the case of tasks that are not episodic but continue indefinitely, a discount factor strictly less than 1 is necessary to avoid infinite rewards and ensure convergence. Therefore, the agent maximizes the discounted, expected sum of future returns R_t , denoted as G_t :

$$G_t = \mathbf{E}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots] = \sum_{s=0}^T \gamma^s \mathbf{E}[R_{t+s}]$$

This relationship can also be defined recursively because the sum starting at the second step is the same as G_{t+1} discounted once:

$$G_t = R_{t+1} + \gamma G_{t+1}$$

We will see later that this type of recursive relationship is frequently used to formulate RL algorithms.

Value functions – how to estimate the long-run reward

As introduced previously, a policy π maps all states to probability distributions over actions so that the probability of choosing action

A_t in state S_t can be expressed as $\pi(a|s) = P(A_t = a|S_t = s)$. The value function estimates the long-run return for each state or state-action pair. It is fundamental to find the policy that is the optimal mapping of states to actions.

The state-value function $v_\pi(s)$ for policy π gives the long-term value v of a specific state s as the expected return G for an agent that starts in s and then always follows policy π . It is defined as follows, where E_π refers to the expected value when the agent follows policy π :

$$v_\pi(s) = E_\pi[G_t|S_t = s] = E_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | s_t = s \right]$$

Similarly, we can compute the **state-action value function** $q(s,a)$ as the expected return of starting in state s , taking action, and then always following the policy π :

$$q_\pi(s, a) = E_\pi[G_t|S_t = s, A_t = a] = E_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | s_t = s, A_t = a \right]$$

The Bellman equations

The Bellman equations define a recursive relationship between the value functions for all states s in S and any of their successor states s' under a policy π . They do so by decomposing the value function into the immediate reward and the discounted value of the next state:

$$v_\pi(s) \doteq \mathbf{E}[G_t | S_t = s]$$

$$\begin{aligned}
 &= \mathbf{E} \left[\underbrace{R_{t+1}}_{\text{reward}} + \underbrace{\gamma v_\pi(S_{t+1})}_{\text{discounted value}} \right] \\
 &= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r | s, a) [r + \gamma v_\pi(s')] \quad \forall_s
 \end{aligned}$$

This equation says that for a given policy, the value of a state must equal the expected value of its successor states under the policy, plus the expected reward earned from arriving at that successor state.

This implies that, if we know the values of the successor states for the currently available actions, we can look ahead one step and compute the expected value of the current state. Since it holds for all states S , the expression defines a set of $n = |S|$ equations. An analogous relationship holds for $q(s, a)$.

Figure 22.2 summarizes this recursive relationship: in the current state, the agent selects an action a based on the policy π . The environment responds by assigning a reward that depends on the resulting new state s' :

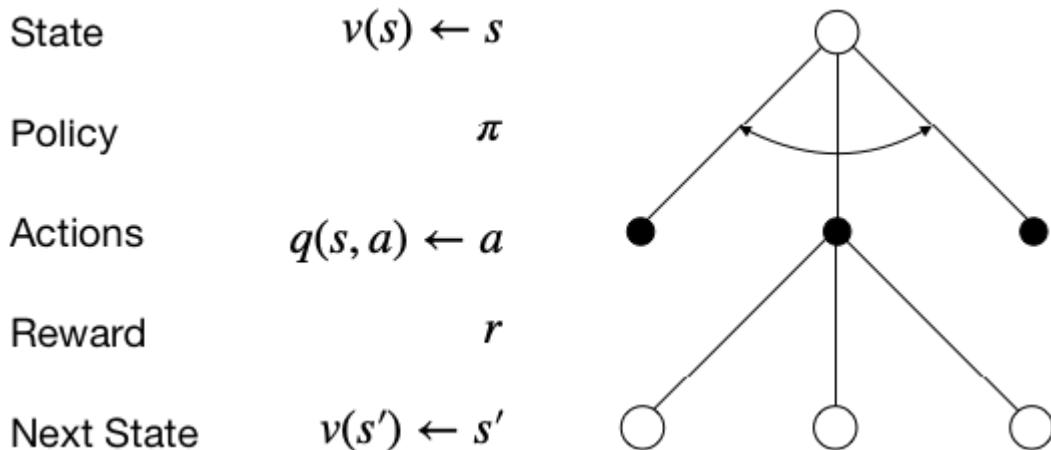


Figure 22.2: The recursive relationship expressed by the Bellman equation

From a value function to an optimal policy

The solution to an RL problem is a policy that optimizes the cumulative reward. Policies and value functions are closely connected: an optimal policy yields a value estimate for each state $v_\pi(s)$ or state-action pair $q_\pi(s, a)$ that is at least as high as for any other policy since the value is the cumulative reward under the given policy. Hence, the optimal value functions $v^*(s) = \max_\pi v_\pi(s)$ and $q^*(s, a) = \max_\pi q_\pi(s, a)$ implicitly define optimal policies and solve the MDP.

The optimal value functions v^* and q^* also satisfy the Bellman equations from the previous section. These Bellman optimality equations can omit the explicit reference to a policy as it is implied by v^* and q^* . For $v^*(s)$, the recursive relationship equates the current value to the sum of the immediate reward from choosing the best action in the current state, as well as the expected discounted value of the successor states:

$$v^*(s) = \max_a q^*(s, a) = \max_a R_t + \gamma \sum_{s'} p(s'|s, a) v^*(s')$$

For the optimal state-action value function $q^*(s, a)$, the Bellman optimality equation decomposes the current state-action value into the sum of the reward for the implied current action and the discounted expected value of the best action in all successor states:

$$q^*(s) = R_t + \gamma \sum_{s'} p(s'|s, a) v^*(s) = R_t + \gamma \sum_{s'} p(s'|s, a) \max_a q^*(s, a)$$

The optimality conditions imply that the best policy is to always select the action that maximizes the expected value in a greedy fashion, that is, to only consider the result of a single time step.

The optimality conditions defined by the two previous expressions are nonlinear due to the max operator and lack a closed-form solution. Instead, MDP solutions rely on an iterative solution - like policy and value iteration or Q-learning, which we will cover next.

Policy iteration

DP is a general method for solving problems that can be decomposed into smaller, overlapping subproblems with a recursive structure that permit the reuse of intermediate results. MDPs fit the bill due to the recursive Bellman optimality equations and the cumulative nature of the value function. More specifically, the **principle of optimality** applies because an optimal policy consists of picking an optimal action and then following an optimal policy.

DP requires knowledge of the MDP's transition probabilities. This is often not the case, but many methods for more general cases follow an approach similar to DP and learn the missing information from the data.

DP is useful for **prediction tasks** that estimate the value function and the control task that focuses on optimal decisions and outputs a policy (while also estimating a value function in the process).

The policy iteration algorithm to find an optimal policy repeats the following two steps until the policy has converged, that is, no longer changes more than a given threshold:

1. **Policy evaluation:** Update the value function based on the current policy.
2. **Policy improvement:** Update the policy so that actions maximize the expected one-step value.

Policy evaluation relies on the Bellman equation to estimate the value function. More specifically, it selects the action determined by the current policy and sums the resulting reward, as well as the discounted value of the next state, to update the value for the current state.

Policy improvement, in turn, alters the policy so that for each state, the policy produces the action that produces the highest value in the next state. This improvement is called greedy because it only considers the return of a single time step. Policy iteration always converges to an optimal policy and often does so in relatively few iterations.

Value iteration

Policy iteration requires the evaluation of the policy for all states after each iteration. The evaluation can be costly, as discussed previously, for search-tree-based policies, for example.

Value iteration simplifies this process by collapsing the policy evaluation and improvement step. At each time step, it iterates over all states and selects the best greedy action based on the current value estimate for the next state. Then, it uses the one-step lookahead implied by the Bellman optimality equation to update the value function for the current state.

The corresponding update rule for the value function $v_{k+1}(s)$ is almost identical to the policy evaluation update; it just adds the maximization over the available actions:

$$v_{k+1}(s) \leftarrow \max_a \sum_{s'} \sum_r p(s', r | s, a) [r + \gamma v_k(s')]$$

The algorithm stops when the value function has converged and outputs the greedy policy derived from its value function estimate. It is also guaranteed to converge to an optimal policy.

Generalized policy iteration

In practice, there are several ways to truncate policy iteration; for example, by evaluating the policy k times before improving it. This just means that the \max operator will only be applied at every k^{th} iteration.

Most RL algorithms estimate value and policy functions and rely on the interaction of policy evaluation and improvement to converge to a solution, as illustrated in *Figure 22.3*. The general approach improves the policy with respect to the value function while adjusting the value function so that it matches the policy:

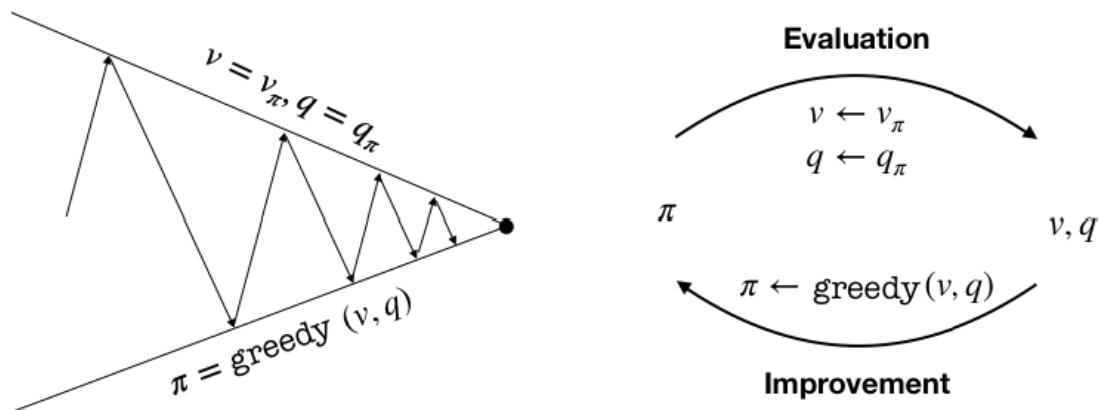


Figure 22.3: Convergence of policy evaluation and improvement

Convergence requires that the value function be consistent with the policy, which, in turn, needs to stabilize while acting greedily with respect to the value function. Thus, both processes stabilize only when a policy has been found that is greedy with respect to its own evaluation function. This implies that the Bellman optimality equation holds, and thus that the policy and the value function are optimal.

Dynamic programming in Python

In this section, we'll apply value and policy iteration to a toy environment that consists of a 3×4 grid, as depicted in *Figure 22.4*, with the following features:

- **States:** 11 states represented as two-dimensional coordinates. One field is not accessible and the top two states in the right-most column are terminal, that is, they end the episode.
- **Actions:** Movements of one step up, down, left, or right. The environment is randomized so that actions can have unintended outcomes. For each action, there is an 80 percent probability of moving to the expected state, and 10 percent each of moving in an adjacent direction (for example, right or left instead of up, or up/down instead of right).
- **Rewards:** As depicted in the left panel, each state results in -.02 except the +1/-1 rewards in the terminal states.

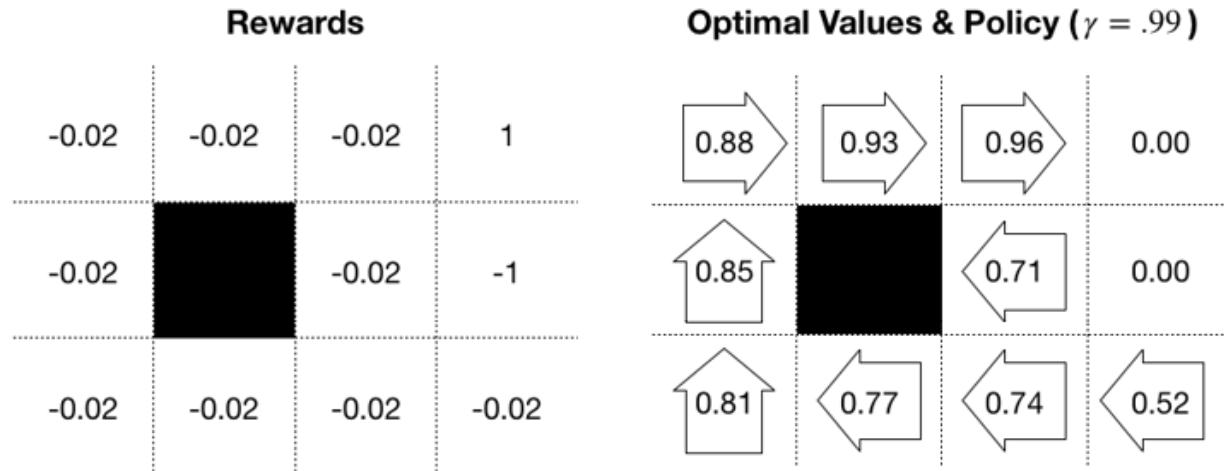


Figure 22.4: 3×4 gridworld rewards, value function, and optimal policy

Setting up the gridworld

We will begin by defining the environment parameters:

```
grid_size = (3, 4)
blocked_cell = (1, 1)
baseline_reward = -0.02
absorbing_cells = {(0, 3): 1, (1, 3): -1}
actions = ['L', 'U', 'R', 'D']
num_actions = len(actions)
probs = [.1, .8, .1, 0]
```

We will frequently need to convert between 1D and 2D representations, so we will define two helper functions for this purpose; states are one-dimensional, and cells are the corresponding 2D coordinates:

```
to_1d = lambda x: np.ravel_multi_index(x, grid_size)
to_2d = lambda x: np.unravel_index(x, grid_size)
```

Furthermore, we will precompute some data points to make the code more concise:

```

num_states = np.product(grid_size)
cells = list(np.ndindex(grid_size))
states = list(range(len(cells)))
cell_state = dict(zip(cells, states))
state_cell = dict(zip(states, cells))
absorbing_states = {to_1d(s):r for s, r in absorbing_cells.items()}
blocked_state = to_1d(blocked_cell)

```

We store the rewards for each state:

```

state_rewards = np.full(num_states, baseline_reward)
state_rewards[blocked_state] = 0
for state, reward in absorbing_states.items():
    state_rewards[state] = reward
state_rewards
array([-0.02, -0.02, -0.02,  1.  , -0.02,  0.  , -0.02, -1.  , -0.02,
       -0.02, -0.02, -0.02])

```

To account for the probabilistic environment, we also need to compute the probability distribution over the actual move for a given action:

```

action_outcomes = {}
for i, action in enumerate(actions):
    probs_ = dict(zip([actions[j % 4] for j in range(i,
                                                    num_actions + i)], prob
    action_outcomes[actions[(i + 1) % 4]] = probs_
Action_outcomes
{'U': {'L': 0.1, 'U': 0.8, 'R': 0.1, 'D': 0}, 'R': {'U': 0.1, 'R': 0.8, 'D': 0.1, 'L': 0}, 'D': {'R': 0.1, 'D': 0.8, 'L': 0.1, 'U': 0}, 'L': {'D': 0.1, 'L': 0.8, 'U': 0.1, 'R': 0}}

```

Now, we are ready to compute the transition matrix, which is the key input to the MDP.

Computing the transition matrix

The **transition matrix** defines the probability of ending up in a certain state S for each previous state and action A $P(s'|s, a)$. We will demonstrate `pymdptoolbox` and use one of the formats available to specify transitions and rewards. For both transition probabilities, we will create a NumPy array with dimensions $A \times S \times S$.

We first compute the target cell for each starting cell and move:

```
def get_new_cell(state, move):
    cell = to_2d(state)
    if actions[move] == 'U':
        return cell[0] - 1, cell[1]
    elif actions[move] == 'D':
        return cell[0] + 1, cell[1]
    elif actions[move] == 'R':
        return cell[0], cell[1] + 1
    elif actions[move] == 'L':
        return cell[0], cell[1] - 1
```

The following function uses the arguments starting `state`, `action`, and `outcome` to fill in the transition probabilities and rewards:

```
def update_transitions_and_rewards(state, action, outcome):
    if state in absorbing_states.keys() or state == blocked_state:
        transitions[action, state, state] = 1
    else:
        new_cell = get_new_cell(state, outcome)
        p = action_outcomes[actions[action]][actions[outcome]]
        if new_cell not in cells or new_cell == blocked_cell:
            transitions[action, state, state] += p
            rewards[action, state, state] = baseline_reward
        else:
            new_state = to_1d(new_cell)
            transitions[action, state, new_state] = p
            rewards[action, state, new_state] = state_rewards[new_state]
```

We generate the transition and reward values by creating placeholder data structures and iterating over the Cartesian product

of $A \times S \times S$, as follows:

```
rewards = np.zeros(shape=(num_actions, num_states, num_states))
transitions = np.zeros((num_actions, num_states, num_states))
actions_ = list(range(num_actions))
for action, outcome, state in product(actions_, actions_, states):
    update_transitions_and_rewards(state, action, outcome)
rewards.shape, transitions.shape
((4,12,12), (4,12,12))
```

Implementing the value iteration algorithm

We first create the value iteration algorithm, which is slightly simpler because it implements policy evaluation and improvement in a single step. We capture the states for which we need to update the value function, excluding terminal states that have a value of 0 for lack of rewards (+1/-1 are assigned to the starting state), and skip the blocked cell:

```
skip_states = list(absorbing_states.keys())+[blocked_state]
states_to_update = [s for s in states if s not in skip_states]
```

Then, we initialize the value function and set the discount factor gamma and the convergence threshold `epsilon`:

```
V = np.random.rand(num_states)
V[skip_states] = 0
gamma = .99
epsilon = 1e-5
```

The algorithm updates the value function using the Bellman optimality equation, as described previously, and terminates when the L1 norm of V changes to less than ϵ in absolute terms:

```

while True:
    V_ = np.copy(V)
    for state in states_to_update:
        q_sa = np.sum(transitions[:, state] * (rewards[:, state] + gamma * V_), axis=1)
        V[state] = np.max(q_sa)
    if np.sum(np.fabs(V - V_)) < epsilon:
        break

```

The algorithm converges in 16 iterations and 0.0117s. It produces the following optimal value estimate, which, together with the implied optimal policy, is depicted in the right panel of *Figure 22.4*, earlier in this section:

```

pd.DataFrame(V.reshape(grid_size))
   0      1      2      3
0 0.884143 0.925054 0.961986 0.000000
1 0.848181 0.000000 0.714643 0.000000
2 0.808344 0.773327 0.736099 0.516082

```

Defining and running policy iteration

Policy iterations involve separate evaluation and improvement steps. We define the improvement part by selecting the action that maximizes the sum of the expected reward and next-state value. Note that we temporarily fill in the rewards for the terminal states to avoid ignoring actions that would lead us there:

```

def policy_improvement(value, transitions):
    for state, reward in absorbing_states.items():
        value[state] = reward
    return np.argmax(np.sum(transitions * value, 2), 0)

```

We initialize the value function as before and also include a random starting policy:

```
pi = np.random.choice(list(range(num_actions)), size=num_states)
```

The algorithm alternates between policy evaluation for a greedily selected action and policy improvement until the policy stabilizes:

```
iterations = 0
converged = False
while not converged:
    pi_ = np.copy(pi)
    for state in states_to_update:
        action = policy[state]
        V[state] = np.dot(transitions[action, state],
                           rewards[action, state] + gamma*
                           pi = policy_improvement(V.copy(), transitions)
    if np.array_equal(pi_, pi):
        converged = True
    iterations += 1
```

Policy iteration converges after only three iterations. The policy stabilizes before the algorithm finds the optimal value function, and the optimal policy differs slightly, most notably by suggesting "up" instead of the safer "left" for the field next to the negative terminal state. This can be avoided by tightening the convergence criteria, for example, by requiring a stable policy of several rounds or by adding a threshold for the value function.

Solving MDPs using `pymdptoolbox`

We can also solve MDPs using the Python library `pymdptoolbox`, which includes a few other algorithms, including Q-learning.

To run value iteration, just instantiate the corresponding object with the desired configuration options, rewards, and transition matrices before calling the `.run()` method:

```
vi = mdp.ValueIteration(transitions=transitions,
                        reward=rewards,
                        discount=gamma,
                        epsilon=epsilon)
vi.run()
```

The value function estimate matches the result in the previous section:

```
np.allclose(V.reshape(grid_size), np.asarray(vi.V).reshape(grid_size))
```

Policy iteration works similarly:

```
pi = mdp.PolicyIteration(transitions=transitions,
                          reward=rewards,
                          discount=gamma,
                          max_iter=1000)
pi.run()
```

It also yields the same policy, but the value function varies by run and does not need to achieve the optimal value before the policy converges.

Lessons learned

The right panel we saw earlier in *Figure 22.4* shows the optimal value estimate produced by value iteration and the corresponding greedy policy. The negative rewards, combined with the uncertainty in the environment, produce an optimal policy that involves moving away from the negative terminal state.

The results are sensitive to both the rewards and the discount factor. The cost of the negative state affects the policy in the surrounding fields, and you should modify the example in the corresponding

notebook to identify threshold levels that alter the optimal action selection.

Q-learning – finding an optimal policy on the go

Q-learning was an early RL breakthrough when developed by Chris Watkins for his PhD thesis

(http://www.cs.rhul.ac.uk/~chrisw/new_thesis.pdf) (1989).

It introduces incremental dynamic programming to learn to control an MDP without knowing or modeling the transition and reward matrices that we used for value and policy iteration in the previous section. A convergence proof followed 3 years later (Christopher J. C. H. Watkins and Dayan 1992).

Q-learning directly optimizes the action-value function q to approximate q^* . The learning proceeds "off-policy," that is, the algorithm does not need to select actions based on the policy implied by the value function alone. However, convergence requires that all state-action pairs continue to be updated throughout the training process. A straightforward way to ensure this is through an ϵ -greedy policy.

Exploration versus exploitation – ϵ -greedy policy

An **ϵ -greedy policy** is a simple policy that ensures the exploration of new actions in a given state while also exploiting the learning

experience . It does this by randomizing the selection of actions. An ε -greedy policy selects an action randomly with a probability of ε , and the best action according to the value function otherwise.

The Q-learning algorithm

The algorithm keeps improving a state-action value function after random initialization for a given number of episodes. At each time step, it chooses an action based on an ε -greedy policy, and uses a learning rate α to update the value function, as follows:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[\underbrace{R_t + \gamma \max_a Q(S_{t+1}, a)}_{\text{TD Target}} - \underbrace{Q(S_t, A_t)}_{\text{Current Q-value}} \right]$$

Temporal Difference

Note that the algorithm does not compute expected values based on the transition probabilities. Instead, it learns the Q function from the rewards R_t produced by the ε -greedy policy and its current estimate of the discounted value function for the next state.

The use of the estimated value function to improve this very estimate is called **bootstrapping**. The Q-learning algorithm is part of the **temporal difference (TD) learning** algorithms. TD learning does not wait until receiving the final reward for an episode. Instead, it updates its estimates using the values of intermediate states that are closer to the final reward. In this case, the intermediate state is one time step ahead.

How to train a Q-learning agent using Python

In this section, we will demonstrate how to build a Q-learning agent using the 3×4 grid of states from the previous section. We will train the agent for 2,500 episodes, using a learning rate of $\alpha = 0.1$ and $\varepsilon = 0.05$ for the ε -greedy policy (see the notebook

`gridworld_q_learning.ipynb` for details):

```
max_episodes = 2500
alpha = .1
epsilon = .05
```

Then, we will randomly initialize the state-action value function as a NumPy array with dimensions *number of states* \times *number of actions*:

```
Q = np.random.rand(num_states, num_actions)
Q[skip_states] = 0
```

The algorithm generates 2,500 episodes that start at a random location and proceed according to the ε -greedy policy until termination, updating the value function according to the Q-learning rule:

```
for episode in range(max_episodes):
    state = np.random.choice([s for s in states if s not in skip_states])
    while not state in absorbing_states.keys():
        if np.random.rand() < epsilon:
            action = np.random.choice(num_actions)
        else:
            action = np.argmax(Q[state])
        next_state = np.random.choice(states, p=transitions[action, state])
        reward = rewards[action, state, next_state]
        Q[state, action] += alpha * (reward +
                                     gamma * np.max(Q[next_state]) - Q[state, action])
        state = next_state
```

The episodes take 0.6 seconds and converge to a value function fairly close to the result of the value iteration example from the previous section. The `pymdptoolbox` implementation works analogously to previous examples (see the notebook for details).

Deep RL for trading with the OpenAI Gym

In the previous section, we saw how Q-learning allows us to learn the optimal state-action value function q^* in an environment with discrete states and discrete actions using iterative updates based on the Bellman equation.

In this section, we will take RL one step closer to the real world and upgrade the algorithm to **continuous states** (while keeping actions discrete). This implies that we can no longer use a tabular solution that simply fills an array with state-action values. Instead, we will see how to **approximate q^* using a neural network (NN)**, which results in a deep Q-network. We will first discuss how deep learning integrates with RL before presenting the deep Q-learning algorithm, as well as various refinements that accelerate its convergence and make it more robust.

Continuous states also imply a **more complex environment**. We will demonstrate how to work with OpenAI Gym, a toolkit for designing and comparing RL algorithms. First, we'll illustrate the workflow by training a deep Q-learning agent to navigate a toy spaceship in the Lunar Lander environment. Then, we'll proceed to **customize OpenAI Gym** to design an environment that simulates a trading

context where an agent can buy and sell a stock while competing against the market.

Value function approximation with neural networks

Continuous state and/or action spaces imply an **infinite number of transitions** that make it impossible to tabulate the state-action values, as in the previous section. Rather, we approximate the Q function by learning a continuous, parameterized mapping from training samples.

Motivated by the success of NNs in other domains, which we discussed in the previous chapters in *Part 4*, deep NNs have also become popular for approximating value functions. However, **machine learning in the RL context**, where the data is generated by the interaction of the model with the environment using a (possibly randomized) policy, **faces distinct challenges**:

- With continuous states, the agent will fail to visit most states and thus needs to generalize.
- Whereas supervised learning aims to generalize from a sample of independently and identically distributed samples that are representative and correctly labeled, in the RL context, there is only one sample per time step, so learning needs to occur online.
- Furthermore, samples can be highly correlated when sequential states are similar and the behavior distribution over states and actions is not stationary, but rather changes as a result of the agent's learning.

We will look at several techniques that have been developed to address these additional challenges.

The Deep Q-learning algorithm and extensions

Deep Q-learning estimates the value of the available actions for a given state using a deep neural network. DeepMind introduced this technique in *Playing Atari with Deep Reinforcement Learning* (Mnih et al. 2013), where agents learned to play games solely from pixel input.

The Deep Q-learning algorithm approximates the action-value function q by learning a set of weights θ of a multilayered **deep Q-network (DQN)** that maps states to actions so that $q(s, a; \theta) \approx q^*(s, a)$.

The algorithm applies gradient descent based on a loss function that computes the squared difference between the DQN's estimate of the target:

$$y_i = \mathbb{E} \left[r + \gamma \max_{a'} Q(s', a'; \theta_{i-1} | s, a) \right]$$

and its estimate of the action-value of the current state-action pair $Q(s, a; \theta)$ to learn the network parameters:

$$L_i(\theta_i) = \left(\frac{\underbrace{y_i}_{\text{Q Target}} - \underbrace{Q(s, a; \theta)}_{\text{Current Prediction}}}_{\text{TD Error}} \right)^2$$

Both the **target and the current estimate depend on the DQN weights**, underlining the distinction from supervised learning where

targets are fixed prior to training.

Rather than computing the full gradient, the Q-learning algorithm uses **stochastic gradient descent (SGD)** and updates the weights θ_i after each time step i . To explore the state-action space, the agent uses an ϵ -greedy policy that selects a random action with probability ϵ and follows a greedy policy that selects the action with the highest predicted q -value otherwise.

The basic **DQN architecture has been refined** in several directions to make the learning process more efficient and improve the final result; Hessel et al. (2017) combined these innovations in the **Rainbow agent** and demonstrated how each contributes to significantly higher performance across the Atari benchmarks. The following subsections summarize some of these innovations.

(Prioritized) Experience replay – focusing on past mistakes

Experience replay stores a history of the state, action, reward, and next state transitions experienced by the agent. It randomly samples mini-batches from this experience to update the network weights at each time step before the agent selects an ϵ -greedy action.

Experience replay increases sample efficiency, reduces the autocorrelation of samples collected during online learning, and limits the feedback due to current weights producing training samples that can lead to local minima or divergence (Lin and Mitchell 1992).

This technique was later refined to prioritize experience that is more important from a learning perspective. Schaul et al. (2015) approximated the value of a transition by the size of the TD error

that captures how "surprising" the event was for the agent. In practice, it samples historical state transitions using their associated TD error rather than uniform probabilities.

The target network – decorrelating the learning process

To further weaken the feedback loop from the current network parameters on the NN weight updates, the algorithm was extended by DeepMind in *Human-level control through deep reinforcement learning* (Mnih et al. 2015) to use a slowly-changing target network.

The target network has the same architecture as the Q-network, but its weights θ^- are only updated periodically after τ steps when they are copied from the Q-network and held constant otherwise. The target network **generates the TD target predictions**, that is, it takes the place of the Q-network to estimate:

$$y_i = \mathbb{E} \left[r + \gamma \max_{a'} Q(s', a'; \theta^- | s, a) \right]$$

Double deep Q-learning – decoupling action and prediction

Q-learning has been shown to overestimate the action values because it purposely samples maximal estimated action values.

This bias can negatively affect the learning process and the resulting policy if it does not apply uniformly and alters action preferences, as shown in *Deep Reinforcement Learning with Double Q-learning* (van Hasselt, Guez, and Silver 2015).

To decouple the estimation of action values from the selection of actions, the **Double DQN (DDQN)** algorithm uses the weights θ of

one network to select the best action given the next state, as well as the weights θ' of another network, to provide the corresponding action value estimate:

$$y_i = \mathbb{E} \left[r + \gamma Q \left(s', \arg \max_{a'} Q(S_{t+1}, a, \theta_t); \theta'_t \right) \right].$$

One option is to randomly select one of two identical networks for training at each iteration so that their weights will differ. A more efficient alternative is to rely on the target network to provide θ' instead.

Introducing the OpenAI Gym

OpenAI Gym is an RL platform that provides standardized environments to test and benchmark RL algorithms using Python. It is also possible to extend the platform and register custom environments.

The **Lunar Lander v2 (LL)** environment requires the agent to control its motion in two dimensions based on a discrete action space and low-dimensional state observations that include position, orientation, and velocity. At each time step, the environment provides an observation of the new state and a positive or negative reward. Each episode consists of up to 1,000 time steps. *Figure 22.5* shows selected frames from a successful landing after 250 episodes by the agent we will train later:

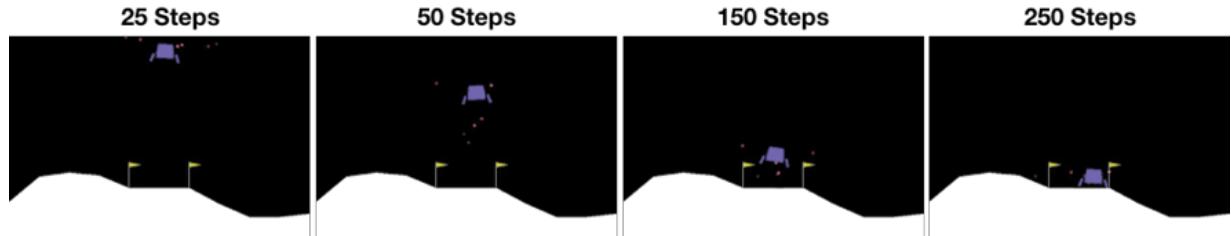


Figure 22.5: RL agent's behavior during the Lunar Lander episode

More specifically, the **agent observes eight aspects of the state**, including six continuous and two discrete elements. Based on the observed elements, the agent knows its location, direction, and speed of movement and whether it has (partially) landed. However, it does not know in which direction it should move, nor can it observe the inner state of the environment to understand the rules that govern its motion.

At each time step, the agent controls its motion using one of **four discrete actions**. It can do nothing (and continue on its current path), fire its main engine (to reduce downward motion), or steer toward the left or right using the respective orientation engines. There are no fuel limitations.

The goal is to land the agent between two flags on a landing pad at coordinates $(0, 0)$, but landing outside of the pad is possible. The agent accumulates rewards in the range of 100-140 for moving toward the pad, depending on the exact landing spot. However, a move away from the target negates the reward the agent would have gained by moving toward the pad. Ground contact by each leg adds 10 points, while using the main engine costs -0.3 points.

An episode terminates if the agent lands or crashes, adding or subtracting 100 points, respectively, or after 1,000 time steps. Solving

LL requires achieving a cumulative reward of at least 200 on average over 100 consecutive episodes.

How to implement DDQN using TensorFlow 2

The notebook `03_lunar_lander_deep_q_learning` implements a DDQN agent using TensorFlow 2 that learns to solve OpenAI Gym's **Lunar Lander** 2.0 (LL) environment. The notebook

`03_lunar_lander_deep_q_learning` contains a TensorFlow 1 implementation that was discussed in the first edition and runs significantly faster because it does not rely on eager execution and also converges sooner. This section highlights key elements of the implementation; please see the notebook for much more extensive details.

Creating the DDQN agent

We create our `DDQNAgent` as a Python class to integrate the learning and execution logic with the key configuration parameters and performance tracking.

The agent's `__init__()` method takes, as arguments, information on:

- The **environment characteristics**, like the number of dimensions for the state observations and the number of actions available to the agent.
- The decay of the randomized exploration for the **ϵ -greedy policy**.
- The **neural network architecture** and the parameters for **training** and target network updates.

```
class DDQNAgent:  
    def __init__(self, state_dim, num_actions, gamma,  
                 epsilon_start, epsilon_end, epsilon_decay_steps,  
                 epsilon_exp_decay, replay_capacity, learning_rate  
                 architecture, l2_reg, tau, batch_size,  
                 log_dir='results'):
```

Adapting the DDQN architecture to the Lunar Lander

The DDQN architecture was first applied to the Atari domain with high-dimensional image observations and relied on convolutional layers. The LL's lower-dimensional state representation makes fully connected layers a better choice (see *Chapter 17, Deep Learning for Trading*).

More specifically, the network maps eight inputs to four outputs, representing the Q values for each action, so that it only takes a single forward pass to compute the action values. The DQN is trained on the previous loss function using the Adam optimizer. The agent's DQN uses three densely connected layers with 256 units each and L2 activity regularization. Using a GPU via the TensorFlow Docker image can significantly speed up NN training performance (see *Chapter 17* and *Chapter 18, CNNs for Financial Time Series and Satellite Images*).

The `DDQNAgent` class's `build_model()` method creates the primary online and slow-moving target networks based on the `architecture` parameter, which specifies the number of layers and their number of units.

We set `trainable` to `True` for the primary online network and to `False` for the target network. This is because we simply periodically copy the online NN weights to update the target network:

```

def build_model(self, trainable=True):
    layers = []
    for i, units in enumerate(self.architecture, 1):
        layers.append(Dense(units=units,
                            input_dim=self.state_dim if i == 1 else
                            activation='relu',
                            kernel_regularizer=l2(self.l2_reg),
                            trainable=trainable))
    layers.append(Dense(units=self.num_actions,
                        trainable=trainable))
    model = Sequential(layers)
    model.compile(loss='mean_squared_error',
                  optimizer=Adam(lr=self.learning_rate))
    return model

```

Memorizing transitions and replaying the experience

To enable experience replay, the agent memorizes each state transition so it can randomly sample a mini-batch during training. The `memorize_transition()` method receives the observation on the current and next state provided by the environment, as well as the agent's action, the reward, and a flag that indicates whether the episode is completed.

It tracks the reward history and length of each episode, applies exponential decay to epsilon at the end of each period, and stores the state transition information in a buffer:

```

def memorize_transition(self, s, a, r, s_prime, not_done):
    if not_done:
        self.episode_reward += r
        self.episode_length += 1
    else:
        self.episodes += 1
        self.rewards_history.append(self.episode_reward)
        self.steps_per_episode.append(self.episode_length)
        self.episode_reward, self.episode_length = 0, 0
    self.experience.append((s, a, r, s_prime, not_done))

```

The replay of the memorized experience begins as soon as there are enough samples to create a full batch. The `experience_replay()` method predicts the Q values for the next states using the online network and selects the best action. It then selects the predicted q values for these actions from the target network to arrive at the TD targets.

Next, it trains the primary network using a single batch of current state observations as input, the TD targets as the outcome, and the mean-squared error as the loss function. Finally, it updates the target network weights every τ steps:

```
def experience_replay(self):
    if self.batch_size > len(self.experience):
        return
    # sample minibatch from experience
    minibatch = map(np.array, zip(*sample(self.experience,
                                           self.batch_size)))
    states, actions, rewards, next_states, not_done = minibatch
    # predict next Q values to select best action
    next_q_values = self.online_network.predict_on_batch(next_stat
    best_actions = tf.argmax(next_q_values, axis=1)
    # predict the TD target
    next_q_values_target = self.target_network.predict_on_batch(
        next_states)
    target_q_values = tf.gather_nd(next_q_values_target,
                                   tf.stack((self.idx, tf.cast(
                                       best_actions, tf.int32))), a>
    targets = rewards + not_done * self.gamma * target_q_values
    # predict q values
    q_values = self.online_network.predict_on_batch(states)
    q_values[[self.idx, actions]] = targets
    # train model
    loss = self.online_network.train_on_batch(x=states, y=q_values)
    self.losses.append(loss)
    if self.total_steps % self.tau == 0:
        self.update_target()
def update_target(self):
    self.target_network.set_weights(self.online_network.get_weight
```

The notebook contains additional implementation details for the ϵ -greedy policy and the target network weight updates.

Setting up the OpenAI environment

We will begin by instantiating and extracting key parameters from the LL environment:

```
env = gym.make('LunarLander-v2')
state_dim = env.observation_space.shape[0] # number of dimensions in
num_actions = env.action_space.n # number of actions
max_episode_steps = env.spec.max_episode_steps # max number of steps
env.seed(42)
```

We will also use the built-in wrappers that permit the periodic storing of videos that display the agent's performance:

```
from gym import wrappers
env = wrappers.Monitor(env,
                      directory=monitor_path.as_posix(),
                      video_callable=lambda count: count % video_freq
                      == 0,
                      force=True)
```

When running on a server or Docker container without a display, you can use `pyvirtualdisplay`.

Key hyperparameter choices

The agent's performance is quite sensitive to several hyperparameters. We will start with the discount and learning rates:

```
gamma=.99, # discount factor
learning_rate=1e-4 # Learning rate
```

We will update the target network every 100 time steps, store up to 1 million past episodes in the replay memory, and sample mini-batches of 1,024 from memory to train the agent:

```
tau=100  # target network update frequency
replay_capacity=int(1e6)
batch_size = 1024
```

The ε -greedy policy starts with pure exploration at $\varepsilon = 1$, linear decay to 0.01 over 250 episodes, and exponential decay thereafter:

```
epsilon_start=1.0
epsilon_end=0.01
epsilon_linear_steps=250
epsilon_exp_decay=0.99
```

The notebook contains the training loop, including experience replay, SGD, and slow target network updates.

Lunar Lander learning performance

The preceding hyperparameter settings enable the agent to solve the environment in around 300 episodes using the TensorFlow 1 implementation.

The left panel of *Figure 22.6* shows the episode rewards and their moving average over 100 periods. The right panel shows the decay of exploration and the number of steps per episode. There is a stretch of some 100 episodes that often take 1,000 time steps each while the agent reduces exploration and "learns how to fly" before starting to land fairly consistently:

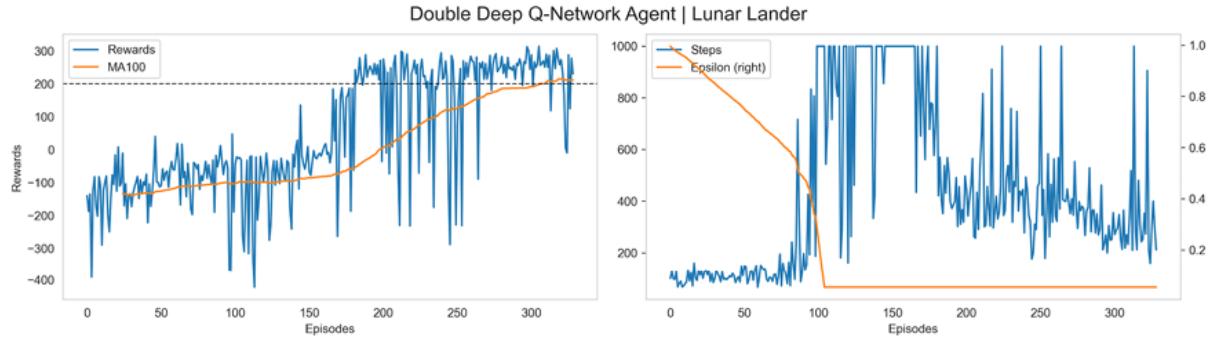


Figure 22.6: The DDQN agent's performance in the Lunar Lander environment

Creating a simple trading agent

In this and the following sections, we will adapt the deep RL approach to design an agent that learns how to trade a single asset. To train the agent, we will set up a simple environment with a limited set of actions, a relatively low-dimensional state with continuous observations, and other parameters.

More specifically, the **environment** samples a stock price time series for a single ticker using a random start date to simulate a trading period that, by default, contains 252 days or 1 year. Each **state observation** provides the agent with the historical returns for various lags and some technical indicators, like the **relative strength index (RSI)**.

The agent can choose from three **actions**:

- **Buy**: Invest all capital for a long position in the stock.
- **Flat**: Hold cash only.
- **Sell short**: Take a short position equal to the amount of capital.

The environment accounts for **trading cost**, set to 10 basis points by default, and deducts one basis point per period without trades. The **reward** of the agent consists of the daily return minus trading costs.

The environment tracks the **net asset value (NAV)** of the agent's portfolio (consisting of a single stock) and compares it against the market portfolio, which trades frictionless to raise the bar for the agent.

An episode begins with a starting NAV of 1 unit of cash:

- If the NAV drops to 0, the episode ends with a loss.
- If the NAV hits 2.0, the agent wins.

This setting limits complexity as it focuses on a single stock and abstracts from position sizing to avoid the need for continuous actions or a larger number of discrete actions, as well as more sophisticated bookkeeping. However, it is useful to demonstrate how to customize an environment and permits for extensions.

How to design a custom OpenAI trading environment

To build an agent that learns how to trade, we need to create a market environment that provides price and other information, offers relevant actions, and tracks the portfolio to reward the agent accordingly. For a description of the efforts to build a large-scale, real-world simulation environment, see Byrd, Hybinette, and Balch (2019).

OpenAI Gym allows for the design, registration, and utilization of environments that adhere to its architecture, as described in the

documentation. The file `trading_env.py` contains the following code examples, which illustrate the process unless noted otherwise.

The trading environment consists of three classes that interact to facilitate the agent's activities. The `DataSource` class loads a time series, generates a few features, and provides the latest observation to the agent at each time step. `TradingSimulator` tracks the positions, trades and cost, and the performance. It also implements and records the results of a buy-and-hold benchmark strategy. `TradingEnvironment` itself orchestrates the process. We will briefly describe each in turn; see the script for implementation details.

Designing a `DataSource` class

First, we code up a `DataSource` class to load and preprocess historical stock data to create the information used for state observations and rewards. In this example, we will keep it very simple and provide the agent with historical data on a single stock. Alternatively, you could combine many stocks into a single time series, for example, to train the agent on trading the S&P 500 constituents.

We will load the adjusted price and volume information for one ticker from the Quandl dataset, in this case for AAPL with data from the early 1980s until 2018:

```
class DataSource:
    """Data source for TradingEnvironment
    Loads & preprocesses daily price & volume data
    Provides data for each new episode.
    """
    def __init__(self, trading_days=252, ticker='AAPL'):
        self.ticker = ticker
        self.trading_days = trading_days
    def load_data(self):
        idx = pd.IndexSlice
        with pd.HDFStore('../data/assets.h5') as store:
```

```

        dt = (store['quandl/wiki/prices']
            .loc[idx[:, self.ticker],
                  ['adj_close', 'adj_volume', 'adj_low', 'adj_high']]
        df.columns = ['close', 'volume', 'low', 'high']
        return df

```

The `preprocess_data()` method creates several features and normalizes them. The most recent daily returns play two roles:

- An element of the observations for the current state
- The net of trading costs and, depending on the position size, the reward for the last period

The method takes the following steps, among others (refer to the *Appendix* for details on the technical indicators):

```

def preprocess_data(self):
    """calculate returns and percentiles, then removes missing values"""
    self.data['returns'] = self.data.close.pct_change()
    self.data['ret_2'] = self.data.close.pct_change(2)
    self.data['ret_5'] = self.data.close.pct_change(5)
    self.data['rsi'] = talib.STOCHRSI(self.data.close)[1]
    self.data['atr'] = talib.ATR(self.data.high,
                                 self.data.low, self.data.close)
    self.data = (self.data.replace((np.inf, -np.inf), np.nan)
                  .drop(['high', 'low', 'close'], axis=1)
                  .dropna())
    if self.normalize:
        self.data = pd.DataFrame(scale(self.data),
                                  columns=self.data.columns,
                                  index=self.data.index)

```

The `DataSource` class keeps track of episode progress, provides fresh data to `TradingEnvironment` at each time step, and signals the end of the episodes:

```

def take_step(self):
    """Returns data for current trading day and done signal"""

```

```
obs = self.data.iloc[self.offset + self.step].values
self.step += 1
done = self.step > self.trading_days
return obs, done
```

The TradingSimulator class

The trading simulator computes the agent's reward and tracks the net asset values of the agent and "the market," which executes a buy-and-hold strategy with reinvestment. It also tracks the positions and the market return, computes trading costs, and logs the results.

The most important method of this class is the `take_step` method, which computes the agent's reward based on its current position, the latest stock return, and the trading costs (slightly simplified; see the script for full details):

```
def take_step(self, action, market_return):
    """ Calculates NAVs, trading costs and reward
        based on an action and latest market return
        returns the reward and an activity summary"""
    start_position = self.positions[max(0, self.step - 1)]
    start_nav = self.navs[max(0, self.step - 1)]
    start_market_nav = self.market_navs[max(0, self.step - 1)]
    self.market_returns[self.step] = market_return
    self.actions[self.step] = action
    end_position = action - 1 # short, neutral, long
    n_trades = end_position - start_position
    self.positions[self.step] = end_position
    self.trades[self.step] = n_trades
    time_cost = 0 if n_trades else self.time_cost_bps
    self.costs[self.step] = abs(n_trades) * self.trading_cost_bps + time_cost
    if self.step > 0:
        reward = start_position * market_return - self.costs[self.step]
        self.strategy_returns[self.step] = reward
        self.navs[self.step] = start_nav * (1 + self.strategy_returns[self.step])
        self.market_navs[self.step] = start_market_nav * (1 + self.market_returns[self.step])
```

```
    self.step += 1
    return reward
```

The TradingEnvironment class

The `TradingEnvironment` class subclasses `gym.Env` and drives the environment dynamics. It instantiates the `DataSource` and `TradingSimulator` objects and sets the action and state-space dimensionality, with the latter depending on the ranges of the features defined by `DataSource`:

```
class TradingEnvironment(gym.Env):
    """A simple trading environment for reinforcement learning.
    Provides daily observations for a stock price series
    An episode is defined as a sequence of 252 trading days with random
    Each day is a 'step' that allows the agent to choose one of three
    """
    def __init__(self, trading_days=252, trading_cost_bps=1e-3,
                 time_cost_bps=1e-4, ticker='AAPL'):
        self.data_source = DataSource(trading_days=self.trading_days,
                                      ticker=ticker)
        self.simulator = TradingSimulator(
            steps=self.trading_days,
            trading_cost_bps=self.trading_cost_bps,
            time_cost_bps=self.time_cost_bps)
        self.action_space = spaces.Discrete(3)
        self.observation_space = spaces.Box(self.data_source.min_value,
                                            self.data_source.max_value)
```

The two key methods of `TradingEnvironment` are `.reset()` and `.step()`. The former initializes the `DataSource` and `TradingSimulator` instances, as follows:

```
def reset(self):
    """Resets DataSource and TradingSimulator; returns first observation
    self.data_source.reset()
```

```
    self.simulator.reset()
    return self.data_source.take_step()[0]
```

Each time step relies on `DataSource` and `TradingSimulator` to provide a state observation and reward the most recent action:

```
def step(self, action):
    """Returns state observation, reward, done and info"""
    assert self.action_space.contains(action),
        '{} {} invalid'.format(action, type(action))
    observation, done = self.data_source.take_step()
    reward, info = self.simulator.take_step(action=action,
                                             market_return=observation[0])
    return observation, reward, done, info
```

Registering and parameterizing the custom environment

Before using the custom environment, just as for the Lunar Lander environment, we need to register it with the `gym` package, provide information about the `entry_point` in terms of module and class, and define the maximum number of steps per episode (the following steps occur in the `q_learning_for_trading` notebook):

```
from gym.envs.registration import register
register(
    id='trading-v0',
    entry_point='trading_env:TradingEnvironment',
    max_episode_steps=252)
```

We can instantiate the environment using the desired trading costs and ticker:

```
trading_environment = gym.make('trading-v0')
trading_environment.env.trading_cost_bps = 1e-3
trading_environment.env.time_cost_bps = 1e-4
```

```
trading_environment.env.ticker = 'AAPL'  
trading_environment.seed(42)
```

Deep Q-learning on the stock market

The notebook `q_learning_for_trading` contains the DDQN agent training code; we will only highlight noteworthy differences from the previous example.

Adapting and training the DDQN agent

We will use the same DDQN agent but simplify the NN architecture to two layers of 64 units each and add dropout for regularization

The online network has 5,059 trainable parameters:

Layer (type)	Output Shape	Param #
Dense_1 (Dense)	(None, 64)	704
Dense_2 (Dense)	(None, 64)	4160
dropout (Dropout)	(None, 64)	0
Output (Dense)	(None, 3)	195
Total params:	5,059	
Trainable params:	5,059	

The training loop interacts with the custom environment in a manner very similar to the Lunar Lander case. While the episode is active, the agent takes the action recommended by its current policy and trains the online network using experience replay after memorizing the current transition. The following code highlights the key steps:

```
for episode in range(1, max_episodes + 1):
    this_state = trading_environment.reset()
    for episode_step in range(max_episode_steps):
        action = ddqn.epsilon_greedy_policy(this_state.reshape(-1, 1), state_c)
```

```

        next_state, reward, done, _ = trading_environment.step(action)

        ddqn.memorize_transition(this_state, action,
                                reward, next_state,
                                0.0 if done else 1.0)
    ddqn.experience_replay()
    if done:
        break
    this_state = next_state
trading_environment.close()

```

We let exploration continue for 2,000 1-year trading episodes, corresponding to about 500,000 time steps; we use linear decay of ϵ from 1.0 to 0.1 over 500 periods with exponential decay at a factor of 0.995 thereafter.

Benchmarking DDQN agent performance

To compare the DDQN agent's performance, we not only track the buy-and-hold strategy but also generate the performance of a random agent.

Figure 22.7 shows the rolling averages over the last 100 episodes of three cumulative return values for the 2,000 training periods (left panel), as well as the share of the last 100 episodes when the agent outperformed the buy-and-hold period (right panel). It uses AAPL stock data, for which there are some 9,000 daily price and volume observations:

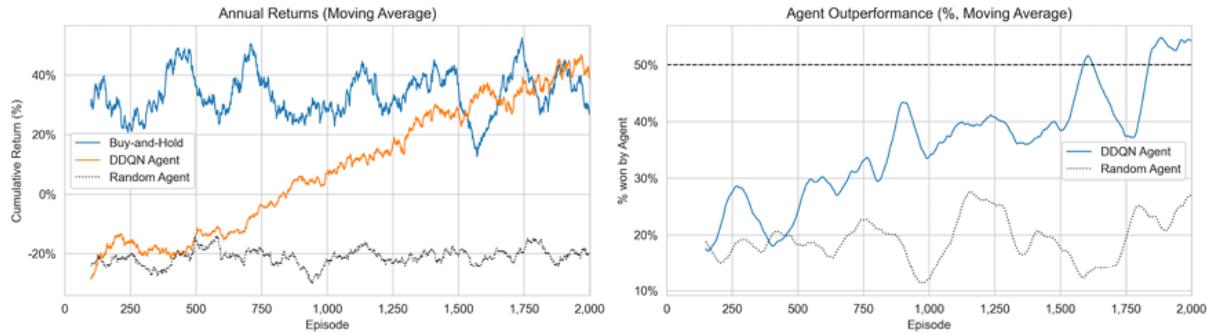


Figure 22.7: Trading agent performance relative to the market

This shows how the agent's performance improves steadily after 500 episodes, from the level of a random agent, and starts to outperform the buy-and-hold strategy toward the end of the experiment more than half of the time.

Lessons learned

This relatively simple agent uses no information beyond the latest market data and the reward signal compared to the machine learning models we covered elsewhere in this book. Nonetheless, it learns to make a profit and achieve performance similar to that of the market (after training on 2,000 years' worth of data, which takes only a fraction of the time on a GPU).

Keep in mind that using a single stock also increases the risk of overfitting to the data—by a lot. You can test your trained agent on new data using the saved model (see the notebook for Lunar Lander).

In summary, we have demonstrated the mechanics of setting up an RL trading environment and experimented with a basic agent that uses a small number of technical indicators. You should try to extend

both the environment and the agent, for example, to choose from several assets, size the positions, and manage risks.

Reinforcement learning is often considered the **most promising approach to algorithmic trading** because it most accurately models the task an investor is facing. However, our dramatically simplified examples illustrate that creating a realistic environment poses a considerable challenge. Moreover, deep reinforcement learning that has achieved impressive breakthroughs in other domains may face greater obstacles given the noisy nature of financial data, which makes it even harder to learn a value function based on delayed rewards.

Nonetheless, the substantial interest in this subject makes it likely that institutional investors are working on larger-scale experiments that may yield tangible results. An interesting complementary approach beyond the scope of this book is **Inverse Reinforcement Learning**, which aims to identify the reward function of an agent (for example, a human trader) given its observed behavior; see Arora and Doshi (2019) for a survey and Roa-Vicens et al. (2019) for an application on trading in the limit-order book context.

Summary

In this chapter, we introduced a different class of machine learning problems that focus on automating decisions by agents that interact with an environment. We covered the key features required to define an RL problem and various solution methods.

We saw how to frame and analyze an RL problem as a finite Markov decision problem, as well as how to compute a solution using value

and policy iteration. We then moved on to more realistic situations, where the transition probabilities and rewards are unknown to the agent, and saw how Q-learning builds on the key recursive relationship defined by the Bellman optimality equation in the MDP case. We saw how to solve RL problems using Python for simple MDPs and more complex environments with Q-learning.

We then expanded our scope to continuous states and applied the Deep Q-learning algorithm to the more complex Lunar Lander environment. Finally, we designed a simple trading environment using the OpenAI Gym platform, and also demonstrated how to train an agent to learn how to make a profit while trading a single stock.

In the next and final chapter, we'll present a few conclusions and key takeaways from our journey through this book and lay out some steps for you to consider as you continue building your skills to use machine learning for trading.

Conclusions and Next Steps

Our goal for this book was to enable you to apply **machine learning (ML)** to a variety of data sources and extract signals that add value to a trading strategy. To this end, we took a more comprehensive view of the investment process, from idea generation to strategy evaluation, and introduced ML as an important element of this process in the form of the **ML4T workflow**.

While demonstrating the use of a broad range of ML algorithms, from the fundamental to the advanced, we saw how ML can add value at multiple steps in the process of designing, testing, and executing a strategy. For the most part, however, we focused on the **core ML value proposition**, which consists of the ability to extract actionable information from much larger amounts of data more systematically than human experts would ever be able to.

This value proposition has really gained currency with the explosion of digital data that made it both more promising and necessary to leverage computing power to extract value from ever more diverse sets of information. However, the application of ML still requires significant human intervention and domain expertise to define objectives, select and curate data, design and optimize a model, and make appropriate use of the results.

Domain-specific aspects of using ML for trading include the nature of financial data and the environment of financial markets. The use

of powerful models with a high capacity to learn patterns requires particular care to avoid overfitting when the signal-to-noise ratio is as low as is often the case with financial data. Furthermore, the competitive nature of trading implies that patterns evolve quickly as signals decay, requiring additional attention to performance monitoring and model maintenance.

In this concluding chapter, we will briefly summarize the key tools, applications, and lessons learned throughout the book to avoid losing sight of the big picture after so much detail. We will then identify areas that we did not cover but would be worthwhile to focus on as you expand on the many ML techniques we introduced and become productive in their daily use.

In sum, in this chapter, we will:

- Review key takeaways and lessons learned
- Point out the next steps to build on the techniques in this book
- Suggest ways to incorporate ML into your investment process

Key takeaways and lessons learned

A central goal of the book was to demonstrate the workflow of extracting signals from data using ML to inform a trading strategy. *Figure 23.1* outlines this ML-for-trading workflow. The key takeaways summarized in this section relate to specific challenges we encounter when building sophisticated predictive models for large datasets in the context of financial markets:

The ML4T Workflow

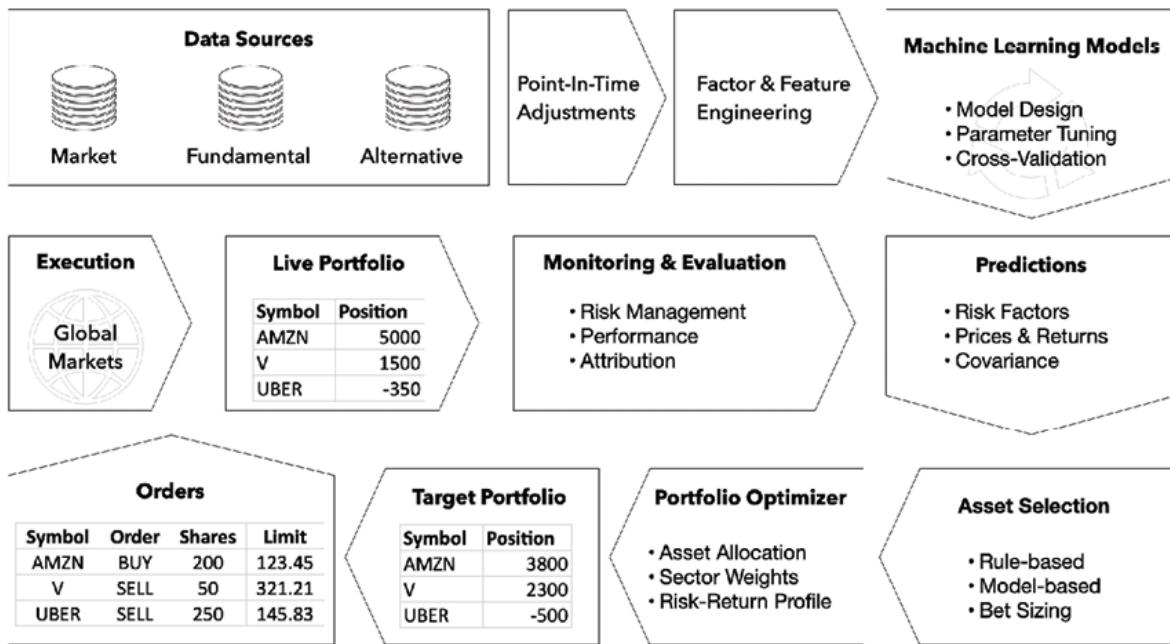


Figure 23.1: Key elements of using ML for trading

Important insights to keep in mind as you proceed to the practice of ML for trading include the following:

- **Data** is the single most important ingredient that requires careful sourcing and handling.
- **Domain expertise** is key to realizing the value contained in data and avoiding some of the pitfalls of using ML.
- ML offers **tools** that you can adapt and combine to create solutions for your use case.
- The choices of **model objectives and performance diagnostics** are key to productive iterations toward an optimal system.
- **Backtest overfitting** is a huge challenge that requires significant attention.

- **Transparency of black-box models** can help build confidence and facilitate the adoption of ML by skeptics.

We will elaborate a bit more on each of these ideas.

Data is the single most important ingredient

The rise of ML in trading and everywhere else largely complements the data explosion that we covered in great detail. We illustrated in *Chapter 2, Market and Fundamental Data – Sources and Techniques*, how to access and work with these data sources, historically the mainstay of quantitative investment. In *Chapter 3, Alternative Data for Finance – Categories and Use Cases*, we laid out a framework with criteria to assess the potential value of alternative datasets.

A key insight is that state-of-the-art ML techniques like deep neural networks are successful because their predictive performance continues to improve with more data. On the flip side, model and data complexity need to match to balance the bias-variance trade-off, which becomes more challenging the higher the noise-to-signal ratio of the data is. Managing data quality and integrating datasets are key steps in realizing the potential value.

The new oil? Quality control for raw and intermediate data

Just like oil, a popular comparison these days, data passes through a **pipeline with several stages** from its raw form to a refined product that can fuel a trading strategy. Careful attention to the quality of the final product is critical to getting the desired mileage out of it.

Sometimes, you get **data in its raw form** and control the numerous transformations required for your purposes. More often, you deal with an **intermediate product** and should get clarity about what exactly the data measures at this point.

Different from oil, there is often **no objective quality standard** as data sources continue to proliferate. Instead, the quality depends on its signal content, which in turn depends on your investment objectives. The cost-effective evaluation of new datasets requires a productive workflow, including appropriate infrastructure that we will address later in this chapter.

Data integration – the whole exceeds the sum of its parts

The value of data for an investment strategy often depends on combining complementary sources of market, fundamental, and alternative data. We saw that the predictive power of ML algorithms, like tree-based ensembles or neural networks, is in part due to their ability to detect nonlinear relationships, in particular **interaction effects among variables**.

The ability to modulate the impact of a variable as a function of other model features thrives on data inputs that capture different aspects of a target outcome. The combination of asset prices with macro fundamentals, social sentiment, credit card payment, and satellite data will likely yield significantly more reliable predictions throughout different economic and market regimes than each source on its own (provided the amount of data is large enough to learn the hidden relationships).

Working with data from multiple sources increases the **challenges of proper labeling**. It is vital to assign accurate timestamps that

accurately reflect historical publication. Otherwise, we introduce lookahead bias by testing an algorithm with data before it actually becomes available. For example, third-party data may have timestamps that require adjustments to reflect the point in time when the information would have been available for a live algorithm.

Domain expertise – telling the signal from the noise

We emphasized that informative data is a necessary condition for successful ML applications. However, domain expertise is equally essential to define the strategic direction, select relevant data, engineer informative features, and design robust models.

In any domain, practitioners have theories about the drivers of key outcomes and relationships among them. Finance is characterized by a **large amount of available quantitative research**, both theoretical and empirical. However, Marcos López de Prado and others (Cochrane 2011) criticize most empirical results: claims of predictive signals found in hundreds of variables are often based on pervasive data mining and are not robust to changes in the experimental setup. In other words, statistical significance often results from large-scale trial-and-error rather than a true systematic relationship, along the lines of "if you torture the data long enough, it will confess."

On the one hand, there exists a robust understanding of how financial markets work. This should inform the selection and use of data as well as the justification of strategies that rely on ML. An important reason is to prioritize ideas that are more likely to be successful and avoid the multiple testing trap that leads to unreliable

results. We outlined key ideas in *Chapter 4, Financial Feature Engineering – How to Research Alpha Factors*, and *Chapter 5, Portfolio Optimization and Performance Evaluation*.

On the other hand, novel ML techniques will likely uncover new hypotheses about drivers of financial outcomes that will inform theory and should then be independently tested.

More than the raw data, feature engineering is often the key to making signals useful for an algorithm. Leveraging decades of research into risk factors that drive returns on theoretical and empirical grounds is a good starting point to prioritize data transformations that are more likely to reflect relevant information.

However, only creative feature engineering will lead to innovative strategies that can compete in the market over time. Even for new alpha factors, a compelling narrative that explains how they work given established ideas on market dynamics and investor behavior will provide more confidence to allocate capital.

The risks of false discoveries and overfitting to historical data make it even more necessary to prioritize strategies prior to testing rather than "letting the data speak." We covered how to deflate the Sharpe ratio in *Chapter 7, Linear Models – From Risk Factors to Return Forecasts*, to account for the number of experiments.

ML is a toolkit for solving problems with data

ML offers algorithmic solutions and techniques that can be applied to many use cases. *Parts 2, 3, and 4* of this book have presented ML as a diverse set of tools that can add value to various steps of the strategy process, including:

- Idea generation and alpha factor research
 - Signal aggregation and portfolio optimization
 - Strategy testing
 - Trade execution
 - Strategy evaluation

Moreover, ML algorithms are designed to be further developed, adapted, and combined to solve new problems in different contexts. For these reasons, it is important to understand key concepts and ideas underlying these algorithms, in addition to being able to apply them to data for productive experimentation and research as outlined in *Chapter 6, The Machine Learning Process*, and summarized in *Figure 23.2*:

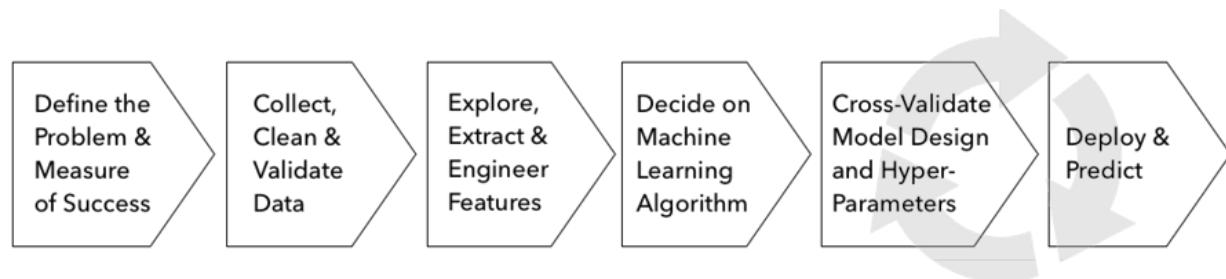


Figure 23.2: The ML workflow

Furthermore, the best results are often achieved by **human-in-the-loop solutions** that combine humans with ML tools. In *Chapter 1, Machine Learning for Trading – From Idea to Execution*, we covered the quantamental investment style where discretionary and algorithmic trading converge. This approach will likely further grow in importance and depends on the flexible and creative application of the fundamental tools that we covered and their extensions to a variety of datasets.

Model diagnostics help speed up optimization

In *Chapter 6, The Machine Learning Process*, we outlined the most important ML-specific concepts. ML algorithms learn relationships between input data and a target by making assumptions about the functional form. If the learning is based on noise rather than signal, predictive performance will suffer.

Of course, we do not know today how to separate signal and noise from the perspective of tomorrow's outcomes. Careful cross-validation that avoids lookahead bias and robust model diagnostics, such as learning curves and the optimization verification test, can help alleviate this fundamental challenge and calibrate the choice or configuration of an algorithm. This task can be made easier by defining focused model objectives and, for complex models, distinguishing between performance shortcomings due to issues with the optimization algorithm and those with the objective itself.

Making do without a free lunch

No system, whether a computer program or a human, can reliably predict outcomes for new examples beyond those it has observed during training. The only way out is to have some additional prior knowledge or make assumptions that go beyond the training examples. We covered a broad range of algorithms from linear models in *Chapter 7, Linear Models – From Risk Factors to Return Forecasts*, to nonlinear ensembles in *Chapter 11, Random Forests – A Long-Short Strategy for Japanese Stocks*, and *Chapter 12, Boosting Your Trading Strategy*, as well as neural networks in various chapters of *Part 4* of this book.

We saw that a linear model makes the strong assumption that the relationship between inputs and outputs has a very simple form, whereas nonlinear models like gradient boosting or neural networks aim to learn more complex functions. While it's probably obvious that a simple model will fail in most circumstances, a complex model is not always better. If the true relationship is linear but the data is noisy, the complex model will learn the noise as part of the complex relationship that it assumes to exist. This is the basic idea behind the **"no free lunch" theorem**, which states that no algorithm is universally superior for all tasks. Good fit in some instances comes at the cost of poor performance elsewhere.

The key tools to tailor the choice of the algorithm to the data are data exploration and experiments based on an understanding of the assumptions the model makes.

Managing the bias-variance trade-off

A key challenge in adapting an algorithm to data is the trade-off between bias and variance, which both increase prediction errors beyond the natural noisiness of the data. A simple model that does not adequately capture the relationships in the data will underfit and exhibit bias, that is, make systematically wrong predictions. A model that is too complex will overfit and learn the noise in addition to any signal so that the result will show a lot of variance for different samples.

The key tool to diagnose this trade-off at any given iteration of the model selection and optimization process is the **learning curve**. It shows how training and validation errors depend on the sample size. This allows us to decide between different options to improve

performance: adjust the complexity of the model or get more data points.

The closer the training error is to human performance or another benchmark, the more likely the model will overfit. A low validation error tells us that we are lucky and found a good model. If the validation error is high, we are not. If it continues to decline with the training size, however, more data may help. If the training error is high, more data is unlikely to help, and we should instead add features or use a more flexible algorithm.

Defining targeted model objectives

One of the first steps in the ML process is the definition of an objective for the algorithm to optimize. Sometimes, the choice is simple, such as in a regression problem. A classification task can be more difficult, for example, when we care about precision and recall. Consolidating conflicting objectives into a single metric like the F1 score helps to focus optimization efforts. We can also include conditions that need to be met rather than optimized for. We also saw that reinforcement learning is all about defining the right reward function to guide the agent's learning process.

The optimization verification test

Andrew Ng emphasizes the distinction between performance shortcomings due to a problem with the learning algorithm or the optimization algorithm. Complex models like neural networks assume nonlinear relationships, and the search process of the optimization algorithm may end up in a local rather than a global optimum.

If a model fails to correctly translate a phrase, for example, the test compares the scores for the correct prediction and the solution discovered by the search algorithm. If the learning algorithm scores the correct solution higher, the search algorithm requires improvements. Otherwise, the learning algorithm is optimizing for the wrong objective.

Beware of backtest overfitting

We covered the risks of false discoveries due to overfitting to historical data repeatedly throughout the book. *Chapter 5, Portfolio Optimization and Performance Evaluation*, on strategy evaluation, lays out the main drivers and potential remedies. The low noise-to-signal ratio and relatively small datasets (compared to web-scale image or text data) make this challenge particularly serious in the trading domain. Awareness is critical since the ease of access to data and tools to apply ML increases the risks significantly.

There are no easy answers because the risks are inevitable. However, we presented methods to adjust backtest metrics to account for repeated trials, such as the deflated Sharpe ratio. When working toward a live trading strategy, staged paper-trading and closely monitored performance during execution in the market need to be part of the implementation process.

How to gain insights from black-box models

Deep neural networks and complex ensembles can raise suspicion when they are considered impenetrable black-box models, particularly in light of the risks of backtest overfitting. We

introduced several methods to gain insights into how these models make predictions in *Chapter 12, Boosting Your Trading Strategy*.

In addition to conventional measures of feature importance, the recent game-theoretic innovation of **SHapley Additive exPlanations (SHAP)** is a significant step toward understanding the mechanics of complex models. SHAP values allow the exact attribution of features and their values to predictions so that it becomes easier to validate the logic of a model in the light of specific theories about market behavior for a given investment target. Besides justification, exact feature importance scores and attribution of predictions allow deeper insights into the drivers of the investment outcome of interest.

On the other hand, there is some controversy over how important transparency around model predictions should be. Geoffrey Hinton, one of the inventors of deep learning, argues that the reasons for human decisions are often obscure. Perhaps machines should be evaluated by their results, just as we do with investment managers.

ML for trading in practice

As you proceed to integrate the numerous tools and techniques into your investment and trading process, there are numerous things you can focus your efforts on. If your goal is to make better decisions, you should select projects that are realistic yet ambitious given your current skill set. This will help you to develop an efficient workflow underpinned by productive tools and gain practical experience.

We will briefly list some of the tools that are useful to expand on the Python ecosystem covered in this book. They include big data

technologies that will eventually be necessary to implement ML-driven trading strategies at scale. We will also list some of the platforms that allow you to implement trading strategies using Python, possibly with access to data sources, and ML algorithms and libraries. Finally, we will point out good practices for adopting ML as an organization.

Data management technologies

The central role of data in the ML4T process requires familiarity with a range of technologies to store, transform, and analyze data at scale, including the use of cloud-based services like Amazon Web Services, Microsoft Azure, and Google Cloud.

Database systems

Data storage implies the use of databases. Historically, these have typically been **relational database management systems (RDBMSes)** that use SQL to store and retrieve data in a well-defined table format. These have included databases from commercial providers like Oracle and Microsoft and open-source implementations like PostgreSQL and MySQL. More recently, non-relational alternatives have emerged that are often collectively labeled NoSQL but are quite diverse, namely:

- **Key-value storage:** Fast read/write access to objects. We covered the HDF5 format in *Chapter 2, Market and Fundamental Data – Sources and Techniques*, which facilitates fast access to a pandas DataFrame.
- **Columnar storage:** Capitalizes on the homogeneity of data in a column to facilitate compression and faster column-based

operations like aggregation. This is used in the popular Amazon Redshift data warehouse solution, Apache Parquet, Cassandra, and Google's Big Table.

- **Document store:** Designed to store data that defies the rigid schema definition required by an RDBMS. This has been popularized by web applications that use JSON or XML format, which we encountered in *Chapter 4, Financial Feature Engineering – How to Research Alpha Factors*. It is used, for example, in MongoDB.
- **Graph database:** Designed to store networks that have nodes and edges and specializes in queries about network metrics and relationships. It is used in Neo4J and Apache Giraph.

There has been some convergence toward the conventions established by the relational database systems. The Python ecosystem facilitates the interaction with many standard data sources and provides the fast HDF5 and Parquet formats, as demonstrated throughout the book.

Big data technologies – from Hadoop to Spark

Data management at scale for hundreds of gigabytes and beyond requires the use of multiple machines that form a cluster to conduct read, write, and compute operations in parallel. In other words, you need a distributed system that operates on multiple machines in an integrated way.

The **Hadoop ecosystem** has emerged as an open-source software framework for distributed storage and processing of big data using the MapReduce programming model developed by Google. The ecosystem has diversified under the roof of the Apache Foundation

and today includes numerous projects that cover different aspects of data management at scale.

Key tools within Hadoop include:

- **Apache Pig:** A data processing language, developed at Yahoo, for implementing large-scale **extract-transform-load (ETL)** pipelines using MapReduce.
- **Apache Hive:** The de facto standard for interactive SQL queries over petabytes of data. It was developed at Facebook.
- **Apache HBASE:** A NoSQL database for real-time read/write access that scales linearly to billions of rows and millions of columns. It can combine data sources using a variety of different schemas.

Apache Spark has become the most popular platform for interactive analytics on a cluster. The MapReduce framework allowed parallel computation but required repeated read/write operations from disk to ensure data redundancy. Spark has dramatically accelerated computation at scale due to the **resilient distributed data (RDD)** structure, which allows highly optimized in-memory computation. This includes iterative computation as required for optimization, for example, gradient descent for numerous ML algorithms. Fortunately, the Spark DataFrame interface has been designed with pandas in mind so that your skills transfer relatively smoothly.

ML tools

We covered many libraries of the Python ecosystem in this book. Python has evolved to become the language of choice for data science and ML. The set of open-source libraries continues to both

diversify and mature, and is built on the robust core of scientific computing libraries NumPy and SciPy.

The popular pandas library has contributed significantly to popularizing the use of Python for data science and has matured with its 1.0 release in January 2020. The scikit-learn interface has become the standard for modern, specialized ML libraries like XGBoost or LightGBM that often interface with the workflow automation tools like GridSearchCV and Pipeline that we have used repeatedly throughout the book.

There are several providers that aim to facilitate the ML workflow:

- **H2O.ai** offers the H2O platform, which integrates cloud computing with ML automation. It allows users to fit thousands of potential models to their data to explore patterns in the data. It has interfaces in Python as well as R and Java.
- **Datarobot** aims to automate the model development process by providing a platform to rapidly build and deploy predictive models in the cloud or on-premises.
- **Dataiku** is a collaborative data science platform designed to help analysts and engineers explore, prototype, build, and deliver their own data products.

There are also several open-source initiatives led by companies that build on and expand the Python ecosystem:

- The quantitative hedge fund **TwoSigma** contributes quantitative analysis tools to the Jupyter Notebook environment under the BeakerX project.
- **Bloomberg** has integrated the Jupyter Notebook into its terminal to facilitate the interactive analysis of its financial data.

Online trading platforms

The main options to develop trading strategies that use ML are online platforms, which often look for and allocate capital to successful trading strategies. Popular solutions include Quantopian, Quantconnect, and QuantRocket. The more recent Alpha Trading Labs focuses on high-frequency trading. In addition, **Interactive Brokers (IB)** offers a Python API that you can use to develop your own trading solution.

Quantopian

We introduced the Quantopian platform and demonstrated the use of its research and trading environment to analyze and test trading strategies against historical data. Quantopian uses Python and offers a lot of educational material.

Quantopian hosts competitions to recruit algorithms for its crowd-sourced hedge fund portfolio. It provides capital to the winning algorithm. Live trading was discontinued in September 2017, but the platform still provides a large range of historical data and attracts an active community of developers and traders. It is a good starting point to discuss ideas and learn from others.

QuantConnect

QuantConnect is another open-source, community-driven algorithmic trading platform that competes with Quantopian. It also provides an IDE to backtest and live trade algorithmic strategies using Python and other languages.

QuantConnect also has a dynamic, global community from all over the world, and provides access to numerous asset classes, including equities, futures, FOREX, and cryptocurrency. It offers live trading integration with various brokers, such as IB, OANDA, and GDAX.

QuantRocket

QuantRocket is a Python-based platform for researching, backtesting, and running automated quantitative trading strategies. It provides data collection tools, multiple data vendors, a research environment, multiple backtest engines, and live and paper trading through IB. It prides itself on support for international equity trading and sets itself apart with its flexibility (but Quantopian is working toward this as well).

QuantRocket supports multiple engines — its own Moonshot, as well as third-party engines as chosen by the user. While QuantRocket doesn't have a traditional IDE, it is integrated well with Jupyter to produce something similar. QuantRocket offers a free version with access to sample data, but access to a wider set of capabilities starts at \$29 per month at the time of writing in early 2020.

Conclusion

We started by highlighting the explosion of digital data and the emergence of ML as a strategic capability for investment and trading strategies. This dynamic reflects global business and technology trends beyond finance and is much more likely to continue than to stall or reverse. Many investment firms are just getting started to

leverage the range of artificial intelligence tools, just as individuals are acquiring the relevant skills and business processes are adapting to these new opportunities for value creation, as outlined in the introductory chapter.

There are also numerous exciting developments for the application of ML to trading on the horizon that are likely to propel the current momentum. They are likely to become relevant in the coming years and include the automation of the ML process, the generation of synthetic training data, and the emergence of quantum computing. The extraordinary vibrancy of the field implies that this alone could fill a book and the journey will continue to remain exciting.

Alpha Factor Library

Throughout this book, we've described how to engineer features from market, fundamental, and alternative data to build **machine learning (ML)** models that yield signals for a trading strategy. The smart design of features, including appropriate preprocessing and denoising, is what typically leads to an effective strategy. This appendix synthesizes some of the lessons learned on feature engineering and provides additional information on this important topic.

Chapter 4, Financial Feature Engineering – How to Research Alpha Factors, summarized the long-standing efforts of academics and practitioners to identify information or variables that help reliably predict asset returns. This research led from the single-factor capital asset pricing model to a "zoo of new factors" (Cochrane, 2011). This *factor zoo* contains hundreds of firm characteristics and security price metrics presented as statistically significant predictors of equity returns in the anomalies literature since 1970 (see a summary in Green, Hand, and Zhang, 2017).

Chapter 4, Financial Feature Engineering – How to Research Alpha Factors, categorized factors by the underlying risk they represent and for which an investor would earn a reward above and beyond the market return. These categories include value versus growth, quality, and sentiment, as well as volatility, momentum, and liquidity. Throughout this book, we used numerous metrics to capture these risk factors. This appendix expands on those examples and collects popular indicators so you can use it as a reference or inspiration for your own strategy development. It also shows you

how to compute them and includes some steps to evaluate these indicators.

To this end, we'll focus on the broad range of indicators implemented by TA-Lib (see *Chapter 4, Financial Feature Engineering – How to Research Alpha Factors*) and the *101 Formulaic Alphas* paper (Kakushadze 2016), which presents real-life quantitative trading factors used in production with an average holding period of 0.6-6.4 days. To facilitate replication, we'll limit this review to indicators that rely on readily available market data. This restriction notwithstanding, the vast and rapidly evolving scope of potentially useful data sources and features implies that this overview is far from comprehensive.

Throughout this chapter, we will use P_t for the closing price and V_t for the trading volume of an asset at time t . Where necessary, superscripts like P_t^{high} or P_t^H differentiate between open, high, low, or close prices. r_t denotes the simple return for the period return at time t .

$P_{t-d,t} = \{p_{t-d}, p_{t-d+1}, \dots, p_t\}$ and $R_{t-d,t} = \{r_{t-d}, r_{t-d+1}, \dots, r_t\}$ refer to a time series of prices and returns, respectively, from $t-d$ to t .

Common alpha factors implemented in TA-Lib

The TA-Lib library is widely used to perform technical analysis of financial market data by trading software developers. It includes over 150 popular indicators from multiple categories that range from

overlap studies, including moving averages and Bollinger Bands, to statistic functions such as linear regression. The following table summarizes the main categories:

Function Group	# Indicators
Overlap Studies	17
Momentum Indicators	30
Volume Indicators	3
Volatility Indicators	3
Price Transform	4
Cycle Indicators	5
Math Operators	11
Math Transform	15
Statistic Functions	9

There are also over 60 functions that aim to recognize candlestick patterns popular with traders that rely on the visual inspection of charts. Given the mixed evidence on their predictive ability (Horton 2009; Marshall, Young, and Rose 2006), and the goal of learning such patterns from data using the ML algorithms covered in this book, we will focus on the categories listed in the preceding table. Specifically,

we will focus on moving averages, overlap studies, momentum, volume and liquidity, volatility, and fundamental risk factors in this section.

See the notebook `common_alpha_factors` for the code examples in this section and additional implementation details regarding TA-Lib indicators. We'll demonstrate how to compute selected indicators for an individual stock, as well as a sample of the 500 most-traded US stocks over the 2007-2016 period (see the notebook `sample_selection` for the preparation of this larger dataset).

A key building block – moving averages

Numerous indicators allow for calculation using different types of **moving averages (MAs)**. They make different tradeoffs between smoothing a series and reacting to new developments. You can use them as building blocks for your own indicators or modify the behavior of existing indicators by altering the type of MA used in its construction, as we'll demonstrate in the next section. The following table lists the available types of MAs, the TA-Lib function to compute them, and the code you can pass to other indicators to select the given type:

Moving Average	Function	Code
Simple	SMA	0
Exponential	EMA	1
Weighted	WMA	2

Double Exponential	DEMA	3
Triple Exponential	TEMA	4
Triangular	TRIMA	5
Kaufman Adaptive	KAMA	6
MESA Adaptive	MAMA	7

In the remainder of this section, we'll briefly outline their definitions and visualize their different behavior.

Simple moving average

For price series P_t with a window of length N , the **simple moving average (SMA)** at time t weighs each data point within the window equally:

$$\text{SMA}(N)_t = \frac{P_{t-N+1} + P_{t-N+2} + P_{t-N+3} + P_t}{N} = \frac{1}{N} \sum_{i=1}^N P_{t-N+i}$$

Exponential moving average

For price series P_t with a window of length N , the **exponential moving average (EMA)** at time t , EMA_t , is recursively defined as the weighted average of the current price and the most recent previous EMA_{t-1} , where the weights α and $1 - \alpha$ are defined as follows:

$$\text{EMA}(N)_t = \alpha P_t + (1 - \alpha) \text{EMA}(N)_{t-1}$$

$$\alpha = \frac{2}{N + 1}$$

Weighted moving average

For price series P_t with a window of length N , the **weighted moving average (WMA)** at time t is computed such that the weight of each data point corresponds to its index within the window:

$$\text{WMA}(N)_t = \frac{P_{t-N+1} + 2P_{t-N+2} + 3P_{t-N+3} + NP_t}{N(N + 1)/2}$$

Double exponential moving average

The **double exponential moving average (DEMA)** for a price series P_t at time t , DEMA_t , is based on the EMA designed to react faster to changes in price. It is computed as the difference between twice the current EMA and the EMA applied to the current EMA, labeled $\text{EMA}_2(N)_t$:

$$\text{DEMA}(N)_t = 2 \times \text{EMA}(N)_t - \text{EMA}_2(N)_t$$

Since the calculation uses EMA_2 , DEMA needs $2 \times N - 1$ samples to start producing values.

Triple exponential moving average

The **triple exponential moving average (TEMA)** for a price series P_t at time t , TEMA_t , is also based on the EMA, yet designed to react even faster to changes in price and indicate short-term price direction. It is computed as the difference between three times the difference between the current EMA and the EMA applied to the

current EMA, EMA_2 , with the addition of the EMA applied to the EMA_2 , labeled EMA_3 :

$$\text{TEMA}(N)_t = 3 \times [\text{EMA}(N)_t - \text{EMA}_2(N)_t] + \text{EMA}_3(N)_t$$

Since the calculation uses EMA_3 , DEMA needs $3 \times N - 2$ samples to start producing values.

Triangular moving average

The **triangular moving average (TRIMA)** with window length N for a price series P_t at time t , $\text{TRIMA}(N)_t$, is a weighted average of the last N $\text{SMA}(N)_t$ values. In other words, it applies the SMA to a time series of SMA values:

$$\text{TRIMA}(N)_t = \frac{1}{N} \sum_{i=1}^N \text{SMA}(N)_{t-N+i}$$

Kaufman adaptive moving average

The computation of the **Kaufman adaptive moving average (KAMA)** aims to take into account changes in market volatility. See the notebook for links to resources that explain the details of this slightly more involved computation.

MESA adaptive moving average

The **MESA adaptive moving average (MAMA)** is an exponential moving average that adapts to price movement based on the rate change of phase, as measured by the **Hilbert Transform discriminator** (see TA-Lib documentation). In addition to the price series, MAMA accepts two additional parameters, *fastlimit* and

slowlimit, that control the maximum and minimum alpha values that should be applied to the EMA when calculating MAMA.

Visual comparison of moving averages

Figure A.1 illustrates how the behavior of the different MAs differs in terms of smoothing the time series and adapting to recent changes. All the time series are calculated for a 21-day moving window (see the notebook for details and color images):



Figure A.1: Comparison of MAs for AAPL closing price

Overlap studies – price and volatility trends

TA-Lib includes several indicators aimed at capturing recent trends, as listed in the following table:

Function	Name
BBANDS	Bollinger Bands
HT TRENDLINE	Hilbert Transform – Instantaneous Trendline
MAVP	Moving average with variable period
MA	Moving average
SAR	Parabolic SAR
SAREXT	Parabolic SAR – Extended

The `MA` and `MAVP` functions are wrappers for the various MAs described in the previous section. We will highlight a few examples in this section; see the notebook for additional information and visualizations.

Bollinger Bands

Bollinger Bands combine an MA with an upper band and a lower band representing the moving standard deviation. We can obtain the three time series by providing an input price series, the length of the moving window, the multiplier for the upper and lower bands, and the type of MA, as follows:

```

s = talib.BBANDS(df.close,      # No. of periods (2 to 100000)
                  timeperiod=20,
                  nbdevup=2,      # Deviation multiplier for Lower band
                  nbdevdn=2,      # Deviation multiplier for upper band
                  matype=1)       # default: SMA

```

For a sample of AAPL closing prices for 2012, we can plot the result like so:

```

bb_bands = ['upper', 'middle', 'lower']
df = price_sample.loc['2012', ['close']]
df = df.assign(**dict(zip(bb_bands, s)))
ax = df.loc[:, ['close'] + bb_bands].plot(figsize=(16, 5), lw=1);

```

The preceding code results in the following plot:

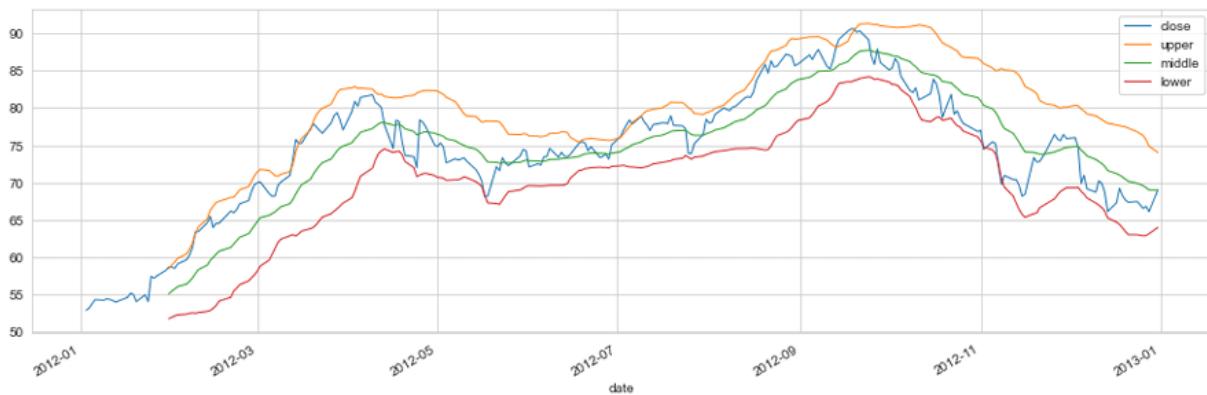


Figure A.2: Bollinger Bands for AAPL close price in 2012

John Bollinger, who invented the concept, also defined over 20 trading rules based on the relationships between the three lines and the current price (see *Chapter 4, Financial Feature Engineering – How to Research Alpha Factors*). For example, a smaller distance between the outer bands implies reduced recent price volatility, which, in turn, is interpreted as greater volatility and price change going forward.

We can standardize the security-specific values of the Bollinger Bands by forming ratios between the upper and lower bands, as well as between each of them and the close price, as follows:

```
fig, ax = plt.subplots(figsize=(16,5))
df.upper.div(df.close).plot(ax=ax, label='bb_up')
df.lower.div(df.close).plot(ax=ax, label='bb_low')
df.upper.div(df.lower).plot(ax=ax, label='bb_squeeze')
plt.legend()
fig.tight_layout();
```

The following plot displays the resulting normalized time series:



Figure A.3: Normalized Bollinger Band indicators

The following function can be used with the pandas `.groupby()` and `.apply()` methods to compute the indicators for a larger sample of 500 stocks, as shown here:

```
def compute_bb_indicators(close, timeperiod=20, matype=0):
    high, mid, low = talib.BBANDS(close,
                                   timeperiod=timeperiod,
                                   matype=matype)

    bb_up = high / close -1
    bb_low = low / close -1
    squeeze = (high - low) / close
    return pd.DataFrame({'BB_UP': bb_up,
                         'BB_LOW': bb_low,
                         'BB_SQUEEZE': squeeze},
```

```

        index=close.index)
data = (data.join(data
                  .groupby(level='ticker')
                  .close
                  .apply(compute_bb_indicators)))

```

Figure A.4 plots the distribution of values for each indicator across the 500 stocks (clipped at the 1st and 99th percentiles, hence the spikes in the plots):

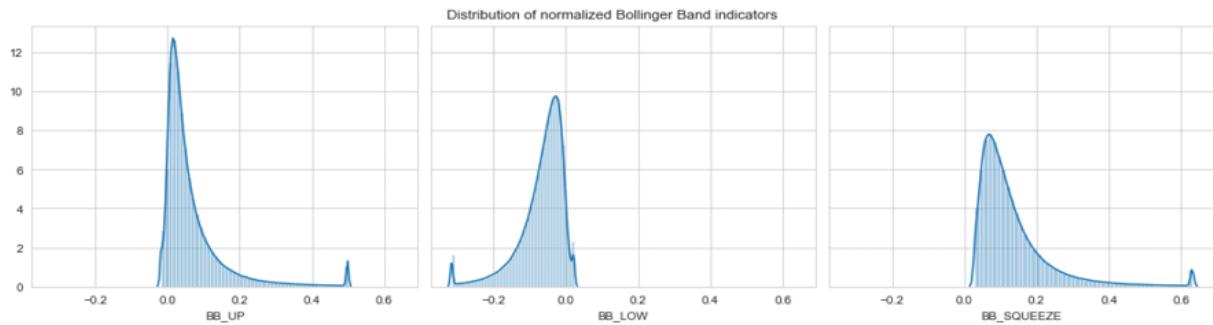


Figure A.4: Distribution of normalized Bollinger Band indicators

Parabolic SAR

The **parabolic SAR** aims to identify trend reversals. It is a trend-following (lagging) indicator that can be used to set a trailing stop loss or determine entry or exit points. It is usually represented in a price chart as a set of dots near the price bars. Generally, when these dots are above the price, it signals a downward trend; it signals an upward trend when the dots are below the price. The change in the direction of the dots can be interpreted as a trade signal. However, the indicator is less reliable in a flat or range-bound market. It is computed as follows:

$$\text{SAR}_t = \text{SAR}_{t-1} + \alpha(\text{EP} - \text{SAR}_{t-1})$$

The **extreme point (EP)** is a record that's kept during each trend that represents the highest value reached by the price during the current uptrend—or lowest value during a downtrend. During each period, if a new maximum (or minimum) is observed, the EP is updated with that value.

The α value represents the acceleration factor and is typically set initially to a value of 0.02. This factor increases by α each time a new EP is recorded. The rate will then quicken to a point where the SAR converges toward the price. To prevent it from getting too large, a maximum value for the acceleration factor is normally set to 0.20.

We can compute and plot it for our sample close price series as follows:

```
df = price_sample.loc['2012', ['close', 'high', 'low']]
df['SAR'] = talib.SAR(df.high, df.low,
                      acceleration=0.02, # common value
                      maximum=0.2)
df[['close', 'SAR']].plot(figsize=(16, 4), style=['-', '--']);
```

The preceding code produces the following plot:



Figure A.5: Parabolic SAR for AAPL stock price

Momentum indicators

Chapter 4, Financial Feature Engineering – How to Research Alpha Factors, introduced **momentum** as one of the best-performing risk factors historically and listed several indicators designed to identify the corresponding price trends. These indicators include the **relative strength index (RSI)**, as well as **price momentum** and **price acceleration**:

Factor	Description	Calculation
Relative strength index (RSI)	RSI compares the magnitude of recent price changes across stocks to identify stocks as overbought or oversold. A high RSI (usually above 70) indicates overbought and a low RSI (typically below 30) indicates oversold. It first computes the average price change for a given number (often 14) of prior trading days with rising (Δ_p^{up}) and falling prices (Δ_p^{down}), respectively.	$RSI = 100 - \frac{100}{1 + \frac{\Delta_p^{up}}{\Delta_p^{down}}}$
Price momentum	This factor computes the total return for a given number of prior trading days d . In the academic literature, it is common to use the last 12 months except for the most recent month due to a short-term reversal effect frequently observed. However, shorter periods have also been widely used.	$Mom^d = \frac{P_t}{P_{t-1}} - 1$
Price acceleration	Price acceleration calculates the gradient of the price trend using the linear regression coefficient β of a time trend on daily prices for a longer and a shorter period, for example, 1 year and 3 months of trading days, and compares the change in the slope as a measure of price acceleration.	$\frac{\beta_{63}}{\beta_{252}}$

TA-Lib implements 30 momentum indicators; the most important ones are listed in the following table. We will introduce a few selected examples; please see the notebook `common_alpha_factors` for additional information:

Function	Name
PLUS_DM/MINUS_DM	Plus/Minus Directional Movement
PLUS_DI/MINUS_DI	Plus/Minus Directional Indicator
DX	Directional Movement Index
ADX	Average Directional Movement Index
ADXR	Average Directional Movement Index Rating
APO/PPO	Absolute/Percentage Price Oscillator
AROON/AROONOSC	Aroon/Aroon Oscillator
BOP	Balance of Power
CCI	Commodity Channel Index
CMO	Chande Momentum Oscillator
MACD	Moving Average Convergence/Divergence
MFI	Money Flow Index

MOM	Momentum
RSI	Relative Strength Index
STOCH	Stochastic
ULTOSC	Ultimate Oscillator
WILLR	Williams' %R

Several of these indicators are closely related and build on each other, as the following example demonstrates.

Average directional movement indicators

The **average directional movement index (ADX)** combines two other indicators, namely the positive and negative directional indicators (`PLUS_DI` and `MINUS_DI`), which, in turn, build on the positive and negative directional movement (`PLUS_DM` and `MINUS_DM`). See the notebook for additional details.

Plus/minus directional movement

For a price series P_t with daily highs P_t^H and daily lows P_t^L , the directional movement tracks the absolute size of price moves over a time period T , as follows:

$$\text{Up}_t = P_t^H - P_{t-T}^H$$

$$\text{Down}_t = P_{t-T}^L - P_t^L$$

$$\text{PLUS_DM}_t = \begin{cases} \text{Up}_t & \text{if } \text{Up}_t > \text{Down}_t \text{ and } \text{Up}_t > 0 \\ 0 & \text{otherwise} \end{cases}$$

$$\text{MINUS_DM}_t = \begin{cases} \text{Down}_t & \text{if } \text{Down}_t > \text{Up}_t \text{ and } \text{Down}_t < 0 \\ 0 & \text{otherwise} \end{cases}$$

We can compute and plot this indicator for a 2-year price series of the AAPL stock in 2012-13:

```
df = price_sample.loc['2012': '2013', ['high', 'low', 'close']]
df['PLUS_DM'] = talib.PLUS_DM(df.high, df.low, timeperiod=10)
df['MINUS_DM'] = talib.MINUS_DM(df.high, df.low, timeperiod=10)
```

The following plot visualizes the resulting time series:

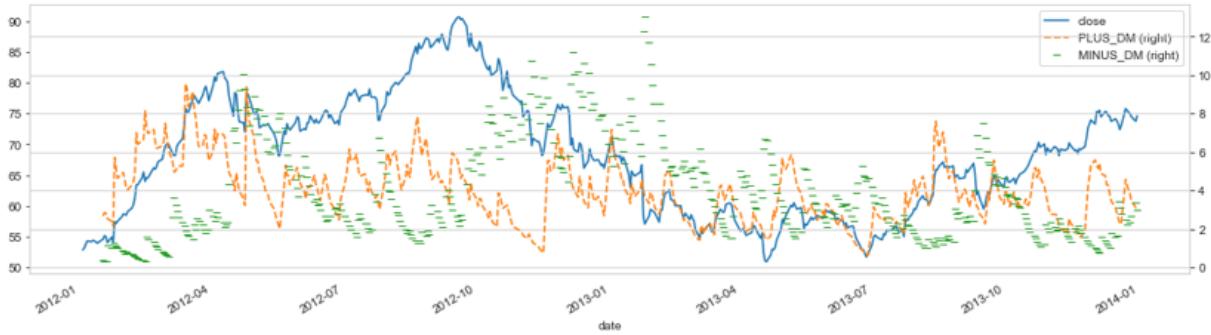


Figure A.6: PLUS_DM/MINUS_DM for AAPL stock price

Plus/minus directional index

`PLUS_DI` and `MINUS_DI` are the simple MAs of `PLUS_DM` and `MINUS_DM`, respectively, each divided by the **average true range (ATR)**. See the *Volatility indicators* section later in this chapter for more details.

The simple MA is calculated over the given number of periods. The ATR is a smoothed average of the true ranges.

Average directional index

Finally, the **average directional index (ADX)** is the (simple) MA of the absolute value of the difference between `PLUS_DI` and `MINUS_DI`, divided by their sum:

$$\text{ADX} = 100 \times \text{SMA}(N)_t \left| \frac{\text{PLUS}_{\text{DI}}_t - \text{MINUS}_{\text{DI}}_t}{\text{PLUS}_{\text{DI}}_t + \text{MINUS}_{\text{DI}}_t} \right|$$

Its values oscillate in the 0-100 range and are often interpreted as follows:

ADX Value	Trend Strength
0-25	Absent or weak trend
25-50	Strong trend
50-75	Very strong trend
75-100	Extremely strong trend

We compute the ADX time series for our AAPL sample series similar to the previous examples, as follows:

```
df[ 'ADX' ] = talib.ADX(df.high,
                         df.low,
                         df.close,
                         timeperiod=14)
```

The following plot visualizes the result over the 2007-2016 period:

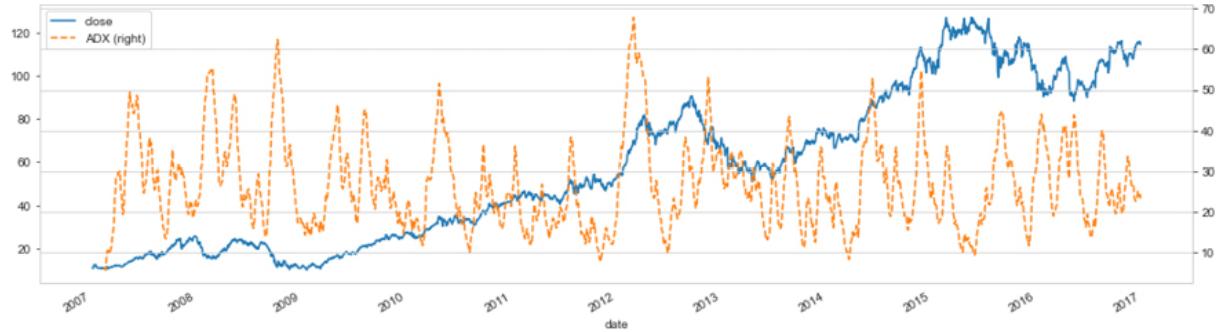


Figure A.7: ADX for the AAPL stock price series

Aroon Oscillator

The Aroon indicator measures the time between highs and the time between lows over a time period. It computes an `AROON_UP` and an `AROON_DOWN` indicator, as follows:

$$\text{AROON_UP} = \frac{T - \text{Periods since T period High}}{T} \times 100$$

$$\text{AROON_DOWN} = \frac{T - \text{Periods since T period Low}}{T} \times 100$$

The Aroon Oscillator is simply the difference between the `AROON_UP` and `AROON_DOWN` indicators and moves within the range from -100 to 100, as shown here for the AAPL price series:



Figure A.8: Aroon Oscillator for the AAPL stock price series

Balance of power

The **balance of power (BOP)** intends to measure the strength of buyers relative to sellers in the market by assessing the influence of each side on the price. It is computed as the difference between the close and the open price, divided by the difference between the high and the low price:

$$BOP_t = \frac{P_t^{\text{Close}} - P_t^{\text{Open}}}{P_t^{\text{High}} - P_t^{\text{Low}}}$$

Commodity channel index

The **commodity channel index (CCI)** measures the difference between the current *typical* price, computed as the average of current low, high, and close price and the historical average price. A positive (negative) CCI indicates that the price is above (below) the historic average. It is computed as follows:

$$\bar{P}_t = \frac{P_t^H + P_t^L + P_t^C}{3}$$

$$CCI_t = \frac{\bar{P}_t - \text{SMA}(N)_t}{0.15 \sum_{t=i}^T |\bar{P}_t - \text{SMA}(N)_t|/T}$$

Moving average convergence divergence

Moving average convergence divergence (MACD) is a very popular trend-following (lagging) momentum indicator that shows the relationship between two MAs of a security's price. It is calculated by subtracting the 26-period EMA from the 12-period EMA.

The TA-Lib implementation returns the MACD value and its signal line, which is the 9-day EMA of the MACD. In addition, the MACD-Histogram measures the distance between the indicator and its signal line. The following charts show the results:

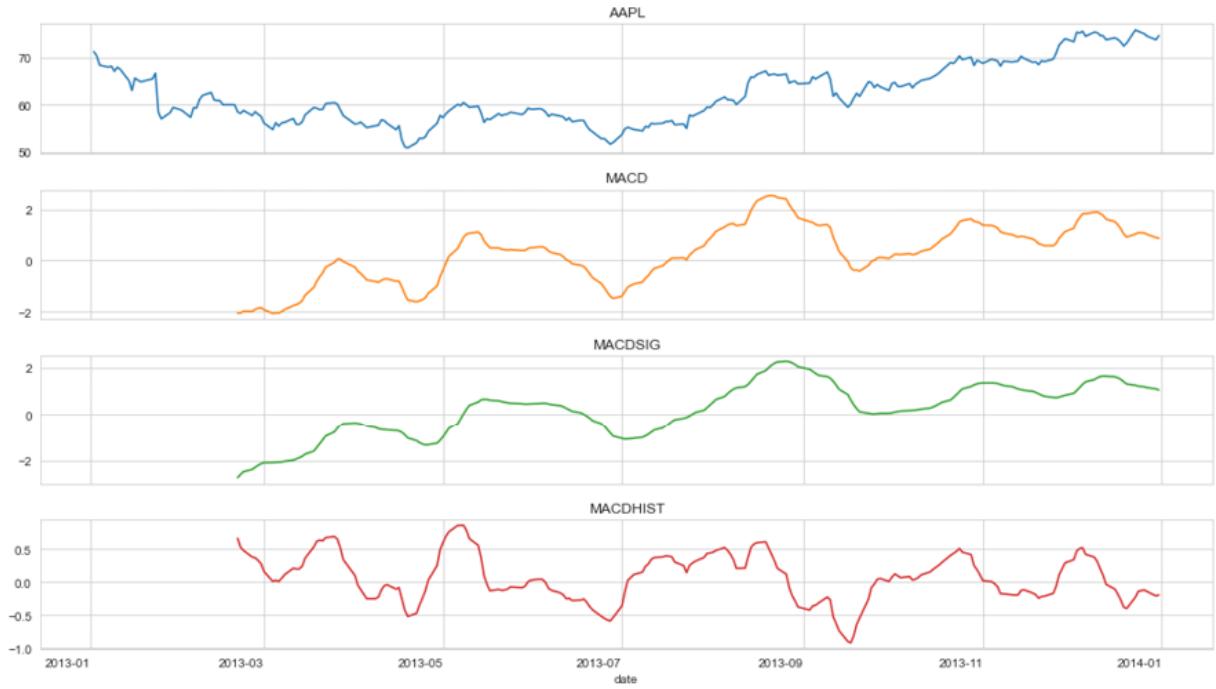


Figure A.9: The three MACD series for the AAPL stock price series

Stochastic relative strength index

The **stochastic relative strength index (StochRSI)** is based on the RSI described at the beginning of this section and intends to identify crossovers, as well as overbought and oversold conditions. It compares the distance of the current RSI to the lowest RSI over a given time period T to the maximum range of values the RSI has assumed for this period. It is computed as follows:

$$\text{STOCHRSI}_t = \frac{\text{RSI}_t - \text{RSI}_t^L(T)}{\text{RSI}_t^H(T) - \text{RSI}_t^L(T)}$$

The TA-Lib implementation offers more flexibility than the original unsmoothed stochastic RSI version by Chande and Kroll (1993). To calculate the original indicator, keep `timeperiod` and `fastk_period` equal.

The return value `fastk` is the unsmoothed RSI. `fastd_period` is used to compute a smoothed StochRSI, which is returned as `fastd`. If you do not care about StochRSI smoothing, just set `fastd_period` to 1 and ignore the `fastd` output:

```
fastk, fastd = talib.STOCHRSI(df.close,
                               timeperiod=14,
                               fastk_period=14,
                               fastd_period=3,
                               fastd_matype=0)
df['fastk'] = fastk
df['fastd'] = fastd
```

Figure A.10 plots the closing price and both the smoothed and unsmoothed stochastic RSI:

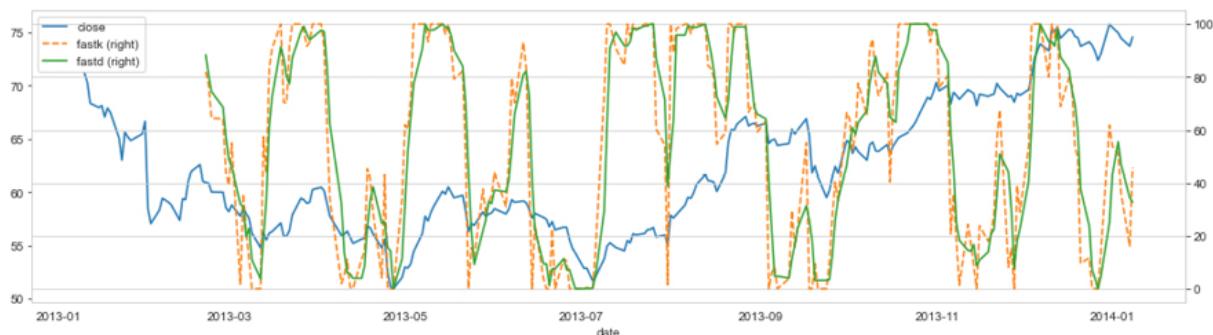


Figure A.10: Smoothed and unsmoothed StochRSI series for the AAPL stock price

Stochastic oscillator

A stochastic oscillator is a momentum indicator that compares a particular closing price of a security to a range of its prices over a

certain period of time. Stochastic oscillators are based on the idea that closing prices should confirm the trend. For Stochastic (STOCH), there are four different lines: K^{Fast} , D^{Fast} , K^{Slow} , and D^{Slow} . D is the signal line usually drawn over its corresponding K function:

$$K^{\text{Fast}}(T_K) = \frac{P_t - P_{T_K}^L}{P_{T_K}^H - P_{T_K}^L} * 100$$

$$D^{\text{Fast}}(T_{\text{FastD}}) = \text{MA}(T_{\text{FastD}})[K^{\text{Fast}}]$$

$$K^{\text{Slow}}(T_{\text{SlowK}}) = \text{MA}(T_{\text{SlowK}})[K^{\text{Fast}}]$$

$$D^{\text{Slow}}(T_{\text{SlowD}}) = \text{MA}(T_{\text{SlowD}})[K^{\text{Slow}}]$$

$P_{T_K}^L$, $P_{T_K}^H$, and $P_{T_K}^L$ are the extreme values of the last T_K period. K^{Slow} and D^{Fast} are equivalent when using the same period. We obtain the series shown in *Figure A.11*, as follows:

```
slowk, slowd = talib.STOCH(df.high,
                           df.low,
                           df.close,
                           fastk_period=14,
                           slowk_period=3,
                           slowk_matype=0,
                           slowd_period=3,
                           slowd_matype=0)
df['STOCH'] = slowd / slowk
```



Figure A.11: STOCH series for the AAPL stock price

Ultimate oscillator

The **ultimate oscillator (ULTOSC)** measures the average difference between the current close and the previous lowest price over three timeframes—with the default values 7, 14, and 28—to avoid overreacting to short-term price changes and incorporate short-, medium-, and long-term market trends.

It first computes the buying pressure, BP_t , then sums it over the three periods T_1 , T_2 , and T_3 , normalized by the true range (TR_t):

$$BP_t = P_t^{\text{Close}} - \min(P_{t-1}^{\text{Close}}, P_t^{\text{Low}})$$

$$TR_t = \max(P_{t-1}^{\text{Close}}, P_t^{\text{High}}) - \min(P_{t-1}^{\text{Close}}, P_t^{\text{Low}})$$

ULTOSC is then computed as a weighted average over the three periods, as follows:

$$\text{Avg}_t(T) = \frac{\sum_{i=0}^{T-1} BP_{t-i}}{\sum_{i=0}^{T-1} TR_{t-i}}$$

$$\text{ULTOSC}_t = 100 * \frac{4\text{Avg}_t(7) + 2\text{Avg}_t(14) + \text{Avg}_t(28)}{4 + 2 + 1}$$

The following plot shows the result of this:

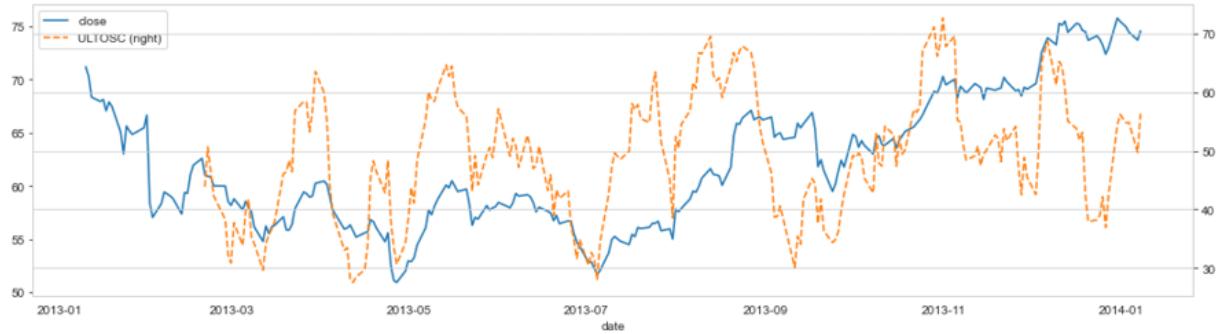


Figure A.12: ULTOSC series for the AAPL stock price

Williams %R

Williams %R, also known as the **Williams Percent Range**, is a momentum indicator that moves between 0 and -100 and measures overbought and oversold levels to identify entry and exit points. It is similar to the stochastic oscillator and compares the current closing price P_t^{Close} to the range of highest (P_T^{High}) and lowest (P_T^{Low}) prices over the last T periods (typically 14). The indicators are computed as follows, and the result is shown in the following chart:

$$\text{WILLR}_t = \frac{P_T^{\text{High}} - P_t^{\text{Close}}}{P_T^{\text{High}} - P_T^{\text{Low}}}$$

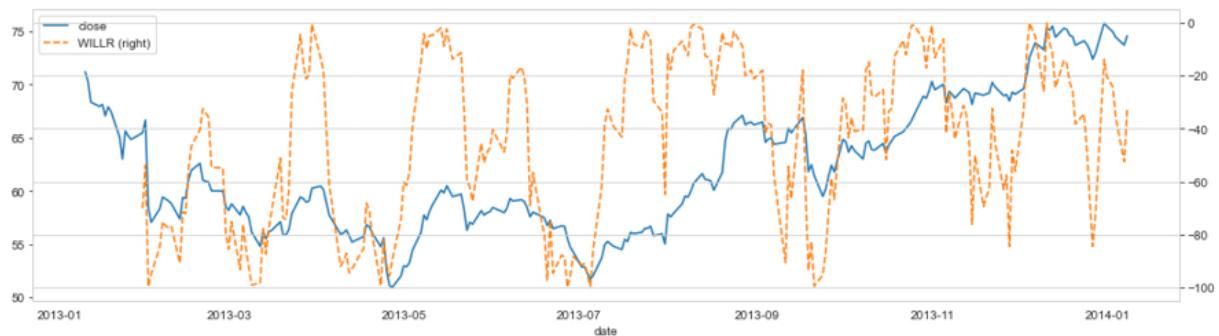


Figure A.13: WILLR series for the AAPL stock price

Volume and liquidity indicators

Risk factors that focus on volume and liquidity incorporate metrics like turnover, dollar volume, or market capitalization. TA-Lib implements three indicators, the first two of which are closely related:

Function	Name
AD	Chaikin A/D Line
ADOSC	Chaikin A/D Oscillator
OBV	On Balance Volume

Also see *Chapter 20, Autoencoders for Conditional Risk Factors and Asset Pricing*, where we use the Amihud Illiquidity indicator to measure a rolling average ratio between absolute returns and the dollar volume.

Chaikin accumulation/distribution line and oscillator

The **Chaikin advance/decline (AD)** or **accumulation/distribution (AD)** line is a volume-based indicator designed to measure the cumulative flow of money into and out of an asset. The indicator assumes that the degree of buying or selling pressure can be determined by the location of the close, relative to the high and low for the period. There is buying (selling) pressure when a stock closes in the upper (lower) half of a period's range. The intention is to signal a change in direction when the indicator diverges from the security price.

The A/D line is a running total of each period's **money flow volume (MFV)**. It is calculated as follows:

1. Compute the **money flow index (MFI)** as the relationship of the close to the high-low range
2. Multiply the MFI by the period's volume V_t to come up with the MFV
3. Obtain the AD line as the running total of the MFV:

$$MFI_t = \frac{P_t^{\text{Close}} - P_t^{\text{Low}}}{P_t^{\text{High}} - P_t^{\text{Low}}}$$

$$MFV_t = MFI_t \times V_t$$

$$AD_t = AD_{t-1} + MFV_t$$

The **Chaikin A/D oscillator (ADOSC)** is the MACD indicator that's applied to the Chaikin AD line. The Chaikin oscillator intends to predict changes in the AD line.

It is computed as the difference between the 3-day EMA and the 10-day EMA of the AD line. The following chart shows the ADOSC series:

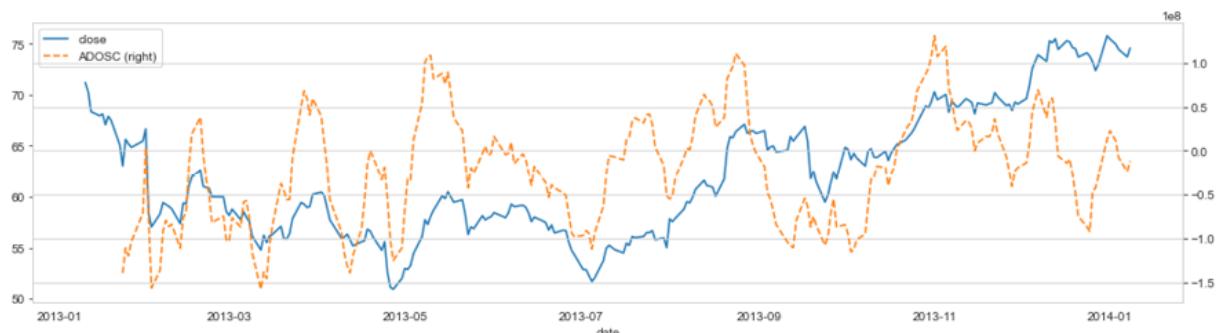


Figure A.14: ADOSC series for the AAPL stock price

On-balance volume

The **on-balance volume (OBV)** indicator is a cumulative momentum indicator that relates volume to price change. It assumes that OBV changes precede price changes because smart money can be seen flowing into the security by a rising OBV. When the public then follows, both the security and OBV will rise.

The current OBV_t is computed by adding (subtracting) the current volume to (from) the last OBV_{t-1} if the security closes higher (lower) than the previous close:

$$OBV_t = \begin{cases} OBV_{t-1} + V_t & \text{if } P_t > P_{t-1} \\ OBV_{t-1} - V_t & \text{if } P_t < P_{t-1} \\ OBV_{t-1} & \text{otherwise} \end{cases}$$

Volatility indicators

Volatility indicators include stock-specific measures like the rolling (normalized) standard deviation of asset prices and returns. It also includes broader market measures like the Chicago Board Options Exchange's **CBOE volatility index (VIX)**, which is based on the implied volatility of S&P 500 options.

TA-Lib implements both normalized and averaged versions of the true range indicator.

Average true range

The **average true range (ATR)** indicator shows the volatility of the market. It was introduced by Wilder (1978) and has been used as a component of numerous other indicators since. It aims to anticipate

changes in trend such that the higher its value, the higher the probability of a trend change; the lower the indicator's value, the weaker the current trend.

ATR is computed as the simple moving average for a period T of the **true range (TRANGE)**, which measures volatility as the absolute value of the largest recent trading range:

$$\text{TRANGE}_t = \max \left[P_t^{\text{High}} - P_t^{\text{low}}, \left| P_t^{\text{High}} - P_{t-1}^{\text{Close}} \right|, \left| P_t^{\text{low}} - P_{t-1}^{\text{Close}} \right| \right]$$

The resulting series is shown in the following plot:



Figure A.15: ATR series for the AAPL stock price

Normalized average true range

TA-Lib also offers a normalized ATR that permits comparisons across assets. The **normalized average true range (NATR)** is computed as follows:

$$\text{NATR}_t = \frac{\text{ATR}_t(T)}{P_t^{\text{Close}}} * 100$$

Normalization makes the ATR more relevant for long-term analysis where the price changes substantially and for cross-market or cross-security comparisons.

Fundamental risk factors

Commonly used measures of risk include the exposure of asset returns to the returns of portfolios designed to represent fundamental factors. We introduced the five-factor model by Fama and French (2015) and showed how to estimate factor loadings and risk factor premia using two-state Fama-Macbeth regressions in *Chapter 7, Linear Models – From Risk Factors to Return Forecasts*.

To estimate the relationship between the price of security and the forces included in the five-factor model such as firm size, value-versus-growth dynamic, investment policy and profitability, in addition to the broad market, we can use the portfolio returns provided by Kenneth French's data library as exogenous variables in a rolling linear regression.

The following example accesses the data using the `pandas_datareader` module (see *Chapter 2, Market and Fundamental Data – Sources and Techniques*). It then computes the regression coefficients for windows of 21, 63, and 252 trading days:

```
factor_data = (web.DataReader('F-F_Research_Data_5_Factors_2x3_daily',
                               start=2005)[0].rename(columns={'Mkt-RF':
factor_data.index.names = ['date']
factors = factor_data.columns[:-1]
t = 1
ret = f'ret_{t:02}'
windows = [21, 63, 252]
for window in windows:
    print(window)
    betas = []
    for ticker, df in data.groupby('ticker', group_keys=False):
        model_data = df[[ret]].merge(factor_data, on='date').dropna()
        model_data[ret] -= model_data.RF
        rolling_ols = RollingOLS(endog=model_data[ret],
                               exog=sm.add_constant(model_data[factc
window=window)
```

```
factor_model = rolling_ols.fit(params_only=True).params.rename(
    columns={'const': 'ALPHA'})
result = factor_model.assign(ticker=ticker).set_index(
    'ticker', append=True).swaplevel()
betas.append(result)
betas = pd.concat(betas).rename(columns=lambda x: f'{x}_{window:02d}')
data = data.join(betas)
```

The risk factors just described are commonly used and also known as **smart beta factors** (see *Chapter 1, Machine Learning for Trading – From Idea to Execution*). In addition, hedge funds have started to resort to alpha factors derived from large-scale data mining exercises, which we'll turn to now.

WorldQuant's quest for formulaic alphas

We introduced WorldQuant in *Chapter 1, Machine Learning for Trading – From Idea to Execution*, as part of a trend toward crowd-sourcing investment strategies. WorldQuant maintains a virtual research center where quants worldwide compete to identify **alphas**. These alphas are trading signals in the form of computational expressions that help predict price movements, just like the common factors described in the previous section.

These **formulaic alphas** translate the mechanism to extract the signal from data into code, and they can be developed and tested individually with the goal to integrate their information into a broader automated strategy (Tulchinsky 2019). As stated repeatedly throughout this book, mining for signals in large datasets is prone to multiple testing bias and false discoveries. Regardless of these

important caveats, this approach represents a modern alternative to the more conventional features presented in the previous section.

Kakushadze (2016) presents 101 examples of such alphas, 80 percent of which were used in a real-world trading system at the time. It defines a range of functions that operate on cross-sectional or time-series data and can be combined, for example, in nested form.

The notebook `101_formulaic_alphas` shows how to implement these functions using pandas and NumPy, and also illustrates how to compute around 80 of these formulaic alphas for which we have input data (we lack, for example, accurate historical sector information).

Cross-sectional and time-series functions

The building blocks of the formulaic alphas proposed by Kakushadze (2016) are relatively simple expressions that compute over longitudinal or cross-sectional data that are readily implemented using pandas and NumPy.

The cross-sectional functions include ranking and scaling, as well as the group-wise normalization of returns, where the groups are intended to represent sector information at different levels of granularity:

We can directly translate the ranking function into a pandas expression, using a DataFrame as an argument in the format *number of period* \times *number of tickers*, as follows:

```
def rank(df):
    """Return the cross-sectional percentile rank
    Args:
```

```

:param df: tickers in columns, sorted dates in rows.
Returns:
    pd.DataFrame: the ranked values
"""
return df.rank(axis=1, pct=True)

```

There are also several time-series functions that will likely be familiar:

Function	Definition
<code>ts_{0}(x, d)</code>	Operator O applied to the time series for the past d days; non-integer number of days d converted to $\text{floor}(d)$.
<code>ts_lag(x, d)</code>	Value of x , d days ago.
<code>ts_delta(x, d)</code>	Difference between the value of x today and d days ago.
<code>ts_rank(x, d)</code>	Rank over the past d days.
<code>ts_mean(x, d)</code>	Simple moving average over the past d days.
<code>ts_weighted_mean(x, d)</code>	Weighted moving average over the past d days with linearly decaying weights $d, d-1, \dots, 1$ (rescaled to sum up to 1).
<code>ts_sum(x, d)</code>	Rolling sum over the past d days.
<code>ts_product(x, d)</code>	Rolling product over the past d days.
<code>ts_stddev(x, d)</code>	Moving standard deviation over the past d days.
<code>ts_max(x, d)</code> ,	Rolling maximum/minimum over the past d days.

<code>ts_min(x, d)</code>	
<code>ts_argmax(x, d), ts_argmin(x, d)</code>	Day of $ts_max(x, d)$, $ts_min(x, c)$.
<code>ts_correlation(x, y, d)</code>	Correlation of x and y for the past d days.

These time-series functions are also straightforward to implement using pandas' rolling window functionality. For the rolling weighted mean, for example, we can combine pandas with TA-Lib, as demonstrated in the previous section:

```
def ts_weighted_mean(df, period=10):
    """
    Linear weighted moving average implementation.
    :param df: a pandas DataFrame.
    :param period: the LWMA period
    :return: a pandas DataFrame with the LWMA.
    """
    return (df.apply(lambda x: WMA(x, timeperiod=period)))
```

To create the rolling correlation function, we provide two DataFrames containing time series for different tickers in the columns:

```
def ts_corr(x, y, window=10):
    """
    Wrapper function to estimate rolling correlations.
    :param x, y: pandas DataFrames.
    :param window: the rolling window.
    :return: DataFrame with time-series min for past 'window' days.
    """
    return x.rolling(window).corr(y)
```

In addition, the expressions use common operators, as we will see as we turn to the formulaic alphas that each combine several of the preceding functions.

Formulaic alpha expressions

To illustrate the computation of the alpha expressions, we need to create the following input tables using the sample of the 500 most-traded stocks from 2007-2016 from the previous section (see the notebook `sample_selection` for details on data preparation).

Each table contains columns of time series for individual tickers:

Variable	Description
<code>returns</code>	Daily close-to-close returns
<code>open</code> , <code>close</code> , <code>high</code> , <code>low</code> , <code>volume</code>	Standard definitions for daily price and volume data
<code>vwap</code>	Daily volume-weighted average price
<code>adv(d)</code>	Average daily dollar volume for the past d days

Our data does not include the daily volume-weighted average price required by many alpha expressions. To be able to demonstrate their computation, we very roughly approximate this value using the simple average of the daily open, high, low, and close prices.

Contrary to the common alphas presented in the previous section, the formulaic alphas do not come with an economic interpretation of

the risk exposure they represent. We will now demonstrate a few simply numbered instances.

Alpha 001

The first alpha expression is formulated as follows:

```
rank(ts_argmax(power(((returns < 0) ? ts_std(returns, 20) : close), 2.
```

The ternary operator `a ? b : c` executes *b* if *a* evaluates to `true`, and *c* otherwise. Thus, if the daily returns are positive, it squares the 20-day rolling standard deviation; otherwise, it squares the current close price. It then proceeds to rank the assets by the index of the day that shows the maximum for this value.

Using *c* and *r* to represent the close and return inputs, the alpha translates into Python using the previous functions and pandas methods, like so:

```
def alpha001(c, r):
    """(rank(ts_argmax(power(((returns < 0)
        ? ts_std(returns, 20)
        : close), 2.), 5)) -0.5)"""
    c[r < 0] = ts_std(r, 20)
    return (rank(ts_argmax(power(c, 2), 5)).mul(-.5)
        .stack().swaplevel())
```

For the 10-year sample of 500 stocks, the distribution of Alpha 001 values and its relationship with one-day forward returns looks as follows:

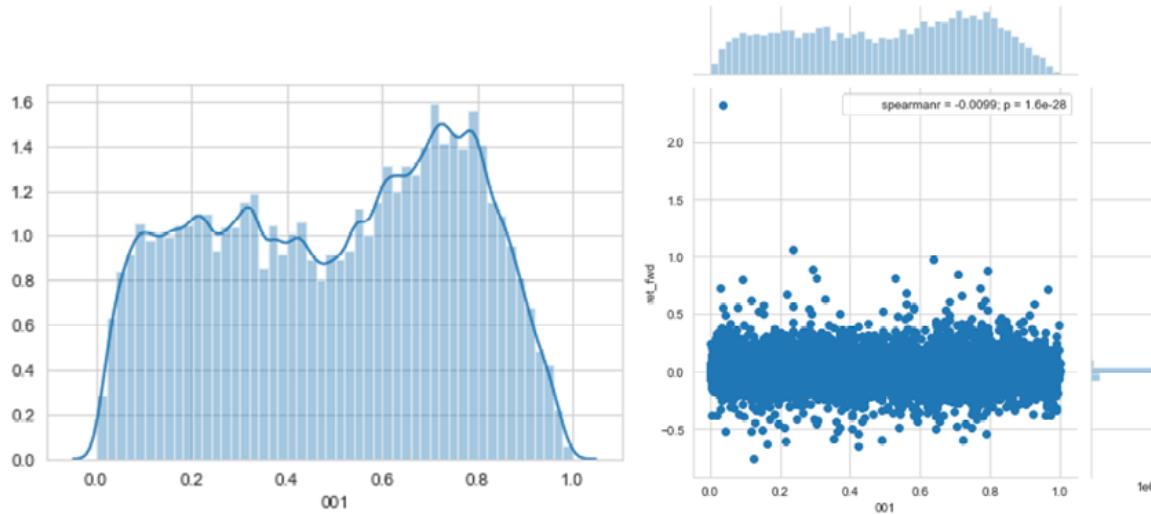


Figure A.16: Alpha 001 histogram and scatter plot

The **information coefficient (IC)** is fairly low, yet it is statistically significant at -0.0099 and the **mutual information (MI)** estimate yields 0.0129 (see *Chapter 4, Financial Feature Engineering – How to Research Alpha Factors*, and the notebook `101_formulaic_alphas`, for implementation details).

Alpha 054

Our second expression is the ratio of the difference between the low and the close price and the low and the high price, each multiplied by the open and close, respectively, raised to the fifth power:

```
-(low - close) * power(open, 5) / ((low - high) * power(close, 5))
```

Similarly, the translation into pandas is straightforward. We use `o`, `h`, `l`, and `c` to represent the DataFrames containing the respective price series for each ticker in the 500 columns:

```
def alpha054(o, h, l, c):
    """-(low - close) * power(open, 5) / ((low - high) * power(close,
```

```
return (l.sub(c).mul(o.pow(5)).mul(-1)
       .div(l.sub(h).replace(0, -0.0001).mul(c ** 5))
       .stack('ticker')
       .swaplevel())
```

In this case, the IC is significant at 0.025, while the MI score is lower at 0.005.

We will now take a look at how these different types of alpha factors compare from a univariate and a multivariate perspective.

Bivariate and multivariate factor evaluation

To evaluate the numerous factors, we rely on the various performance measures introduced in this book, including the following:

- Bivariate measures of the signal content of a factor with respect to the one-day forward returns
- Multivariate measures of feature importance for a gradient boosting model trained to predict the one-day forward returns using all factors
- Financial performance of portfolios invested according to factor quantiles using Alphalens

We will first discuss the bivariate metrics and then turn to the multivariate metrics; we will conclude by comparing the results. See the notebook `factor_evaluation` for the relevant code examples and

additional exploratory analysis, such as the correlation among the factors, which we'll omit here.

Information coefficient and mutual information

We will use the following bivariate metrics, which we introduced in *Chapter 4, Financial Feature Engineering – How to Research Alpha Factors*:

- The IC measured as the Spearman rank correlation
- The MI score computed using `mutual_info_regression`, provided by scikit-learn

The MI score uses a sample of 100,000 observations to limit the computational cost of the nearest neighbor computations. Both metrics are otherwise easy to compute and have been used repeatedly; see the notebook for implementation details. We will see, however, that they can yield quite different results.

Feature importance and SHAP values

To measure the predictive relevance of a feature given all other available factors, we can train a LightGBM gradient boosting model with default settings to predict the forward returns using all of the (approximately) 130 factors. The model uses 8.5 years of data to train 104 trees using early stopping. We will obtain test predictions for the last year of data, which yield a global IC of 3.40 and a daily average of 2.01.

We will then proceed to compute the feature importance and **SHapley Additive exPlanation (SHAP)** values, as described in *Chapter 12, Boosting Your Trading Strategy*; see the notebook for details. The influence plot in *Figure A.17* highlights how the values of the 20 most important features impact the model's predictions positively or negatively relative to the model's default output. In SHAP value terms, alphas 054 and 001 are among the top five factors:

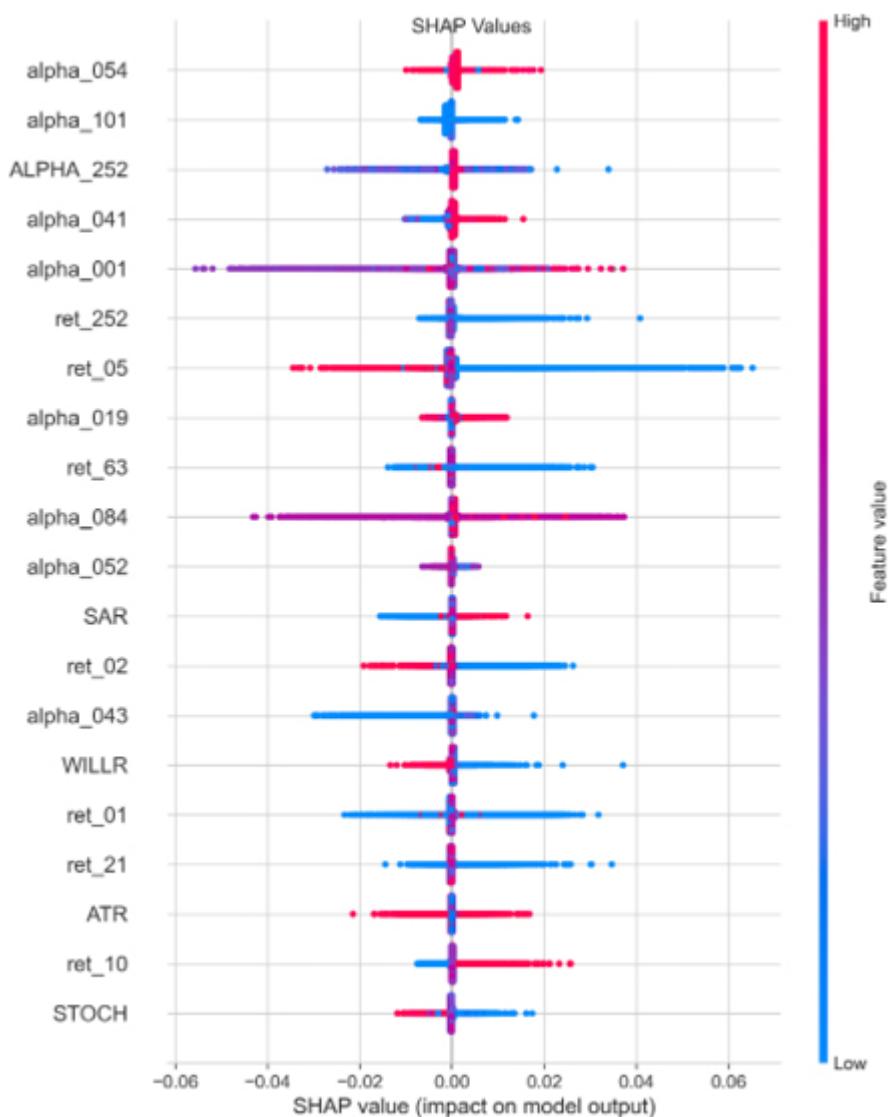


Figure A.17: SHAP values for common and formulaic alphas

Now, let's compare how the different metrics rate our factors.

Comparison – the top 25 features for each metric

The rank correlation among SHAP values and conventional feature importance measured as the weighted contribution of a feature to the reduction of the model's loss function is high at 0.89. It is also substantial between SHAP values and both univariate metrics at around 0.5.

Interestingly, though, MI and IC disagree significantly in their feature rankings with a correlation of only 0.16, as shown in the following diagram:

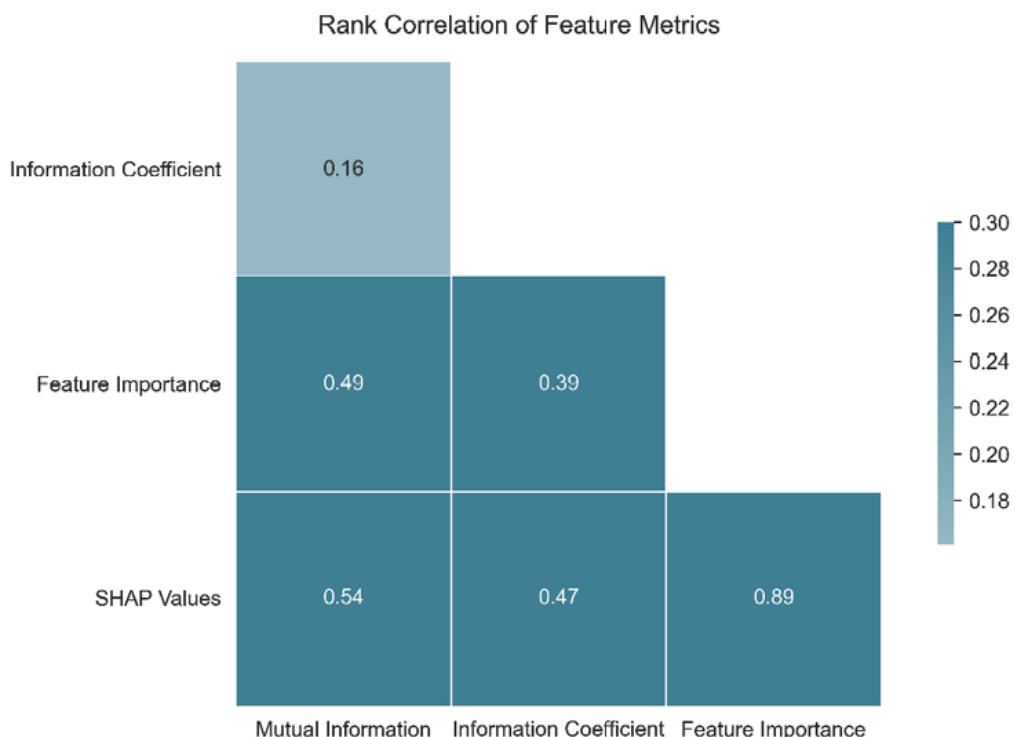


Figure A.18: Rank correlation of performance metrics

Figure A.19 displays the top 25 features according to each metric. Except for the MI score, which prefers the "common" alpha factors, features from both sources are ranked highly:

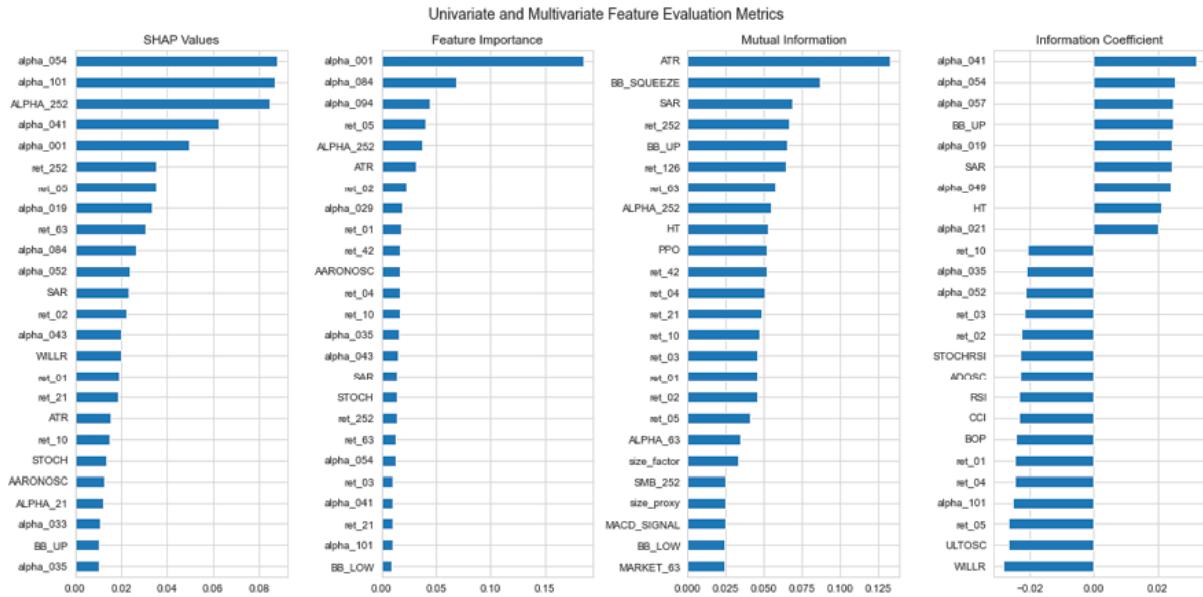


Figure A.19: Top 25 features for each performance metric

It is not immediately apparent why MI disagrees with the other metrics and why few of the features it assigns a high score play a significant role in the gradient boosting model. A possible explanation is that the computation uses only a 10 percent sample and the scores appear sensitive to the sample size.

Financial performance – Alphalens

Finally, we mostly care about the value of the trading signals emitted by an alpha factor. As introduced in *Chapter 4, Financial Feature Engineering – How to Research Alpha Factors*, and demonstrated repeatedly, Alphalens evaluates factor performance on a standalone basis.

The notebook `alphalens_analysis` lets you select an individual factor and compute how portfolios invested for a given horizon according to how factor quantile values would have performed.

The example in *Figure A.20* shows the result for Alpha 54; while portfolios in the top and bottom quintiles did achieve a 1.5 bps average spread on a daily basis, the cumulative returns of a long-short portfolio were negative:



Figure A.20: Alphalens performance metric for Alpha 54

Feel free to use the notebook as a template to evaluate the sample factors or others of your own choosing more systematically.

References

Abdel-Hamid, Ossama, Abdel-rahman Mohamed, Hui Jiang, Li Deng, Gerald Penn, and Dong Yu. 2014. "Convolutional Neural Networks for Speech Recognition." *IEEE/ACM Transactions on Audio, Speech, and Language Processing* 22 (10): 1533–45.

<https://doi.org/10.1109/TASLP.2014.2339736>.

Alqahtani, Hamed, Manolya Kavakli-Thorne, and Gulshan Kumar. 2019. "Applications of Generative Adversarial Networks (GANs): An Updated Review." *Archives of Computational Methods in Engineering*, December. <https://doi.org/10.1007/s11831-019-09388-y>.

Ang, Andrew, Robert J Hodrick, Yuhang Xing, and Xiaoyan Zhang. 2006. "The Cross-Section of Volatility and Expected Returns." *The Journal of Finance* 61 (1): 259–99.

Ang, Andrew. 2014. *Asset Management: A Systematic Approach to Factor Investing*. 1 edition. Oxford: Oxford University Press.

Araci, Dogu. 2019. "FinBERT: Financial Sentiment Analysis with Pre-Trained Language Models." *ArXiv:1908.10063 [Cs]*, August. <http://arxiv.org/abs/1908.10063>.

Arora, Saurabh, and Prashant Doshi. 2019. "A Survey of Inverse Reinforcement Learning: Challenges, Methods and Progress." *ArXiv:1806.06877 [Cs, Stat]*, August. <http://arxiv.org/abs/1806.06877>.

Asness, Clifford S., Tobias J. Moskowitz, and Lasse Heje Pedersen. 2013. "Value and Momentum Everywhere." *The Journal of Finance* 68 (3): 929–85. <https://www.jstor.org/stable/42002613>.

Bahdanau, Dzmitry, Kyunghyun Cho, and Yoshua Bengio. 2016. "Neural Machine Translation by Jointly Learning to Align and Translate." *ArXiv:1409.0473 [Cs, Stat]*, May. <http://arxiv.org/abs/1409.0473>.

Bailey, David H., Jonathan M. Borwein, and Marcos Lopez de Prado. 2016. *The Probability of Backtest Overfitting*. Journal of Computational Finance, September. <https://www.risk.net/node/2471206>.

Banz, Rolf W. 1981. "The Relationship between Return and Market Value of Common Stocks." *Journal of Financial Economics* 9 (1): 3–18.

Barberis, Nicholas, Andrei Shleifer, and Robert Vishny. 1998. "A Model of Investor Sentiment." *Journal of Financial Economics* 49 (3): 307–43.

Basu, Sanjoy, and others. 1981. "The Relationship between Earnings' Yield, Market Value and Return for NYSE Common Stocks: Further Evidence."

Bengio, Y., A. Courville, and P. Vincent. 2013. "Representation Learning: A Review and New Perspectives." *IEEE Transactions on Pattern Analysis and Machine Intelligence* 35 (8): 1798–1828. <https://doi.org/10.1109/TPAMI.2013.50>.

Betancourt, Michael. 2018. "A Conceptual Introduction to Hamiltonian Monte Carlo." *ArXiv:1701.02434 [Stat]*, July. <http://arxiv.org/abs/1701.02434>.

Bishop, Christopher. 2006. *Pattern Recognition and Machine Learning*. Information Science and Statistics. New York: Springer-Verlag. <https://www.springer.com/gp/book/9780387310732>.

Blei, David M., Andrew Y. Ng, and Michael I. Jordan. 2003. "Latent Dirichlet Allocation." *Journal of Machine Learning Research* 3 (Jan): 993–1022.

<http://jmlr.csail.mit.edu/papers/v3/blei03a.html>.

Burges, Chris J. C. 2010. "Dimension Reduction: A Guided Tour." *Foundations and Trends in Machine Learning*, January.

<https://www.microsoft.com/en-us/research/publication/dimension-reduction-a-guided-tour-2/>.

Byrd, David, Maria Hybinette, and Tucker Hybinette Balch. 2019. "ABIDES: Towards High-Fidelity Market Simulation for AI Research." ArXiv:1904.12066 [Cs], April.

<http://arxiv.org/abs/1904.12066>.

Casella, George, and Edward I. George. 1992. "Explaining the Gibbs Sampler." *The American Statistician* 46 (3): 167–74.

<https://doi.org/10.2307/2685208>.

Chan, Ernie. 2008. *Quantitative Trading: How to Build Your Own Algorithmic Trading Business*, 1 edition. ed. Wiley, Hoboken, N.J.

Chan, Ernie. 2013. *Algorithmic Trading: Winning Strategies and Their Rationale*. 1st ed. Wiley Publishing.

Chen, Tianqi, and Carlos Guestrin. 2016. "XGBoost: A Scalable Tree Boosting System." *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '16*, 785–94.

<https://doi.org/10.1145/2939672.2939785>.

Chen, Wei, Tie-yan Liu, Yanyan Lan, Zhi-ming Ma, and Hang Li. 2009. "Ranking Measures and Loss Functions in Learning to Rank."

In Advances in Neural Information Processing Systems 22, edited by Y. Bengio, D. Schuurmans, J. D. Lafferty, C. K. I. Williams, and A. Culotta, 315–323. Curran Associates, Inc.

<http://papers.nips.cc/paper/3708-ranking-measures-and-loss-functions-in-learning-to-rank.pdf>.

Cheung, W., 2010. *The Black–Litterman model explained*. J Asset Manag 11, 229–243. <https://doi.org/10.1057/jam.2009.28>.

Chib, Siddhartha, and Edward Greenberg. 1995. "Understanding the Metropolis-Hastings Algorithm." *The American Statistician* 49 (4): 327–35. <https://doi.org/10.1080/00031305.1995.10476177>.

Cho, Kyunghyun, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. "Learning Phrase Representations Using RNN Encoder–Decoder for Statistical Machine Translation." *In Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 1724–1734. Doha, Qatar: Association for Computational Linguistics. <https://doi.org/10.3115/v1/D14-1179>.

Chung, Junyoung, Caglar Gulcehre, Kyunghyun Cho, and Yoshua Bengio. 2014. "Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling." *NIPS 2014 Workshop on Deep Learning*, December 2014.
<https://nyuscholars.nyu.edu/en/publications/empirical-evaluation-of-gated-recurrent-neural-networks-on-sequen>.

Clarke, R., Silva, H. de, Thorley, S., 2002. *Portfolio Constraints and the Fundamental Law of Active Management*. Financial Analysts Journal 58, 48–66.

Creswell, Antonia, Tom White, Vincent Dumoulin, Kai Arulkumaran, Biswa Sengupta, and Anil A. Bharath. 2018. "Generative Adversarial Networks: An Overview." *IEEE Signal Processing Magazine* 35 (1): 53–65.
<https://doi.org/10.1109/MSP.2017.2765202>.

Cubuk, Ekin D. 2019. "AutoAugment: Learning Augmentation Strategies From Data." *CVFPR*, 11.

Cummins, Mark, and Andrea Bucca. 2012. "Quantitative Spread Trading on Crude Oil and Refined Products Markets." *Quantitative Finance* 12 (12): 1857–75.
<https://doi.org/10.1080/14697688.2012.715749>.

Cybenko, G. 1989. "Approximation by Superpositions of a Sigmoidal Function." *Mathematics of Control, Signals and Systems* 2 (4): 303–14.
<https://doi.org/10.1007/BF02551274>.

David H. Bailey et al. (2015), *Backtest Overfitting: An Interactive Example*. <http://datagrid.lbl.gov/backtest/>.

De Prado, Marcos Lopez. 2018. *Advances in Financial Machine Learning*. John Wiley & Sons.

DeMiguel, V., Garlappi, L., Uppal, R., 2009. *Optimal Versus Naive Diversification: How Inefficient is the 1/N Portfolio Strategy?* Rev Financ Stud 22, 1915–1953. <https://doi.org/10.1093/rfs/hhm075>.

Devlin, Jacob, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. "BERT: Pre-Training of Deep Bidirectional Transformers for Language Understanding." In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1*

(Long and Short Papers), 4171–4186. Minneapolis, Minnesota: Association for Computational Linguistics.
<https://doi.org/10.18653/v1/N19-1423>.

Dumais, S. T., G. W. Furnas, T. K. Landauer, S. Deerwester, and R. Harshman. 1988. "Using Latent Semantic Analysis to Improve Access to Textual Information." In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 281–285. CHI '88. Washington, D.C., USA: Association for Computing Machinery.
<https://doi.org/10.1145/57167.57214>.

Elliott, Robert J., John Van Der Hoek, and William P. Malcolm. 2005. "Pairs Trading." *Quantitative Finance* 5 (3): 271–76.
<https://doi.org/10.1080/14697680500149370>.

Esposito, F., D. Malerba, G. Semeraro, and J. Kay. 1997. "A Comparative Analysis of Methods for Pruning Decision Trees." *IEEE Transactions on Pattern Analysis and Machine Intelligence* 19 (5): 476–91.
<https://doi.org/10.1109/34.589207>.

Esteban, Cristóbal, Stephanie L. Hyland, and Gunnar Rätsch. 2017. "Real-Valued (Medical) Time Series Generation with Recurrent Conditional GANs." *ArXiv:1706.02633 [Cs, Stat]*, December.
<http://arxiv.org/abs/1706.02633>.

Fabozzi, Frank J, Sergio M Focardi, and Petter N Kolm. 2010. *Quantitative Equity Investing: Techniques and Strategies*. John Wiley & Sons.

Fama, E.F., French, K.R., 2004. *The Capital Asset Pricing Model: Theory and Evidence*. Journal of Economic Perspectives 18, 25–46.
<https://doi.org/10.1257/0895330042162430>.

Fama, Eugene F, and James D MacBeth. 1973. "Risk, Return, and Equilibrium: Empirical Tests." *Journal of Political Economy* 81 (3): 607–36.

Fama, Eugene F, and Kenneth R French. 1993. "Common Risk Factors in the Returns on Stocks and Bonds." *Journal of Financial Economics* 33: 3–56.

Fama, Eugene F, and Kenneth R French. 1998. "Value versus Growth: The International Evidence." *The Journal of Finance* 53 (6): 1975–99.

Fama, Eugene F., and Kenneth R. French. 2015. "A Five-Factor Asset Pricing Model." *Journal of Financial Economics* 116 (1): 1–22.
<https://doi.org/10.1016/j.jfineco.2014.10.010>.

Fawcett, Tom. 2006. "An Introduction to ROC Analysis." *Pattern Recognition Letters*, ROC Analysis in Pattern Recognition, 27 (8): 861–74. <https://doi.org/10.1016/j.patrec.2005.10.010>.

Fefferman, Charles, Sanjoy Mitter, and Hariharan Narayanan. 2016. "Testing the Manifold Hypothesis." *Journal of the American Mathematical Society* 29 (4): 983–1049.
<https://doi.org/10.1090/jams/852>.

Fei-Fei, Li. 2015. "ImageNet Large Scale Visual Recognition Challenge." *International Journal of Computer Vision* 115 (3): 211–52.
<https://doi.org/10.1007/s11263-015-0816-y>.

Fisher, Walter D. 1958. "On Grouping for Maximum Homogeneity." *Journal of the American Statistical Association* 53 (284): 789–98.
<https://doi.org/10.2307/2281952>.

Freund, Yoav, and Robert E Schapire. 1997. "A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting."

Journal of Computer and System Sciences 55 (1): 119–39.
<https://doi.org/10.1006/jcss.1997.1504>.

Friedman, Jerome H. 2001. "Greedy Function Approximation: A Gradient Boosting Machine." *The Annals of Statistics* 29 (5): 1189–1232. <https://www.jstor.org/stable/2699986>.

Fu, Rao, Jie Chen, Shutian Zeng, Yiping Zhuang, and Agus Sudjianto. 2019. "Time Series Simulation by Conditional Generative Adversarial Net." *ArXiv:1904.11419 [Cs, Eess, Stat]*, April. <http://arxiv.org/abs/1904.11419>.

Gatev, Evan, William N. Goetzmann, and K. Geert Rouwenhorst. 2006. "Pairs Trading: Performance of a Relative-Value Arbitrage Rule." *The Review of Financial Studies* 19 (3): 797–827.
<https://doi.org/10.1093/rfs/hhj020>.

Gelman, Andrew, John B. Carlin, Hal S. Stern, David B. Dunson, Aki Vehtari, and Donald B. Rubin. 2013. *Bayesian Data Analysis, Third Edition*. CRC Press.

Gómez, David, and Alfonso Rojas. 2015. "An Empirical Overview of the No Free Lunch Theorem and Its Effect on Real-World Machine Learning Classification." *Neural Computation* 28 (1): 216–28.
https://doi.org/10.1162/NECO_a_00793.

Gonzalo, Jesús, and Tae-Hwy Lee. 1998. "Pitfalls in Testing for Long-Run Relationships." *Journal of Econometrics* 86 (1): 129–54.
[https://doi.org/10.1016/S0304-4076\(97\)00111-5](https://doi.org/10.1016/S0304-4076(97)00111-5).

Goodfellow, Ian, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. "Generative Adversarial Nets." In *Advances in Neural*

Information Processing Systems 27, edited by Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, 2672–2680. Curran Associates, Inc.

<http://papers.nips.cc/paper/5423-generative-adversarial-nets.pdf>

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT press.

Goodfellow, Ian. 2014. "Multi-Digit Number Recognition from Street View Imagery Using Deep Convolutional Neural Networks." In *ICLR2014*.

Goyal, Amit. 2012. "Empirical Cross-Sectional Asset Pricing: A Survey." *Financial Markets and Portfolio Management* 26 (1): 3–38.
<https://doi.org/10.1007/s11408-011-0177-7>.

Graham, Benjamin, David Dodd, and David Le Fevre Dodd. 1934. *Security Analysis: The Classic 1934 Edition*. McGraw Hill Professional.

Green, Jeremiah, John R. M. Hand, and X. Frank Zhang. 2017. "The Characteristics That Provide Independent Information about Average U.S. Monthly Stock Returns." *The Review of Financial Studies* 30 (12): 4389–4436. <https://doi.org/10.1093/rfs/hhx019>.

Grinold, R.C., 1989. *The fundamental law of active management*. The Journal of Portfolio Management 15, 30–37.
<https://doi.org/10.3905/jpm.1989.409211>.

Gu, S., Kelly, B., and Xu, D. 2020 "Autoencoder Asset Pricing Models." *Journal of Econometrics* (forthcoming).

Gu, Shihao, Bryan Kelly, and Dacheng Xiu. 2020. "Empirical Asset Pricing via Machine Learning." *The Review of Financial Studies*, no.

hhaa009 (February). <https://doi.org/10.1093/rfs/hhaa009>.

Harris, Larry. 2003. *Trading and Exchanges: Market Microstructure for Practitioners*. Oxford University Press.

Hasselt, Hado van, Arthur Guez, and David Silver. 2015. "Deep Reinforcement Learning with Double Q-Learning." *ArXiv:1509.06461 [Cs]*, September. <http://arxiv.org/abs/1509.06461>.

Hastie, Trevor, Robert Tibshirani, and Jerome Friedman. 2009. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction, Second Edition*. 2nd ed. Springer Series in Statistics. New York: Springer-Verlag. <https://doi.org/10.1007/978-0-387-84858-7>.

Hastie, Trevor, Robert Tibshirani, and Martin Wainwright. 2015. *Statistical Learning with Sparsity: The Lasso and Generalizations*. CRC press.

He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. "Deep Residual Learning for Image Recognition." *ArXiv:1512.03385 [Cs]*, December. <http://arxiv.org/abs/1512.03385>.

Hendricks, Dieter, and Diane Wilcox. 2014. "A Reinforcement Learning Extension to the Almgren-Chriss Framework for Optimal Trade Execution." In *2014 IEEE Conference on Computational Intelligence for Financial Engineering Economics (CIFEr)*, 457–64. <https://doi.org/10.1109/CIFEr.2014.6924109>.

Hessel, Matteo, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. 2017. "Rainbow: Combining Improvements in Deep Reinforcement Learning." *ArXiv:1710.02298 [Cs]*, October. <http://arxiv.org/abs/1710.02298>.

Hihi, Salah El, and Yoshua Bengio. 1996. "Hierarchical Recurrent Neural Networks for Long-Term Dependencies." In *Advances in Neural Information Processing Systems 8*, edited by D. S. Touretzky, M. C. Mozer, and M. E. Hasselmo, 493–499. MIT Press.

<http://papers.nips.cc/paper/1102-hierarchical-recurrent-neural-networks-for-long-term-dependencies.pdf>.

Hochreiter, Sepp, and Jürgen Schmidhuber. 1996. "LSTM Can Solve Hard Long Time Lag Problems." In *Proceedings of the 9th International Conference on Neural Information Processing Systems*, 473–479. NIPS'96. Denver, Colorado: MIT Press.

Hochreiter, Sepp, Yoshua Bengio, Paolo Frasconi, Jürgen Schmidhuber, and others. 2001. "Gradient Flow in Recurrent Nets: The Difficulty of Learning Long-Term Dependencies."

Hoerl, Arthur E, and Robert W Kennard. 1970. "Ridge Regression: Biased Estimation for Nonorthogonal Problems." *Technometrics* 12 (1): 55–67.

Hoffman, Matthew D., and Andrew Gelman. 2011. "The No-U-Turn Sampler: Adaptively Setting Path Lengths in Hamiltonian Monte Carlo." *ArXiv:1111.4246 [Cs, Stat]*, November.

<http://arxiv.org/abs/1111.4246>.

Hofmann, Thomas. 2001. "Unsupervised Learning by Probabilistic Latent Semantic Analysis." *Machine Learning* 42 (1): 177–96.

[https://doi.org/10.1023/A:1007617005950.](https://doi.org/10.1023/A:1007617005950)

Hong, Harrison, Terence Lim, and Jeremy C. Stein. 2000. "Bad News Travels Slowly: Size, Analyst Coverage, and the Profitability of

Momentum Strategies." *The Journal of Finance* 55 (1): 265–95.
<https://doi.org/10.1111/0022-1082.00206>.

Hornik, Kurt. 1991. "Approximation Capabilities of Multilayer Feedforward Networks." *Neural Networks* 4 (2): 251–57.
[https://doi.org/10.1016/0893-6080\(91\)90009-T](https://doi.org/10.1016/0893-6080(91)90009-T).

Hou, Kewei, Chen Xue, and Lu Zhang. 2015. "Digesting Anomalies: An Investment Approach." *The Review of Financial Studies* 28 (3): 650–705. <https://doi.org/10.1093/rfs/hhu068>.

Hou, K., Xue, C., Zhang, L., 2017. *Replicating Anomalies* (SSRN Scholarly Paper No. ID 2961979). Social Science Research Network, Rochester, NY.

Huang, Gao, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. 2018. "Densely Connected Convolutional Networks." *ArXiv:1608.06993 [Cs]*, January.
<http://arxiv.org/abs/1608.06993>.

Ismail Fawaz, Hassan, Germain Forestier, Jonathan Weber, Lhassane Idoumghar, and Pierre-Alain Muller. 2019. "Deep Learning for Time Series Classification: A Review." *Data Mining and Knowledge Discovery* 33 (4): 917–63. <https://doi.org/10.1007/s10618-019-00619-1>.

Jaeger, Herbert. 2001. "The 'Echo State' Approach to Analysing and Training Recurrent Neural Networks with an Erratum Note." *Bonn, Germany: German National Research Center for Information Technology GMD Technical Report* 148 (34): 13.

James, Gareth, Daniela Witten, Trevor Hastie, and Robert Tibshirani. 2013. *An Introduction to Statistical Learning*. Vol. 112. Springer.

Jegadeesh, Narasimhan, and Sheridan Titman. 1993. "Returns to Buying Winners and Selling Losers: Implications for Stock Market Efficiency." *The Journal of Finance* 48 (1): 65–91.

Jones, Charles. 2018. "Understanding the Market for Us Equity Market Data." NYSE.

<https://www0.gsb.columbia.edu/faculty/cjones/papers/2018.08.31%20US%20Equity%20Market%20Data%20Paper.pdf>.

JP Morgan, 2012. *Improving on risk parity – Hedging Forecast Uncertainty.*

https://am.jpmorgan.com/blobcontent/1378404528937/83456/11_77%20Risk%20Parity_.pdf.

Ke, Guolin, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. 2017. "LightGBM: A Highly Efficient Gradient Boosting Decision Tree." In *Advances in Neural Information Processing Systems* 30, edited by I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, 3146–3154. Curran Associates, Inc.

<http://papers.nips.cc/paper/6907-lightgbm-a-highly-efficient-gradient-boosting-decision-tree.pdf>.

Kearns, Michael, and Yuriy Nevmyvaka. 2013. "Machine Learning for Market Microstructure and High Frequency Trading." *High Frequency Trading: New Realities for Traders, Markets, and Regulators*.

Kelley, David. 2019. "Which Leading Indicators Have Done Better at Signaling Past Recessions?" *Chicago Fed Letter*, no. 425: 1.

Kelly, Bryan T., Seth Pruitt, and Yinan Su. 2019. "Characteristics Are Covariances: A Unified Model of Risk and Return." *Journal of*

Financial Economics 134 (3): 501–24.

<https://doi.org/10.1016/j.jfineco.2019.05.001>.

Kelly, J.L., 2011. *A New Interpretation of Information Rate*, in: The Kelly Capital Growth Investment Criterion, World Scientific Handbook in Financial Economics Series. WORLD SCIENTIFIC, pp. 25–34.

https://doi.org/10.1142/9789814293501_0003.

Kingma, Diederik P., and Max Welling. 2014. "Auto-Encoding Variational Bayes." *ArXiv:1312.6114 [Cs, Stat]*, May.

<http://arxiv.org/abs/1312.6114>.

Kingma, Diederik, and Jimmy Ba. 2014. "Adam: A Method for Stochastic Optimization," December.

<https://arxiv.org/abs/1412.6980v8>.

Kingma, Diederik P., and Max Welling. 2019. "An Introduction to Variational Autoencoders." *Foundations and Trends® in Machine Learning* 12 (4): 307–92. <https://doi.org/10.1561/2200000056>.

Kolanovic, Marko, and Rajesh Krishnamachari. 2017. "Big Data and AI Strategies - Machine Learning and Alternative Data Approach to Investing." White Paper. JP Morgan.

<http://www.fullertreacymoney.com/system/data/files/PDFs/2017/October/18th/Big%20Data%20and%20AI%20Strategies%20-%20Machine%20Learning%20and%20Alternative%20Data%20Approach%20to%20Investing.pdf>.

Koshiyama, Adriano, Nick Firoozye, and Philip Treleaven. 2019. "Generative Adversarial Networks for Financial Trading Strategies Fine-Tuning and Combination." *ArXiv:1901.01751 [Cs, q-Fin, Stat]*, March. <http://arxiv.org/abs/1901.01751>.

Krauss, Christopher, Xuan Anh Do, and Nicolas Huck. 2017. "Deep Neural Networks, Gradient-Boosted Trees, Random Forests: Statistical Arbitrage on the S&P 500." *European Journal of Operational Research* 259 (2): 689–702.
<https://doi.org/10.1016/j.ejor.2016.10.031>.

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E Hinton. 2012. "ImageNet Classification with Deep Convolutional Neural Networks." In *Advances in Neural Information Processing Systems 25*, edited by F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, 1097–1105. Curran Associates, Inc.
<http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf>.

Lecun, Y., L. Bottou, Y. Bengio, and P. Haffner. 1998. "Gradient-Based Learning Applied to Document Recognition." *Proceedings of the IEEE* 86 (11): 2278–2324. <https://doi.org/10.1109/5.726791>.

LeCun, Yann, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. 1989. "Backpropagation Applied to Handwritten Zip Code Recognition." *Neural Computation* 1 (4): 541–51.

LeCun, Yann, Yoshua Bengio, and Geoffrey Hinton. 2015. "Deep Learning." *Nature* 521 (7553): 436–44.
<https://doi.org/10.1038/nature14539>.

Ledig, Christian, Lucas Theis, Ferenc Huszar, Jose Caballero, Andrew Cunningham, Alejandro Acosta, Andrew Aitken, et al. 2017. "Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network." In , 4681–90.

http://openaccess.thecvf.com/content_cvpr_2017/html/Ledig_Photo-Realistic_Single_Image_CVPR_2017_paper.html.

Ledoit, O., Wolf, M., 2003. *Improved estimation of the covariance matrix of stock returns with an application to portfolio selection*. Journal of Empirical Finance 10, 603–621. [https://doi.org/10.1016/S0927-5398\(03\)00007-0](https://doi.org/10.1016/S0927-5398(03)00007-0).

Levy, Omer, and Yoav Goldberg. 2014. "Neural Word Embedding as Implicit Matrix Factorization." In *Advances in Neural Information Processing Systems 27*, edited by Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, 2177–2185. Curran Associates, Inc. <http://papers.nips.cc/paper/5477-neural-word-embedding-as-implicit-matrix-factorization.pdf>.

Lin, Long-Ji, and Tom M Mitchell. 1992. *Memory Approaches to Reinforcement Learning in Non-Markovian Domains*. Citeseer.

Liu, Yinhan, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. "RoBERTa: A Robustly Optimized BERT Pretraining Approach." *ArXiv:1907.11692 [Cs]*, July. <http://arxiv.org/abs/1907.11692>.

Lo, A.W., 2002. *The Statistics of Sharpe Ratios*. <https://doi.org/10.2469/faj.v58.n4.2453>.

Maaten, Laurens van der, and Geoffrey Hinton. 2008. "Visualizing Data Using T-SNE." *Journal of Machine Learning Research* 9 (Nov): 2579–2605.

<http://www.jmlr.org/papers/v9/vandermaaten08a.html>.

Madhavan, Ananth. 2002. "Market Microstructure: A Practitioner's Guide." *Financial Analysts Journal* 58 (5): 28–42.
www.jstor.org/stable/4480415.

Madhavan, Ananth. 2000. "Market Microstructure: A Survey." *Journal of Financial Markets* 3 (3): 205–58.
[https://doi.org/10.1016/S1386-4181\(00\)00007-0](https://doi.org/10.1016/S1386-4181(00)00007-0).

Malkiel, Burton G. 2003. "The Efficient Market Hypothesis and Its Critics." *Journal of Economic Perspectives* 17 (1): 59–82.
<https://doi.org/10.1257/089533003321164958>.

Man, Xiliu, Tong Luo, and Jianwu Lin. 2019. "Financial Sentiment Analysis(FSA): A Survey." In *2019 IEEE International Conference on Industrial Cyber Physical Systems (ICPS)*, 617–22.
<https://doi.org/10.1109/ICPHYS.2019.8780312>.

Markowitz, H., 1952. *Portfolio Selection*. The Journal of Finance 7, 77–91. <https://doi.org/10.2307/2975974>.

Meredith, Mike, and John Kruschke. 2018. "Bayesian Estimation Supersedes the T-Test," 14.

Michaud, Richard O., Esch, D.N., Michaud, Robert O., 2017. *The "Fundamental Law of Active Management" is No Law of Anything*.
<https://doi.org/10.2139/ssrn.2834020>.

Mikolov, Tomas, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. "Distributed Representations of Words and Phrases and Their Compositionality." In *Proceedings of the 26th International Conference on Neural Information Processing Systems – Volume 2*, 3111–3119. NIPS'13. USA: Curran Associates Inc.
<http://dl.acm.org/citation.cfm?id=2999792.2999959>.

Miller, Rena S. 2016. "High Frequency Trading: Overview of Recent Developments." *High Frequency Trading*, 19.

Mimno, David, Hanna M. Wallach, Edmund Talley, Miriam Leenders, and Andrew McCallum. 2011. "Optimizing Semantic Coherence in Topic Models." In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, 262–272. EMNLP '11. Edinburgh, United Kingdom: Association for Computational Linguistics.

Mitchell, Tom M. 1997. "Machine Learning."

Mnih, Andriy, and Koray Kavukcuoglu. 2013. "Learning Word Embeddings Efficiently with Noise-Contrastive Estimation." In *Advances in Neural Information Processing Systems* 26, edited by C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, 2265–2273. Curran Associates, Inc.

<http://papers.nips.cc/paper/5165-learning-word-embeddings-efficiently-with-noise-contrastive-estimation.pdf>.

Mnih, Volodymyr, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. "Playing Atari with Deep Reinforcement Learning." *ArXiv:1312.5602 [Cs]*, December. <http://arxiv.org/abs/1312.5602>.

Mnih, Volodymyr, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, et al. 2015. "Human-Level Control through Deep Reinforcement Learning." *Nature* 518 (7540): 529–33.
<https://doi.org/10.1038/nature14236>.

Morin, Frederic, and Yoshua Bengio. 2005. "Hierarchical Probabilistic Neural Network Language Model." In *Aistats*, 5:246–52.

Nakamoto, Yukikazu. 2011. "A Short Introduction to Learning to Rank." *IEICE Transactions on Information and Systems* E94-D (1): 1–2. <https://doi.org/10.1587/transinf.E94.D.1>.

Nasseri, Alya Al, Allan Tucker, and Sergio de Cesare. 2015. "Quantifying StockTwits Semantic Terms' Trading Behavior in Financial Markets: An Effective Application of Decision Tree Algorithms." *Expert Systems with Applications* 42 (23): 9192–9210. <https://doi.org/10.1016/j.eswa.2015.08.008>.

Netzer, Yuval. 2011. "Reading Digits in Natural Images with Unsupervised Feature Learning." In *NIPS Workshop on Deep Learning*.

Nevmyvaka, Yuriy, Yi Feng, and Michael Kearns. 2006. "Reinforcement Learning for Optimized Trade Execution." In *Proceedings of the 23rd International Conference on Machine Learning*, 673–680. ICML '06. Pittsburgh, Pennsylvania, USA: Association for Computing Machinery. <https://doi.org/10.1145/1143844.1143929>.

Ng, Andrew Y., and Michael I. Jordan. 2002. "On Discriminative vs. Generative Classifiers: A Comparison of Logistic Regression and Naive Bayes." In *Advances in Neural Information Processing Systems* 14, edited by T. G. Dietterich, S. Becker, and Z. Ghahramani, 841–848. MIT Press. <http://papers.nips.cc/paper/2020-on-discriminative-vs-generative-classifiers-a-comparison-of-logistic-regression-and-naive-bayes.pdf>.

Nilsson, Nils J. 2009. *The Quest for Artificial Intelligence*. Cambridge University Press.

Novy-Marx, R., 2015. *Backtesting Strategies Based on Multiple Signals* (Working Paper No. 21329). National Bureau of Economic Research. <https://doi.org/10.3386/w21329>.

Odena, Augustus. 2019. "Open Questions about Generative Adversarial Networks." *Distill* 4 (4): e18. <https://doi.org/10.23915/distill.00018>.

Pan, Zhaoqing, Weijie Yu, Xiaokai Yi, Asifullah Khan, Feng Yuan, and Yuhui Zheng. 2019. "Recent Progress on Generative Adversarial Networks (GANs): A Survey." *IEEE Access* 7: 36322–33. <https://doi.org/10.1109/ACCESS.2019.2905015>.

Pennington, Jeffrey, Richard Socher, and Christoper Manning. 2014. "Glove: Global Vectors for Word Representation." In *EMNLP*, 14:1532–43. <https://doi.org/10.3115/v1/D14-1162>.

Perold, A.F., 2004. *The Capital Asset Pricing Model*. Journal of Economic Perspectives 18, 3–24. <https://doi.org/10.1257/0895330042162340>.

Prabhavalkar, Rohit, Kanishka Rao, Tara N. Sainath, Bo Li, Leif Johnson, and Navdeep Jaitly. 2017. "A Comparison of Sequence-to-Sequence Models for Speech Recognition." In *Interspeech 2017*, 939–43. ISCA. <https://doi.org/10.21437/Interspeech.2017-233>.

Prado, M.L. de, 2016. "Building Diversified Portfolios that Outperform Out of Sample." *The Journal of Portfolio Management* 42, 59–69. <https://doi.org/10.3905/jpm.2016.42.4.059>.

Preis, Tobias, Helen Susannah Moat, and H. Eugene Stanley. 2013. "Quantifying Trading Behavior in Financial Markets Using Google

Trends." *Scientific Reports* 3 (1): 1–6.
<https://doi.org/10.1038/srep01684>.

Prokhorenkova, Liudmila, Gleb Gusev, Aleksandr Vorobev, Anna Veronika Dorogush, and Andrey Gulin. 2019. "CatBoost: Unbiased Boosting with Categorical Features." *ArXiv:1706.09516 [Cs]*, January.
<http://arxiv.org/abs/1706.09516>.

Radford, Alec, Luke Metz, and Soumith Chintala. 2016. "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks." *ArXiv:1511.06434 [Cs]*, January.
<http://arxiv.org/abs/1511.06434>.

Raffinot, T., 2017. *Hierarchical Clustering-Based Asset Allocation*. The Journal of Portfolio Management 44, 89–99.
<https://doi.org/10.3905/jpm.2018.44.2.089>.

Rasekhshaffe, Keywan Christian, and Robert C. Jones. 2019. "Machine Learning for Stock Selection." *Financial Analysts Journal* 75 (3): 70–88. <https://doi.org/10.1080/0015198X.2019.1596678>.

Redmon, Joseph. 2016. "You Only Look Once: Unified, Real-Time Object Detection." *ArXiv:1506.02640 [Cs]*, May.

Reed, Scott, Zeynep Akata, Santosh Mohan, Samuel Tenka, Bernt Schiele, and Honglak Lee. 2016. "Learning What and Where to Draw." *ArXiv:1610.02454 [Cs]*, October.
<http://arxiv.org/abs/1610.02454>.

Reinganum, Marc R. 1981. "Misspecification of Capital Asset Pricing: Empirical Anomalies Based on Earnings' Yields and Market Values." *Journal of Financial Economics* 9 (1): 19–46.

Ren, Shaoqing, Kaiming He, Ross Girshick, and Jian Sun. 2015. "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks." In *Advances in Neural Information Processing Systems* 28, edited by C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, 91–99. Curran Associates, Inc.
<http://papers.nips.cc/paper/5638-faster-r-cnn-towards-real-time-object-detection-with-region-proposal-networks.pdf>.

Roa-Vicens, Jacobo, Cyrine Chtourou, Angelos Filos, Francisco Rullan, Yarin Gal, and Ricardo Silva. 2019. "Towards Inverse Reinforcement Learning for Limit Order Book Dynamics." *ArXiv:1906.04813 [Cs, q-Fin, Stat]*, June.
<http://arxiv.org/abs/1906.04813>.

Röder, Michael, Andreas Both, and Alexander Hinneburg. 2015. "Exploring the Space of Topic Coherence Measures." In *Proceedings of the Eighth ACM International Conference on Web Search and Data Mining*, 399–408. WSDM '15. Shanghai, China: Association for Computing Machinery.
<https://doi.org/10.1145/2684822.2685324>.

Rokach, Lior, and Oded Z. Maimon. 2008. *Data Mining with Decision Trees: Theory and Applications*. World Scientific.

Roll, Richard, and Stephen A. Ross. 1984. "The Arbitrage Pricing Theory Approach to Strategic Portfolio Planning." *Financial Analysts Journal* 40 (3): 14–26. <https://doi.org/10.2469/faj.v40.n3.14>.

Romero, P.J., Balch, T., 2014. *What Hedge Funds Really Do: An Introduction to Portfolio Management*. Business Expert Press.

Ruder, Sebastian. 2017. "An Overview of Gradient Descent Optimization Algorithms." *ArXiv:1609.04747 [Cs]*, June. <http://arxiv.org/abs/1609.04747>.

Salimans, Tim, Diederik P. Kingma, and Max Welling. 2015. "Markov Chain Monte Carlo and Variational Inference: Bridging the Gap." *ArXiv:1410.6460 [Stat]*, May. <http://arxiv.org/abs/1410.6460>.

Samuelson, P., Thorp, E., T. Kassouf, S., 1968. *Beat the Market: A Scientific Stock Market System*. Journal of the American Statistical Association 63, 1049. <https://doi.org/10.2307/2283900>.

Saul, Lawrence K, and Sam T Roweis. 2000. "An Introduction to Locally Linear Embedding." Unpublished. Available at: <https://cs.nyu.edu/~roweis/lle/papers/lleintro.pdf>.

Schapire, Robert E., and Yoav Freund. 2012. *Boosting: Foundations and Algorithms*. MIT Press.

Schaul, Tom, John Quan, Ioannis Antonoglou, and David Silver. 2015. "Prioritized Experience Replay." *ArXiv:1511.05952 [Cs]*, November. <http://arxiv.org/abs/1511.05952>.

Schuster, M., and K.K. Paliwal. 1997. "Bidirectional Recurrent Neural Networks." *IEEE Transactions on Signal Processing* 45 (11): 2673–81. <https://doi.org/10.1109/78.650093>.

Sezer, Omer Berat, and Ahmet Murat Ozbayoglu. 2018. "Algorithmic Financial Trading with Deep Convolutional Neural Networks: Time Series to Image Conversion Approach." *Applied Soft Computing* 70 (September): 525–38. <https://doi.org/10.1016/j.asoc.2018.04.024>.

Sievert, Carson, and Kenneth Shirley. 2014. "LDAvis: A Method for Visualizing and Interpreting Topics." In *Proceedings of the Workshop on Interactive Language Learning, Visualization, and Interfaces*, 63–70.

Sigtia, Siddharth, Emmanouil Benetos, Srikanth Cherla, Tillman Weyde, A. Garcez, and Simon Dixon. 2014. "RNN-Based Music Language Models for Improving Automatic Music Transcription."

Simonyan, Karen. 2015. "Very Deep Convolutional Networks for Large-Scale Image Recognition." *ArXiv:1409.1556 [Cs]*, April.

Srivastava, Nitish, Elman Mansimov, and Ruslan Salakhutdinov. 2016. "Unsupervised Learning of Video Representations Using LSTMs," 10.

Stevens, Keith, Philip Kegelmeyer, David Andrzejewski, and David Buttler. 2012. "Exploring Topic Coherence over Many Models and Many Topics."

Strumeyer, Gary. 2017. *The Capital Markets: Evolution of the Financial Ecosystem*. John Wiley & Sons.

Sutskever, Ilya, James Martens, George Dahl, and Geoffrey Hinton. 2013. "On the Importance of Initialization and Momentum in Deep Learning." In *International Conference on Machine Learning*, 1139–47. <http://proceedings.mlr.press/v28/sutskever13.html>.

Sutton, Richard S, and Andrew G Barto. 2018. *Reinforcement Learning: An Introduction*. MIT press.

Sutton, Richard S, David A. McAllester, Satinder P. Singh, and Yishay Mansour. 2000. "Policy Gradient Methods for Reinforcement Learning with Function Approximation." In *Advances in Neural Information Processing Systems* 12, edited by S. A. Solla, T. K. Leen,

and K. Müller, 1057–1063. MIT Press.

<http://papers.nips.cc/paper/1713-policy-gradient-methods-for-reinforcement-learning-with-function-approximation.pdf>.

Szegedy, Christian, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. "Going Deeper with Convolutions." In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 1–9. <https://doi.org/10.1109/CVPR.2015.7298594>.

Trichilo, D., Braun, J.L., 2005. *Extending the Fundamental Law of Investment Management*.

Vaswani, Ashish, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. "Attention Is All You Need." In *Advances in Neural Information Processing Systems 30*, edited by I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, 5998–6008. Curran Associates, Inc. <http://papers.nips.cc/paper/7181-attention-is-all-you-need.pdf>.

Vincent, Pascal, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol. 2008. "Extracting and Composing Robust Features with Denoising Autoencoders." In *Proceedings of the 25th International Conference on Machine Learning*, 1096–1103. ICML '08. Helsinki, Finland: Association for Computing Machinery. <https://doi.org/10.1145/1390156.1390294>.

Wang, Alex, Yada Pruksachatkun, Nikita Nangia, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. 2019. "SuperGLUE: A Stickier Benchmark for General-Purpose Language

Understanding Systems." *ArXiv:1905.00537 [Cs]*, May.
<http://arxiv.org/abs/1905.00537>.

Wang, Jia, Tong Sun, Benyuan Liu, Yu Cao, and Hongwei Zhu. 2019. "CLVSA: A Convolutional LSTM Based Variational Sequence-to-Sequence Model with Attention for Predicting Trends of Financial Markets." In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence*, 3705–11. Macao, China: International Joint Conferences on Artificial Intelligence Organization. <https://doi.org/10.24963/ijcai.2019/514>.

Watkins, Christopher J. C. H., and Peter Dayan. 1992. "Q-Learning." In *Machine Learning*, 279–292.

Watkins, Christopher John Cornish Hellaby. 1989. "Learning from Delayed Rewards.",
http://www.cs.rhul.ac.uk/~chrisw/new_thesis.pdf.

Werbos, P.J. 1990. "Backpropagation through Time: What It Does and How to Do It." *Proceedings of the IEEE* 78 (10): 1550–60.
<https://doi.org/10.1109/5.58337>.

Wiese, Magnus, Robert Knobloch, Ralf Korn, and Peter Kretschmer. 2019. "Quant GANs: Deep Generation of Financial Time Series." *ArXiv:1907.06673 [Cs, q-Fin, Stat]*, December.
<http://arxiv.org/abs/1907.06673>.

Williams, Ronald J, and David Zipser. 1989. "A Learning Algorithm for Continually Running Fully Recurrent Neural Networks." *Neural Computation* 1 (2): 270–80.

Wooldridge, Jeffrey M. 2002. "Econometric Analysis of Cross Section and Panel Data" MIT Press. *Cambridge*, MA 108.

Wooldridge, Jeffrey M. 2008. *Introductory Econometrics: A Modern Approach*. Cengage Learning.

Yoon, Jinsung, Daniel Jarrett, and Mihaela van der Schaar. 2019. "Time-Series Generative Adversarial Networks." In *Advances in Neural Information Processing Systems* 32, edited by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, 5508–5518. Curran Associates, Inc.
<http://papers.nips.cc/paper/8789-time-series-generative-adversarial-networks.pdf>.

Zhang, Han, Tao Xu, Hongsheng Li, Shaoting Zhang, Xiaogang Wang, Xiaolei Huang, and Dimitris Metaxas. 2017. "StackGAN: Text to Photo-Realistic Image Synthesis with Stacked Generative Adversarial Networks." *ArXiv:1612.03242 [Cs, Stat]*, August.
<http://arxiv.org/abs/1612.03242>.

Zhou, Xingyu, Zhisong Pan, Guyu Hu, Siqi Tang, and Cheng Zhao. 2018. "Stock Market Prediction on High-Frequency Data Using Generative Adversarial Nets." *Research Article: Mathematical Problems in Engineering*. Hindawi. 2018.
<https://doi.org/10.1155/2018/4907423>.

Zhu, Jun-Yan, Taesung Park, Phillip Isola, and Alexei A. Efros. 2018. "Unpaired Image-to-Image Translation Using Cycle-Consistent Adversarial Networks." *ArXiv:1703.10593 [Cs]*, November.
<http://arxiv.org/abs/1703.10593>.

Index

Symbols

1/N portfolio [132](#)

(first), high, low, and closing (last) price and volume (OHLCV) [35](#)

LSTM architecture [599](#), [600](#)

A

Accession Number (adsh) [55](#)

AdaBoost algorithm [367](#)

advantages [369](#)

disadvantages [369](#)

used, for predicting monthly price moves [369](#), [371](#)

AdaGrad [528](#)

adaptive boosting [366](#), [367](#)

adaptive learning rates

about [527](#)

AdaGrad [528](#)

adaptive moment derivation (Adam) [528](#)

RMSProp [528](#)

adaptive moment derivation (Adam) [528](#)

agglomerative clustering [435](#)

aggressive strategies [4](#)

Akaike information criterion (AIC) [183](#)

AlexNet [564](#)

AlexNet performance

comparing [566](#), [567](#)

algorithm

finding, for task [149](#)

Algorithm API [243](#)

algorithmic trading libraries

Alpha Trading Labs

Interactive Brokers

pybacktest

Python Algorithmic Trading Library (PyAlgoTrade)

QuantConnect

Trading with Python

ultrafinance

WorldQuant

AlgoSeek [41](#)

AlgoSeek intraday data

processing [43](#), [44](#), [45](#)

AlgoSeek NASDAQ 100 dataset

AllenNLP

all or none orders [23](#)

alpha [124](#)

alpha factor

from market data [110](#), [111](#), [112](#)

resources

alpha factor research [14](#)

execution phase [15](#)

research phase [15](#)

alpha factors [82](#), [83](#)

denoising, with Kalman filter [102](#), [103](#)

engineering [94](#)

engineering, NumPy used [95](#)

engineering, pandas used [95](#)

Alphalens

factor evaluation [114](#)

pyfolio input, obtaining from [142](#)

use, for backtesting long-short trading strategy

Alphalens analysis

information coefficient [394](#)

quantile spread [394](#)

Alpha Trading Labs

alternative betas [127](#)

alternative data

data providers [71](#)

email receipt data [74](#)

geolocation data [73](#)

market [70, 71](#)

satellite data [72](#)

social sentiment data [71](#)

URL [71](#)

use cases [71](#)

working with [74](#)

alternative data revolution [60](#)

alternative data revolution, technical aspects

format [70](#)

latency [69](#)

alternative datasets

evaluating, based on quality of data [67](#)

evaluating, based on quality of signal content [65](#)

evaluating, criteria [65](#)

alternative datasets, sources

about [62](#)

business processes data [63](#)

individuals data [62, 63](#)

sensors data [63](#)

alternative RNN architectures [595](#)

attention mechanism [596](#)

bidirectional RNNs [595](#)

encoder-decoder architectures [596](#)

output recurrence [595](#)

teacher forcing [595](#)

transformer architecture [596](#)

alternative trading system (ATS) [3](#)

alternative trading systems (ATSSs) [24](#)

American Depository Receipts (ADR) [24](#)

Amihud Illiquidity

analytical tools

for diagnostics and feature extraction [256](#), [257](#)

Apache HBASE

Apache Hive

Apache MXNet [546](#)

Apache Pig

Apache Spark

API access

to market data [45](#)

Applied Quantitative Research (AQR) [8](#), [10](#)

appraisal risk [124](#)

approximate inference [296](#), [301](#)

approximate split-finding algorithm [384](#)

arbitrage pricing theory [6](#)

arbitrage pricing theory (APT) [190](#)

arbitrage strategy

backtesting, based on boosting ensemble [403](#), [404](#)

area under the curve (AUC) [158](#)

ARIMA models [268](#)

AR model

relationship, with MA model [268](#)

artificial intelligence (AI) [153](#)

relation, with DL [517](#)

asset price moves

probabilities, estimating of [299](#), [301](#)

asset quality

measuring [93](#), [94](#)

assets under management (AUM) [7](#)

assumptions

updating, from empirical evidence [297](#), [298](#)

attention mechanism [511](#), [596](#)

augmented Dickey-Fuller test (ADF test) [262](#)

autocorrelation

measuring [259](#), [260](#)

autocorrelation coefficient [259](#)

autocorrelation function (ACF) [259](#)

autoencoders [152](#)

data, preparing [634](#), [636](#)

designing, with TensorFlow 2 [634](#)

for nonlinear feature extraction [627](#)

training, with TensorFlow 2 [634](#)

Automatic Differentiation Variational Inference (ADVI) [304](#)

autoregressive CNN, building with one-dimensional convolutions
[580](#)

data preprocessing [581](#), [582](#)

model architecture, defining [582](#), [583](#)

model training [583](#)

performance evaluation [583](#)

autoregressive conditional heteroskedasticity (ARCH) model [272](#)

autoregressive models

building [266](#)

model fit, diagnosing [266](#)

number of lags, identifying [266](#)

autoregressive moving-average (ARMA) [266](#)

**autoregressive moving-average model with exogenous inputs
(ARMAX) model** [269](#)

autoregressive stochastic process [262](#)

averaging method [352](#)

B

backprop [518](#)

implementing, with Python [533](#)

backpropagation through time [595](#)

backtesting

optimal stopping [229](#)

pitfalls [223](#)

pitfalls, avoiding [224](#)

right data, obtaining [224](#)

right simulation, obtaining [225](#)

right statistics, obtaining [227](#)

backtesting engine

working [229](#)

backtesting engine, key implementation aspects [232](#)

data ingestion [232](#)

factor engineering [233](#)

ML models [233](#)

performance evaluation [234](#)

predictions [233](#)

signals [233](#)

trading rules and execution [233](#)

backtest overfitting [722](#)

backtest portfolio performance

measuring, with pyfolio [141](#)

backtests [66](#)

backtrader [234](#)

data, loading [238](#)

overview [241](#)

pairs, tracking with custom DataClass [294](#)

price, loading [238](#)

strategy, backtesting [294](#)

strategy, evaluating

strategy, running

trading logic, formulating [238](#), [239](#)

using [237](#)

backtrader Cerebro architecture [234](#)

data feeds [235](#)

indicators [235](#)

lines [235](#)

Strategy object [236](#)

trades [236, 237](#)

bagged decision trees [356, 358](#)

bagging [354](#)

lower model variance [354](#)

bagging methods

pasting [354](#)

random patches [356](#)

random subspaces [356](#)

bag-of-words model

about [451](#)

text data, converting into numbers [450](#)

Baltic Dry Index (BDI) [62](#)

baseline model [175](#)

assumptions, validating [182](#)

formulating [175](#)

Gauss-Markov theorem (GMT) [179, 181](#)

learning, with MLE [177, 178](#)

learning, with OLS [176, 177](#)

learning, with SGD [178, 179](#)

problems, diagnosing [182](#)
statistical inference, conducting [181](#), [182](#)
training [176](#)

Bayesian credible interval [307](#)

Bayesian information criterion (BIC) [183](#)

Bayesian machine learning

with Theano [305](#)

working [296](#)

Bayesian ML

for trading [317](#)

Bayesian rolling regression

for pairs trading [320](#), [321](#), [322](#)

Bayesian Sharpe ratio

for performance comparison [317](#)

performance of two return series, comparing [319](#)

Bayesian statistics [296](#)

Bayesian view [296](#)

Bayes theorem [296](#), [457](#)

Beautiful Soup

used, for extracting data from HTML [74](#), [75](#)

behavioral algorithms [17](#)

Bellman equations [686](#)

best linear unbiased estimates (BLUE) [181](#)

bias [162](#)

bias-variance trade-off [150](#), [162](#)

diagnosing [171](#)

managing [163](#)

bid-ask bounce [35](#)

Bidirectional Encoder Representation from Transformers (BERT) ,
[511](#)

attention mechanism

bidirectional pretraining

unsupervised pretraining

bidirectional language models [510](#)

bidirectional RNN GRU, using with SEC filings to predict weekly return

RNN model, data preparing for , [623](#), [624](#)

SEC filling data, preprocessing [623](#)

yfinance, using to obtain stock price data [622](#)

bidirectional RNNs [596](#)

binary data formats

creating [388](#), [389](#)

binary order messages

parsing [28](#), [29](#), [30](#), [31](#)

binary sentiment classification, with Twitter data

about [459](#)

comparison, with TextBlob sentiment scores [460](#)

multinomial naive Bayes [459](#)

binary tree [330](#)

black-box models

insights, obtaining from [722](#)

Black-Litterman approach [133](#)

blind source separation [420](#)

Bollinger Band [291](#)

Bollinger Bands [100](#)

computing [292](#)

boosting method [352](#)

bootstrap aggregation [354](#)

bootstrapping [683](#), [697](#)

breakthrough of DL algorithm [517](#)

browser automation tool

Selenium [76](#)

Broyden-Fletcher-Goldfarb-Shanno (BFGS) [309](#)

bundle [242](#)

burn-in samples [303](#)

business processes data [63](#)

buy stop order [23](#)

C

C++ [388](#)

capital asset pricing model (CAPM) [5](#), [91](#), [121](#), [127](#)

migrating, to Fama-French five-factor model [189](#), [190](#)

carry strategy [6](#)

CatBoost [382](#)

versus LightGBM [393](#)

CatBoost models

signals, generating [387](#), [388](#)

causal inference [155](#)

Central Index Key (CIK) [55](#)

central limit theorem (CLT) [123](#)

Cerebro control system [237](#)

Cerebro instance

configuring [240](#)

CIFAR-10 data

preprocessing, with image augmentation [564](#)

classical linear model

normality [181](#)

classical statistics [296](#)

classification [154](#), [214](#)

classification error rate [336](#)

classification performance

comparing, with regression performance [347](#)

classification problems [157](#)

classification tree

building [336](#)

node purity, optimizing [336](#), [337](#)

training [337](#), [339](#)

classification trees [329](#), [359](#)

cluster algorithms [151](#)

clustered standard errors [184](#)

clustering

for portfolio optimization

clustering algorithms

about [432](#)

CNN architectures

evolution [558](#)

network size [558](#)

performance breakthroughs [558](#)

CNNs

for images [559](#)

for time-series data [580](#)

grid-like data, modeling [552](#), [553](#)

CNN-TA [584](#)

cointegration [282](#)

testing approaches [283](#)

cointegration approach [285](#)

cointegration tests

precomputing [290](#)

columnar storage

combinatorial algorithms [432](#)

combinatorial cross-validation [169](#)

components, TimeGAN architecture

adversarial network, generator and discriminator elements [662](#)

autoencoder, embedding and recovery components [662](#)

computational graph [594](#)

computer-to-computer interface (CTCI) [27](#)

conditional autoencoder architecture [646](#)

conditional autoencoder, for return forecasts and trading [644](#), [646](#)

architecture, creating

dataset, creating with stock price and metadata information ,
[646](#), [647](#)

predictive asset characteristics, computing

conditional GANs (cGANs)

using, for image-to-image translation [653](#)

conjugate priors [296](#), [299](#)

conservative-minus-aggressive (CMA) investment factor [85](#)

consolidated feed [26](#), [42](#)

constant proportion portfolio insurance (CPPI) [86](#)

context [593](#)

context-free models

continuous-bag-of-words (CBOW) model [486](#)

skip-gram (SG) model [487](#)

convolutional autoencoders [642](#), [643](#)

using, to compress images [629](#)

convolutional layer

convolution stage [555](#)

detector stage [557](#)

key elements, operating [554](#), [555](#)

pooling stage [557](#)

convolutional neural networks (CNNs) [627](#)

cophenetic correlation [436](#)

correlogram [260](#)

corrupted data

fixing, with denoising autoencoders [631](#)

CountVectorizer

used, for finding similar documents [454](#)

used, for visualizing vocabulary distribution [453](#)

using [453](#)

credit-assignment problem [683](#)

cross-asset relative value strategies [88](#)

cross-entropy [336](#)

cross-entropy cost function [533](#)

cross-validation

implementing, in Python [166](#)

results [393](#)

used, for selecting model [164](#), [165](#)

cross-validation, in finance

challenges [168](#)

cross-validation performance

analyzing

information coefficient, for lookahead periods

information coefficient, for lookback periods

information coefficient, for roll-forward periods

OLS regression, of random forest configuration parameters on IC

crowdsourcing trading algorithms [12](#)

CSV [58](#)

curse of dimensionality [409](#), [410](#), [411](#), [413](#)

approaches [464](#)

custom metric

GridsearchCV, using for decision trees [344](#), [345](#)

custom ML factor

designing [253](#)

custom OpenAI trading environment

DataSource class, designing [709](#), [710](#)

designing [709](#)

parameterizing

registering

TradingEnvironment class [712](#)

TradingSimulator class [711](#)

custom probability model

defining [318](#)

custom TradingCalendar

creating [246](#)

registering [246](#)

custom word embeddings, with LSTM

for sentiment classification [616](#)

custom word embeddings, with LSTM for sentiment classification

embedding, defining [617](#)

IMDB movie review data, loading [617](#)

RNN architecture, defining [617, 618](#)

custom word embeddings, with LSTM for sentiment classification

embedding, defining [618](#)

CycleGAN [654](#)

D

dark pools [3, 25](#)

data

collecting [159](#)

hierarchical features, extracting from [516, 517](#)

loading [95](#)

preparing [159](#)

reshaping [95](#)

slicing [95](#)

data-driven risk factors [644](#)

about [427](#)

data, preparing [427](#)

Dataiku

Dataminr [72](#)

data quality, aspects

about [67](#)

exclusivity [67](#)

frequency [69](#)

legal and reputational risks [67](#)

reliability [69](#)

time horizon [69](#)

Datarobot

DDQN, implementing with TensorFlow 2 [703](#)

DDQN agent, creating [703](#)

DDQN architecture, adapting to LL environment [703](#), [704](#)

experience replay, enabling [704](#), [706](#)

key hyperparameter choices [707](#)

Lunar Lander learning performance [708](#)

OpenAI environment, setting up [707](#)

transitions, memorizing [704](#), [706](#)

decision trees [329](#)

advantages [351](#)
classification tree, building [336](#)
disadvantages [351](#)
features [331](#)
GridsearchCV, using with custom metric [344](#), [345](#)
monthly stock returns [331](#)
overfittting [340](#)
practice [331](#)
predictions, evaluating [340](#)
pruning [342](#)
regression tree, building with time series data [332](#), [334](#)
regularization [340](#), [341](#), [342](#)
rules, applying [329](#), [330](#)
rules, learning [329](#), [330](#)
strengths and weaknesses [350](#)
visualizing [339](#)

decoder function [596](#)

Deep convolutional GANs (DCGANs) [653](#)

using, for unsupervised representation learning [653](#)

deep feedforward autoencoder [639](#), [641](#)

encoding, visualizing [641](#)

deep learning (DL)

hierarchical features, extracting from data [516](#)

hierarchical features, using for high-dimensional data [515](#)

libraries [538](#)

need for [514](#), [515](#)

optimizations for [525](#)

relation, with AI [517](#)

relation, with ML [517](#)

using, as representation learning [516](#)

deep NNs

regularizing [523](#)

deep NNs, regularization technique

dropout [525](#)

early stopping [523](#)

parameter norm penalties [523](#)

Deep Q-learning algorithm [699](#), [701](#)

Deep Q-learning, on stock market

DDQN agent, adapting

DDQN agent performance, benchmarking

DDQN agent, training

deep Q-network (DQN) [699](#)

Deep RL for algorithmic trading

with OpenAI Gym [698](#)

with TensorFlow 2 [698](#)

Deep RL for algorithm trading

trading agent, creating [708](#)

deep RNNs

designing [596](#), [598](#)

define-by-run [542](#)

deflated SR [229](#)

deflation [261](#)

degrees of freedom (DF) [317](#)

dendrograms [436](#), [437](#)

denoising autoencoders [643](#), [644](#)

used, for fixing corrupted data [631](#)

DenseNet201 [575](#)

density-based clustering algorithm [437](#)

density-based clusters [151](#)

density-based spatial clustering of applications with noise (DBSCAN) algorithm [437](#)

deterministic methods

versus stochastic techniques [301](#)

diagnostic tests, baseline model

conducting [182](#)

goodness-of-fit measures [183](#)

heteroskedasticity [183](#)

multicollinearity [185](#)

serial correlation [184](#)

differentiable function [552](#)

dimensionality reduction [152, 340, 409, 410](#)

direct market access (DMA) [4](#)

discriminative models [650](#)

versus generative models [651](#)

discriminator network [652](#)

distance approach [285](#)

distance-based heuristics

computing, to identify cointegrated pairs [287, 288](#)

distributed bag of words (DBOW) model [505](#)

distributed memory (DM) model [505](#)

divisive clustering [435](#)

document store

document-term matrix

about [450](#)

creating [451](#)

similarity of documents, measuring [451](#)

with sklearn [452](#)

dollar bars [40](#)

domain expertise [717](#)

domain-specific embeddings

training, for financial news [493](#)

Double DQN (DDQN) algorithm [701](#), [702](#)

DQN architecture

experience replay [701](#)

target network, changing [701](#)

dropout [525](#)

dropout for additive regression trees (DART) [386](#)

Durbin-Watson statistic diagnoses serial correlation [184](#)

dynamic programming [684](#)

Finite MDPs [684](#)

generalized policy iteration [688, 689](#)

policy iteration [687, 688](#)

value iteration [688](#)

dynamic programming (DP) [683](#)

dynamic programming, in Python [689](#)

gridworld, setting up [690](#)

MDPs, solving with pymdptoolbox [693](#)

policy iteration, defining [693](#)

policy iteration, running [693](#)

transition matrix, computing [691, 692](#)

value iteration algorithm, implementing [692](#)

E

early stopping [376, 523](#)

earnings before interest and taxes (EBIT) [94](#)

earnings calls data, topic modeling

data preprocessing [479, 480](#)

experiments, running [481, 482](#)

model evaluation [480, 481](#)

model, training [480, 481](#)

earnings call transcript

parsing , [80](#)

scraping , [80](#)

earnings per diluted share (EPS) [56](#)

earnings-per-share (EPS) [224](#)

EBITDA [90](#)

efficient frontier [125](#)

finding, in Python [128](#), [129](#), [130](#), [131](#)

efficient market hypothesis (EMH) [5](#), [127](#)

eigenportfolios [430](#), [431](#), [432](#)

elastic net regression [196](#)

electronic communication network (ECN) [3](#)

electronic communication networks (ECNs) [24](#)

Electronic Data Gathering, Analysis, and Retrieval (EDGAR) [52](#)

electronic Financial Information eXchange (FIX) [26](#)

electronic trading [3](#)

email receipt data [74](#)

embargoing [169](#)

embeddings [483](#)

embeddings evaluation

vector analogies, expressing [488, 489, 490](#)

vector arithmetic, expressing [488, 489, 490](#)

empirical evidence

assumptions, updating from [297, 298](#)

empirical prior [299](#)

encoder [596](#)

encoder-decoder architectures [596](#)

engineering of features [160](#)

Engle-Granger two-step method [283](#)

ensembled signals

long-short strategy, backtesting based on

ensemble learning, goal

accurate [352](#)

independent [352](#)

ensemble methods

averaging method [352](#)

boosting method [352](#)

ensemble models [340](#)

ensemble size [376](#)

epoch [179](#), [487](#)

equal-weighted (EW)

equity quote [41](#)

error [176](#)

error correction model (ECM) [283](#)

Euclidean distance [411](#)

event-driven backtest

versus vectorized backtest [230](#), [231](#)

exact greedy algorithm [384](#)

exact inference [298](#)

exchange [24](#)

exchange-traded funds (ETFs) [3](#)

explained variance score [156](#)

exponential GARCH (EGARCH) model [275](#)

exponential smoothing models [259](#)

extensions

extract-transform-load (ETL)

F

F1 score [159](#)

Facebook AI Research (FAIR) [542](#)

factor betas

computing [97](#)

factor establishment [84](#)

factor evaluation, with Alphalens [114](#)

factor quantiles, creating [115](#)

factor turnover

forward returns, creating [115](#)

information coefficient , [118](#), [119](#)

predictive performance, by factor quantiles [116](#), [117](#), [118](#)

factor loadings [644](#)

factors

combining, from data sources [112](#), [113](#), [114](#)

false positive rates (FPR) [158](#)

fama-French factor models [644](#)

Fama-French five-factor model

risk factors [190](#), [191](#), [192](#)

Fama-Macbeth regression [192, 193, 194](#)

fastai library [547](#)

feasible generalized least squares (GLSAR) [184](#)

feature extraction

automating [516](#)

feature importance [396](#)

capturing [350](#)

feature maps [555](#)

features

evaluating, information theory used [161](#)

Federal Reserve’s Economic Data (FRED) [306](#)

feedforward autoencoder

with sparsity constraints [639](#)

fill or kill orders [23](#)

filters

synthesizing, from data [553](#)

Financial Industry Regulatory Authority (FINRA) [26](#)

financial news

domain-specific embeddings, training for [493](#)

financial news, word embeddings

n-grams, creating [494](#), [495](#)

sentence detection [494](#), [495](#)

training with Gensim [499](#), [500](#), [501](#)

visualizing, with TensorBoard [498](#)

financial statement and notes (FSN) datasets [53](#), [54](#)

financial statement data [52](#)

financial text data

trading

financial time series

RNNs, using with TensorFlow 2 [600](#)

financial time series, clustering in two-dimensional image format [584](#)

convolutional neural network, creating [588](#)

hierarchical feature clustering [587](#), [588](#)

long-short trading strategy, backtesting

models, assembling to generate tradeable signals

relevant features, selecting based on mutual information [586](#)

rolling factor betas, computing for horizons [585](#)

technical indicators, creating at different intervals [584](#), [585](#)

Finite Markov decision problems [684](#)

actions [685](#)

Bellman equations [686](#)

long-run reward, estimating [685](#)

optimal value functions [687](#)

rewards [685](#)

sequences of states [685](#)

value functions [685, 686](#)

five Fama-French risk factors [585](#)

five Vs, big data

value [60](#)

variety [60](#)

velocity [60](#)

veracity [60](#)

volume [60](#)

FIX protocol [27](#)

forward propagation [520, 532](#)

forward returns

creating [100](#)

fundamental data [52](#)

fundamental data time series

building [53](#)

Fundamental Law of Active Management [124](#), [125](#)

fundamental value strategies [88](#)

G

GAN applications, to images and time-series data [653](#)

CycleGAN [654](#)

SRGAN [654](#)

StackGAN [654](#)

GAN architecture [652](#)

evolution [652](#)

GAN, building with TensorFlow 2 [655](#)

adversarial training process, designing [658](#)

adversarial training process, setting up [657](#)

discriminator loss functions, defining [658](#)

discriminator network, creating [656](#)

generator loss functions, defining [658](#)

generator network, building [655](#), [656](#)

results, evaluating [660](#)

training loop [659](#)

gated recurrent units (GRUs) [598](#), [600](#)

Gaussian mixture model (GMM) algorithm , [433](#), [438](#)

Gaussian mixture models [151](#)

Gaussian white noise [257](#)

Gauss-Markov theorem (GMT) [179](#), [181](#)

assumptions [179](#)

GBM models

training [374](#), [376](#)

tuning [374](#), [376](#)

GBM results

interpreting [395](#)

Gelman-Rubin statistic [314](#)

General Data Protection Regulation (GDPR) [67](#)

generalized autoregressive conditional heteroskedasticity (GARCH) model [272](#), [274](#)

generalized least squares (GLS) [184](#)

generalized linear models (GLM) [173](#)

Generalized Linear Models (GLM) [309](#)

generalized policy iteration [688](#), [689](#)

General Language Understanding Evaluation (GLUE)

Generally Accepted Accounting Principles (GAAP) [52](#)

generative adversarial networks (GANs) [19](#)

training [651](#), [652](#)

using, for synthetic data [650](#)

generative adversarial what-where network (GAWWN) [653](#)

generative models [650](#)

versus discriminative models [651](#)

generator network [651](#)

Gensim

used, for implementing LDA [477](#), [478](#), [479](#)

used, for training word embeddings of financial news [499](#), [500](#), [501](#)

geolocation data [73](#)

Gibbs sampling [303](#)

Gini impurity [336](#)

global minimum-variance (GMV) portfolio [132](#)

Global Portfolio Optimization [133](#)

global vectors (GloVe) [619](#)

Global vectors (GloVe)

using, for word representation [490](#), [492](#)

goodness-of-fit measures [183](#)

GoogLeNet [568](#)

GPU acceleration

leveraging [538](#), [539](#)

gradient-based One-Side sampling (GOSS) [384](#)

gradient boosting [367](#), [372](#), [373](#), [374](#)

for high-frequency strategy

long-short trading strategy [387](#)

using, with sklearn [377](#), [378](#)

gradient boosting machines (GBMs) [373](#)

gradient boosting model predictions

ensembling

gradients

computing [533](#), [534](#)

testing [536](#)

Granger causality [278](#)

graph database

graphics processing units (GPUs) [538](#)

greedy approach [330](#)

greedy policy [697](#)

GridsearchCV

using, with custom metric for decision trees [344](#), [345](#)

GridSearchCV

for parameter tuning [171](#)

parameters, tuning [379](#)

H

H2O.ai

Hadoop ecosystem

Hadoop ecosystem, tools

Apache HBASE

Apache Hive

Apache Pig

half-life of mean reversion

estimating [292](#)

Hamiltonian Monte Carlo (HMC) [304](#)

handwritten digit classification [560](#), [562](#)

HDF5 [58](#)

heads [511](#)

heterogeneous ARCH (HARCH) model [275](#)

heterogeneous autoregressive processes (HAR) [275](#)

heteroskedasticity [168](#), [183](#)

heuristics

significant cointegration, predicting [288](#), [290](#)

hidden layer gradients [535](#)

hierarchical Bayesian model [472](#)

hierarchical clustering algorithm

about [432](#), [435](#)

dendograms [436](#), [437](#)

drawbacks [437](#)

strengths [437](#)

hierarchical clustering algorithm, approaches

agglomerative clustering [435](#)

divisive clustering [435](#)

hierarchical clustering portfolios (HCP) [137](#)

hierarchical clusters [151](#)

Hierarchical DBSCAN (HDBSCAN) algorithm [438](#)

hierarchical risk parity (HRP)

about

backtesting, with ML trading strategy

working

Hierarchical risk parity (HRP) [136](#), [137](#)

hierarchical softmax [487](#)

high-dimensional data

hierarchical features, using for [515](#)

higher-order features [554](#)

highest posterior density (HPD) [313](#)

high-frequency market data [26](#)

high-frequency trading (HFT) [4](#)

engineering features

high-minus-low (HML) value factor [85](#)

holdout set

testing on [382](#)

HRP performance

HRP weights

computing, with PyPortfolioOpt

HTML

data, extracting with BeautifulSoup [74, 75](#)

data, extracting with requests [74, 75](#)

HTML tables

reading [46](#)

Hugging Face Transformers library

hyperparameters

tuning [390](#)

hyperparameter tuning [342](#)

decision trees, strengths and weaknesses [350](#)

feature importance, capturing [350](#)

GridsearchCV, using with custom metric for decision trees [344](#)

training set size, diagnosing with learning curve [348](#)

tree structure, inspecting [345, 347](#)

Hypertext Transfer Protocol (HTTP) [74](#)

hypothesis space [149](#)

I

idiosyncratic volatility

illiquidity premium [6](#)

image augmentation

CIFAR-10 data, preprocessing [564](#)

image classification [559](#)

ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [558](#)

images

compressing, with convolutional autoencoders [629](#)

immediate or cancel orders [23](#)

impulse-response function [278](#)

Inception module [568](#)

independent and identically-distributed (IID) [257](#)

independent component analysis (ICA) algorithm [421](#)

about [420](#)

assumptions [421](#)

with sklearn [421](#)

independently and identically distributed (IID) [123](#), [165](#)

individuals data [62](#), [63](#)

inertia [433](#)

inference

versus prediction [154](#)

information coefficient (IC) [119](#), [124](#), [204](#), [390](#), [605](#)

information ratio (IR) [123](#)

information theory [18](#)

used, for evaluating features [161](#)

initial public offerings (IPOs) [32](#)

Instrumented Principal Component Analysis (IPCA) [646](#)

Interactive Brokers

interactive development environment (IDE) [49](#)

Internet of Things (IoT) [64](#)

inverse document frequency (IDF) [452](#)

inverted yield curve [307](#)

investment industry

algorithmic pioneers, using [7](#), [8](#)

alternative data, using [10](#), [11](#)

crowdsourcing trading algorithms, using [11](#)

high-frequency trading, using [3](#), [4](#)

ML-driven funds [8](#), [9](#)

ML-driven strategies, designing for [12, 14](#)

ML-driven strategies, executing for [12, 14](#)

ML, using [2, 3, 10, 11](#)

quantamental funds [9](#)

risk factors, investing [5, 6](#)

smart beta funds [7](#)

strategic capabilities, investments [9](#)

investment industry, ML-driven strategies

alpha factor research [14](#)

backtesting [16](#)

data, managing [14](#)

data, sourcing [14](#)

J

Japanese equities

features

outcomes

Johansen likelihood-ratio test [283](#)

K

Kalman filter

alpha factors, denoising [102](#), [103](#)
applying, pykalman used [105](#)
working [103](#), [104](#)

Kalman filter (KF)

prices, smoothing [291](#)
rolling hedge ratio, computing [291](#)

Kelly criterion [133](#)

Keras Functional API [608](#)

kernel [555](#)

key elements, RL systems

model-free agents, versus model-based agents [682](#)
policy [681](#)
reward signal [681](#)
value function [682](#)

key hyperparameters

tuning [529](#), [530](#)

key return drivers

identifying, with PCA execution [428](#), [429](#), [430](#)

key-value storage [723](#)

KFold iterator [167](#)

k-means clustering

about [433](#)

observations, assigning [433](#), [434](#)

quality, evaluating [434](#), [435](#)

K-means clustering [151](#)

k-means objective function [434](#)

k-nearest neighbors (KNN) [154](#)

L

L1 regularization [631](#)

labeling [444](#)

lagged return features

creating [98](#)

lagged returns

using [97](#)

Lagrange multiplier (LM) test [182](#)

language features

engineering [456](#)

lasso regression

sklearn, using [212](#)

working [198](#)

lasso regression, with sklearn [212](#)

IC and lasso path, evaluating [213](#)

lasso model, cross-validating [212](#)

Latent Dirichlet allocation (LDA) [472](#)

Dirichlet distribution [473](#)

generative model [473](#)

generative process, reverse engineering [474](#)

implementing, with Gensim [477](#), [478](#), [479](#)

implementing, with sklearn [476](#)

working [472](#)

latent semantic analysis (LSA) [466](#)

latent semantic indexing (LSI) [466](#)

implementing, with sklearn [467](#), [468](#), [469](#)

limitations [470](#)

strengths [469](#)

latent space [596](#)

LDA results

visualizing, with pyLDAvis [476](#)

LDA topics

evaluating [474](#)

LDA topics, evaluating options

perplexity [475](#)

topic coherence [475](#)

learning curve [163](#), [171](#), [348](#)

training set size, diagnosing with [348](#)

learning parameters [390](#)

learning rate [376](#)

leave-one-out method [167](#)

leave-P-out CV [168](#)

LeNet5 [560](#)

LeNet5 architecture

defining [562](#), [563](#)

model, evaluating [563](#)

model, training [563](#)

LightGBM [382](#)

minute-frequency signals, generating

versus CatBoost [393](#)

LightGBM documentation

reference link

LightGBM models

best-performing parameter settings [393](#), [394](#)

hyperparameter impact [394](#)

signals, generating [387](#), [388](#)

LightGBM Random Forest model

ML4T workflow, using

limit order [23](#)

linear classification [214](#)

inference, conducting with statsmodels [217](#), [218](#), [219](#)

with logistic regression model [215](#)

linear classification, with logistic regression model [215](#)

logistic function [216](#)

maximum likelihood estimation [216](#), [217](#)

objective function [215](#)

linear dimensionality reduction algorithm [410](#)

linear dimensionality reduction algorithms [413](#)

linear factor model

building [188](#)

linear models [329](#)

linear OLS regression

with statsmodels [205](#)

linear OLS regression, with statsmodels [205](#)

diagnostic statistics [206](#)

relevant universe, selecting [205](#)

vanilla OLS regression, estimating [206](#)

linear regression

scikit-learn, using [207](#)

used, for predicting stock returns [199](#)

linear regression models [174](#)

implementing [185](#)

linear regression, with scikit-learn

cross-validating [207](#), [208](#)

features and targets, selecting [207](#)

information coefficient, evaluating [208](#), [209](#)

RMSE [208](#), [209](#)

linguistic annotation [443](#)

linguistic annotation, concepts

dependency parsing [443](#)

lemmatization [443](#)

POS [443](#)

stemming [443](#)

liquidity detection [4](#)

Ljung-Box Q-statistic [267](#)

local features

extracting [555](#)

locally linear embedding (LLE) [423](#)

logarithm [261](#)

logistic regression

AUC and IC results, evaluating

converting, into classification problem [219](#)

hyperparameters, cross-validating

used, for predicting price movements [219](#)

log-likelihood function [177](#)

log-likelihood ratio (LLR) [218](#)

log-odds [216](#)

long-range dependencies

learning, challenges [598](#)

long-short signals

for Japanese stocks, with random forest

long-short strategy, backtesting based on ensembled signals

about

predictions, ensembling to produce tradeable signals

long short-term memory (LSTM) [598](#), [633](#)

long-short trading strategy

backtesting, with Alphalens

backtesting, with Zipline

cross-validating design options, for tuning NN , [549](#)

engineering features, to predict daily stock returns [547](#)

neural networks (NNs), optimizing for [547](#)

NN architecture framework, defining [547](#), [549](#)

predictive performance, evaluating

with gradient boosting [387](#)

look-ahead bias [224](#)

lookahead bias [165](#)

loss function [390](#)

loss function gradient [534](#), [535](#)

lower hedge fund fees [3](#)

LSTM cell state

forget gate [600](#)

input gate [600](#)

output gate [600](#)

LSTM unit

information flow [599](#)

Lunar Lander v2 (LL) environment [702](#)

M

machine learning [148](#)

workflow [153](#)

machine learning (ML) [409](#)

about [514](#), [719](#)

bias-variance trade-off, managing [720](#)

linear models [720](#)

model diagnostics, to speed up optimization [719](#), [720](#)

nonlinear models [720](#)

optimization verification test [722](#)

relation, with DL [517](#)

targeted model objectives, defining [722](#)

using, for investment industry [2](#), [3](#)

with text data [441](#)

workflow [719](#)

machine learning model

ensembling

hyperparameters

lookahead period

lookback period

test period

machine learning signal

backtesting [247](#)

manifold [409](#)

manifold hypothesis [421](#)

manifold learning [152](#)

market beta

market data [22](#)

regularizing [35](#)

market-data providers [51, 52](#)

market makers [24](#)

market microstructure [23, 86](#)

market order [23](#)

market portfolio [127](#)

market value strategies [88](#)

Markov chain [302](#)

Markov chain Monte Carlo (MCMC) [301](#)

Markov chain Monte Carlo sampling

using, for stochastic inference [302](#)

Markov decision process (MDP) [684](#)

Markowitz curse [136](#)

mark-to-market performance [225](#)

material non-public information (MNPI) [67](#)

maximum a posteriori probability (MAP) [298](#)

maximum likelihood estimation (MLE)

used, for learning baseline model [177](#), [178](#)

Maximum Likelihood Estimation (MLE) [298](#)

max pooling [557](#)

mean of the absolute errors (MAE) [156](#)

mean-squared error (MSE) [334](#)

mean-variance (MV)

mean-variance optimization

about [128](#)

implementing [139](#), [140](#), [141](#)

working [128](#)

mean-variance optimization, alternatives

1/N portfolio [132](#)

Global Portfolio Optimization [133](#)

Kelly criterion [133](#)

minimum-variance portfolio [132](#)

median of the absolute errors (MedAE) [156](#)

Metropolis-Hastings sampling [303](#)

Microsoft Cognitive Toolkit (CNTK) [546](#)

minimum backtest length [229](#)

minute bars [41](#), [42](#)

minute-frequency signals

generating, with LightGBM

ML4T workflow

cross-validating signals, over horizons

cross-validation performance, analyzing

ensembling forecasts
experimental design
hyperparameter tuning
lookahead, defining
lookback, defining
roll-forward periods, defining
universe selection
with LightGBM Random Forest model

ML algorithm

selecting [161](#)

ML-driven strategy

backtesting [222](#)

ML for algorithmic trading

quantitative strategies, evolution [16, 17](#)

use cases [17](#)

ML for trading

data [715](#)

data integration [715](#)

domain expertise [717](#)

in practice [723](#)

key elements [714](#)

ML tools

online trading platforms

quality control, for intermediate data sources [715](#)

quality control, for raw sources [715](#)

ML for trading, data management technologies

big data technologies

database systems [723](#)

model

designing [161](#)

selecting, cross-validation used [164](#), [165](#)

training, during backtest [252](#)

tuning [161](#)

model features and forward returns preparation [199](#)

alpha factors, computing with TA-Lib [202](#), [203](#)

alpha factors, selecting with TA-Lib [201](#), [202](#)

dummy encoding, of categorical variables [204](#), [205](#)

investment universe, creating [199](#), [200](#), [201](#)

lagged returns, adding [203](#)

target forward returns, generating [204](#)

model selection problem [164](#)

model transparency [18](#)

modern portfolio theory (MPT) [5](#), [125](#), [127](#)

challenges [131](#)

shortcomings [131](#)

modern portfolio theory (MPT), approaches

mean-variance optimization [128](#)

momentum [526](#)

excess returns, driving [85](#)

measuring [86](#)

momentum effect [6](#)

momentum factors [98](#)

momentum investing [84](#), [85](#)

momentum updates

implementing, with Python [536](#)

monotonicity constraints [387](#)

Monte Carlo (MC) method [683](#)

Monte Carlo methods [302](#)

Montreal Institute for Learning Algorithms (MILA) [305](#)

moving average convergence/divergence (MACD)

moving-average models [259](#)

building [267](#)

number of lags, identifying [267](#)

moving averages [259](#)

multiclass sentiment analysis, Yelp business reviews

about [460, 461](#)

benchmark accuracy [461](#)

LightGBM gradient boosting tree, training [462](#)

logistic regression [462](#)

multinomial naive Bayes model, training [461](#)

predictive performance

text and numerical features, combining [461](#)

multicollinearity

challenges [185](#)

multilabel problems [150](#)

multilayer perceptron (MLP) [520](#)

multiple testing bias [165](#)

multivariate time-series models [277](#)

systems of equations [278](#)

vector autoregressive (VAR) model [278, 279](#)

multivariate time-series regression, for macro data [612](#)

data stationary, making [612](#)
model, defining [615](#), [616](#)
model, training [615](#), [616](#)
multivariate RNN inputs, creating [614](#)
scale, adjusting [612](#)
sentiment and industrial production data, loading from Fed [612](#)

mutual information (MI) [161](#)

N

naive Bayes classifier [457](#)

conditional independence assumption [457](#), [458](#)

naive Bayes model

used, for classifying news articles [458](#), [459](#)

named-entity recognition (NER)

labeling [444](#)

NASDAQ order book data [26](#)

NASDAQ TotalView-ITCH data feed [27](#)

National Best Bid and Offer (NBBO) [25](#), [42](#)

National Bureau of Economic Research (NBER) [306](#)

National Financial Conditions Index (NFCI) [306](#)

natural language generation

transforming [511](#)

natural language processing (NLP) [152](#)

used, for trading [456](#)

with TextBlob [449](#)

natural language processing (NLP) library

Gensim [499](#)

Natural Language Toolkit (NLTK) [449](#)

negative sampling (NEG) [487](#)

nesterov momentum [526](#)

Net Order Imbalance Indicator (NOII) [28](#)

network

training [536](#), [537](#), [538](#)

network-in-network concept [568](#)

neural language models [486](#)

neural networks (NNs) [552](#)

architecture [520](#), [521](#)

building, in Python [530](#)

cost functions [522](#)

designing [518](#)
optimizing, for long-short trading strategy [547](#)
output units [522](#)
training, in Python [530](#)
using, for value function approximation [699](#)

neural networks (NNs), key design choices

about [521](#)
activation functions [522](#)
hidden units [522](#)

New York Stock Exchange (NYSE) [24](#)

n-grams [443](#)
creating, for financial news [494](#), [495](#)

NLP pipeline

constructing [444](#)
constructing, with spaCy [445](#)
constructing, with textacy [445](#)

NLP pipeline construction

documents, batch-processing [446](#)
multi-language NLP [448](#), [449](#)
named entity recognition [447](#)
n-grams [448](#)

sentence, annotating [445](#), [446](#)
sentence boundary detection [447](#)
sentence, parsing [445](#), [446](#)
sentence, tokenizing [445](#), [446](#)
spaCy’s streaming API [448](#)

NLP workflow

about [442](#)
labeling [444](#)
linguistic annotation [443](#)
semantic annotation [444](#)
text data, parsing [443](#)
text data, tokenizing [443](#)

node purity [336](#)

optimizing [336](#), [337](#)

no-free-lunch theorem [149](#)

noise contrastive estimation (NCE) [487](#)

noisy signals

preprocessing, with wavelets [106](#), [107](#), [108](#)

non-diversifiable risk [127](#)

nonlinear activation functions [629](#)

nonlinear dimensionality reduction

PCA, generalizing [627](#), [629](#)

nonlinear dimensionality reduction algorithm [410](#)

about [421](#), [423](#)

nonlinear feature extraction

autoencoders [627](#)

non-observable [644](#)

non-traditional sources of risk premiums [127](#)

Normalized Average True Range (NATR) [585](#)

not-held orders [23](#)

No U-Turn Sampler (NUTS) [304](#)

numerical evaluations [160](#)

NumPy

alpha factors, engineering [95](#)

O

object detection [559](#), [576](#)

object detection, Google's Street View House Numbers

custom loss function, creating [578](#)

evaluation metrics [578](#)

source images, preprocessing [576](#)

transfer learning, with custom final layer [577](#)

two-step training [579](#)

objective function [390](#)

objective priors [299](#)

object segmentation [576](#)

odds [216](#)

OHLCV bundles

custom bundle ingest function, writing [245](#)

data, obtaining to be bundled [245](#)

loading, with minute data [244](#)

registering [245](#)

OLS estimates correction ways, heteroskedasticity

clustered standard errors [184](#)

robust standard errors [184](#)

one-layer feedforward autoencoder [636](#), [637](#)

decoder, defining [638](#)

encoder, defining [637](#)

model, training [638](#)

results, evaluating [638](#)

online learning [526](#)

online trading platforms, ML for trading

QuantConnect

Quantopian

QuantRocket

on state-of-the-art architectures

building [568](#)

OpenAI Gym

about [702](#)

open/close orders [23](#)

OpenTable data

restaurant bookings and ratings dataset, building [76](#), [78](#), [79](#)

scraping [74](#)

optimal size of bet [133](#), [134](#)

optimal value functions [687](#)

order book

reconstructing [32](#), [33](#), [34](#)

orders [23](#)

ordinary least squares (OLS)

used, for learning baseline model [176, 177](#)
using, with with statsmodels [187](#)

outlier control [225](#)

out-of-bag (OOB) [364](#)

testing [362, 364](#)

output recurrence [595](#)

overfitting [150, 162](#)

addressing, with regularized autoencoders [631](#)

controlling, with regularization [195, 196](#)

over-the-counter (OTC) markets [24](#)

P

padding [556](#)

pairs trading [284](#)

in practice [286](#)

pandas

alpha factors, engineering [95](#)

pandas-datareader library [57](#)

pandas library

datareader, for market data [46, 47](#)

data storage [58](#)

remote data access [45](#)

paper trading [49](#)

parameter norm penalties [523](#)

parameter tuning

with GridSearchCV [171](#)

with pipeline [171](#)

with scikit-learn [170](#)

with Yellowbrick [170](#)

Parquet [58](#)

partial autocorrelation [260](#)

partial autocorrelation function (PACF) [260](#)

partial dependence plots [397, 398, 399](#)

passive strategies [4](#)

performance gains, obtaining from algorithmic innovations [383](#)

additional features, and optimizations [386, 387](#)

depth-wise, versus leaf-wise growth [385](#)

dropout for additive regression trees (DART) [385, 386](#)

GPU-based training [385](#)

second-order loss function approximation [383](#), [384](#)

simplified split-finding algorithms [384](#)

treatment, of categorical features [386](#)

perplexity

used, for evaluating LDA topics [475](#)

personally identifiable information (PII) [67](#)

pipeline

creating, with custom ML factor [249](#), [250](#), [251](#)

for parameter tuning [171](#)

Pipeline API [247](#)

DataFrameLoader, enabling [248](#)

pipeline factors

defining [252](#)

plain-vanilla denoising [36](#), [39](#)

plate notation [470](#)

point72 [9](#)

pointwise mutual information (PMI) [475](#)

policy gradient methods [682](#)

policy iteration [687](#)

polysemy [510](#)

portfolio benchmark inputs

generating [142](#)

portfolio management

with Zipline [137](#)

portfolio performance

measuring [122](#)

testing [143](#)

portfolio position data

generating [142](#)

portfolio return

managing [125](#)

portfolio returns

generating [142](#)

portfolio risk

managing [125](#)

posterior predictive checks (PPCs) [314](#)

posterior probability distribution [297](#), [298](#)

precision [159](#)

precision-recall curves [159](#)

prediction

versus inference [154](#)

predictions

evaluating, during backtest , [254](#)

generating [315](#), [316](#)

predictive modeling

outcomes, assigning [444](#)

predictive signals

quality, comparing [214](#)

pretrained RoBERTa model

pretrained state-of-the-art models

AllenNLP

Hugging Face Transformers library

using

pretrained word vectors

used, for sentiment analysis [619](#)

using [490](#)

price/earnings time series

building [56](#)

price/earnings to growth (PEG) ratio [90](#)

price formation [24](#)

price-to-earnings (P/E) ratio [90](#)

price-to-earnings (P/E) valuation ratio [56](#)

principal component analysis (PCA) [152](#)

generalizing, with nonlinear dimensionality reduction [627](#), [629](#)

principal component analysis (PCA) algorithm

about [413](#)

based on covariance matrix [417](#), [418](#)

key assumptions [415](#)

running, to identify key return drivers [428](#), [429](#), [430](#)

using, for algorithmic trading [427](#)

visualizing, in 2D [414](#)

with sklearn [419](#), [420](#)

with SVD algorithm [418](#), [419](#)

working [415](#)

principal components analysis (PCA) [673](#)

principal diagnostic tool [530](#)

priors

selecting [299](#)

probabilistic latent semantic analysis (pLSA) [470](#)

implementing, with sklearn [471](#)

limitations [472](#)

strengths [472](#)

probabilistic modeling [432](#)

probabilistic programming

with PyMC3 [305](#)

probabilities

estimating, of asset price moves [299](#), [301](#)

probability distribution [298](#)

proprietary products [26](#)

pseudo-R2 statistic [218](#)

Public Dissemination Service (PDS) [53](#)

purging [169](#)

p-value [181](#)

pybacktest

pyfolio [234](#)

drawdown periods [145](#), [146](#)
factor exposure [145](#), [146](#)
used, for measuring backtest portfolio performance [141](#)

pyfolio event risk analysis , [146](#)

pyfolio input

obtaining, from Alphalens [142](#)
obtaining, from Zipline backtest [142](#)

pyfolio summary performance statistics [144](#), [145](#)

pykalman

used, for applying Kalman filter [105](#)

pyLDAvis

used, for visualizing LDA results [476](#)

PyMC3 [305](#)

PyMC3 workflow, recession prediction [305](#)
approximate inference, MCMC [309](#), [311](#)
approximate inference, variational Bayes [312](#)
convergence [312](#), [313](#), [314](#)
data [306](#)
exact MAP Inference [309](#)
model definition [307](#)

model diagnostic [312](#)

PyPortfolioOpt

using, to compute HRP weights

Python

cross-validation, implementing [166](#)

dynamic programming [689](#)

efficient frontier, finding [128](#), [129](#), [130](#), [131](#)

neural networks (NNs), building [530](#)

neural networks (NNs), training [530](#)

used, for implementing backprop [533](#)

used, for implementing momentum updates [536](#)

used, for training Q-learning agent [697](#), [698](#)

Python Algorithmic Trading Library (PyAlgoTrade)

PyTorch 1.4

model predictions, evaluating [546](#)

model training [545](#), [546](#)

NN architecture, defining [543](#), [545](#)

using [542](#)

PyTorch DataLoader

creating [543](#)

Q

Q-learning agent

training, with Python [697](#), [698](#)

Q-learning algorithm [697](#)

greedy policy [697](#)

optimal policy, finding [695](#)

quality factors

for quantitative investing [92](#), [93](#)

Quandl [51](#)

QuantConnect

quantile sketch algorithm [384](#)

Quantopian [49](#)

production-ready backtesting [241](#)

research environment, using on

Quantopian factors [112](#)

QuantRocket

quarterly Apple filings

retrieving [55](#)

quote data

fields [43](#)

R

R2 score [156](#)

random forest [352](#)

advantages [364](#)

bootstrap aggregation [354](#)

building [359](#)

disadvantages [364](#)

ensemble models, performance [352](#)

feature importance [362](#)

long-short signals, for Japanese stocks

out-of-bag (OOB), testing [362](#), [364](#)

training [360](#), [361](#)

tuning [360](#), [361](#)

randomized grid search [391](#), [392](#)

random walk [262](#)

ranking problem [154](#)

RavenPack [72](#)

recall [159](#)

receiver operating characteristics (ROC) curve [158](#)

receptive field [554](#)

rectified linear unit (ReLU) [522](#), [557](#)

rectified linear units (ReLU) [629](#)

recurrent conditional GANs (RCGANs)

with synthetic time series [654](#)

recurrent neural networks (RNN) [633](#)

recurrent neural networks (RNNs) [511](#), [520](#), [654](#)

applying, to text data for detecting return prediction [616](#)

applying, to text data for detecting sentiment analysis [616](#)

backpropagation through time [594](#)

computational graph [594](#)

long-range dependencies, learning challenges [598](#)

using, for financial time series with TensorFlow 2 [600](#)

working [592](#), [593](#)

recursive binary splitting [330](#)

regression performance

comparing, with classification performance [347](#)

regression problem [154](#)

regression problems [156](#), [157](#)

regression tree

building, with time series data [332](#)

regression trees [329](#), [359](#)

regularization [195](#), [391](#)

controlling, with regularization [196](#)

regularized autoencoders

used, for addressing overfitting [631](#)

regulated exchanges [24](#)

reinforcement learning (RL) [152](#)

reinforcement learning (RL) algorithms [651](#)

relational database management systems (RDBMSes) [723](#)

relative strength index (RSI) [331](#), [585](#)

relative value strategies [88](#)

remote data access

with pandas library [45](#)

Renaissance Technologies (RenTec) [124](#)

requests

used, for extracting data from HTML [74](#), [75](#)

resampling [96](#)

research environment

using, on Quantopian

residual [176](#)

residual network (ResNet) [568](#), [569](#)

residual sum of squares (RSS) [176](#)

resilient distributed data (RDD)

returns

computing, for multiple historical periods [96](#), [97](#)

reward signal [681](#)

ridge regression [196](#)

scikit-learn, using [210](#)

working [196](#), [198](#)

ridge regression, with scikit-learn [210](#)

cross-validation results [211](#)

regularization parameters, tuning with cross-validation [210](#), [211](#)

ridge coefficient paths [211](#)

top 10 coefficients [212](#)

riding the yield curve [6](#)

risk-factor exposure [2](#)

risk factor investment [136](#)

risk factors as latent [644](#)

risk parity [135](#), [136](#)

risk-return trade-offs

capturing, in single number [122](#)

RL problems

solving [682](#)

RL problems, solving fundamental approaches

dynamic programming (DP) methods [683](#)

Monte Carlo (MC) methods [683](#)

temporal difference (TD) learning algorithm [683](#)

RL problems, solving key challenges

exploration, versus exploitation [683](#)

RL problems, solving key challenges

credit assignment [683](#)

RL systems

key elements [680](#)

RMSProp [528](#)

RNN input tensor

dimensions [601](#)

robust estimation methods [173](#)

robust-minus-weak (RMW) profitability factor [85](#)

robust simulations

calendars and Pipeline API, exchanging [242](#)

robust standard errors [184](#)

rolling window statistics [259](#)

roll return [6](#)

root-mean-square error (RMSE) [156](#)

root-mean-square of the log of the error (RMSLE) [156](#)

S

sample period [225](#)

sandwich estimator [184](#)

SARIMAX [270](#)

satellite data [72](#)

scikit-learn

parameter tuning [170](#)

time series cross-validation [168](#)

Scrapy

using [79](#), [80](#)

SEC filings

labeling, with stock returns [502](#), [503](#)

using, with bidirectional RNN GRU to predict weekly returns [622](#)

SEC filings, word embeddings [501](#)

automatic phrase detection [502](#)

content selection [502](#)

model evaluation [504](#)

model training [503](#)

n-grams, creating [502](#)

parameter settings, performance impact [504](#)

sentence detection [502](#)

Securities Information Processor (SIP) [42](#)

Selenium

using [76](#)

self-supervised learning [627](#)

semantic annotation [444](#)

semantic segmentation [576](#)

semi-supervised pretraining [510](#)

sensors data

about [64](#)

geolocation data [64](#)

satellites images [64](#)

sentiment

excess returns, driving [85](#)

measuring [86](#)

sentiment analysis

about [456](#), [459](#)

Twitter data, used for binary sentiment classification [459](#)

Yelp business reviews, used for multiclass sentiment analysis
[460](#), [461](#)

sentiment analysis, with doc2vec embeddings [505](#)

doc2vec input, creating from Yelp sentiment data [505](#), [507](#)

doc2vec model, training [507](#)

sentiment classifier, training with document vectors [508](#), [509](#)

sentiment analysis, with pretrained word vectors

architecture, defining with frozen weights [620](#)

pretrained GloVe embeddings, loading [619](#)

text data, processing [619](#)

seq2seq autoencoders

used, for extracting time-series features [633](#)

sequence-to-sequence models, types

many-to-many [593](#)

many-to-one [593](#)

one-to-many [593](#)

sequence-to-sequence (seq2seq) [596](#)

serial correlation [184, 259](#)

SHapley Additive explanations

feature interaction, analyzing [402, 403](#)

plots, forcing to explain predictions [401](#)

SHapley Additive exPlanations [399](#)

SHapley Additive exPlanations (SHAP) [18, 723](#)

SHAP values

summarizing, by feature [400](#)

Sharpe ratio (SR) [123, 224](#)

shrinkage methods [173](#)

interpretation, improving [195](#)
prediction accuracy, improving [195](#)
regularizing, for linear regression [195](#)

shrinkage techniques [376](#)

ShuffleSplit class [168](#)

signal content evaluation

alpha content and quality [66](#)
asset classes [66](#)
investment style [66](#)
risk premiums [66](#)

signal generation

scheduling [138](#), [139](#)

silhouette coefficient [434](#)

simple moving average (SMA) [100](#)

single-layer architecture

about [530](#)
cross-entropy cost function [533](#)
forward propagation [532](#)
hidden layer [531](#)
input layer [530](#), [531](#)

output layer [532](#)

singular value decomposition (SVD) [466](#)

singular value decomposition (SVD) algorithm [415](#)

PCA, using [418](#), [419](#)

size

measuring [92](#)

returns, predicting [91](#)

size anomaly [91](#)

skip-gram architecture, implementing in TensorFlow 2 [495](#), [496](#)

noise-contrastive estimation (NCE) [496](#)

target-context word pairs, generating [497](#)

validation samples, creating [496](#)

word2vec model layers, creating [497](#), [498](#)

skip-gram (SG) model [486](#)

versus continuous-bag-of-words (CBOW) model [487](#)

sklearn

gradient boosting, using [377](#), [378](#)

stochastic gradient descent (SGD), using with [187](#), [188](#)

used, for implementing LDA [476](#)

used, for implementing LSI [467](#), [468](#), [469](#)

used, for implementing pLSA [471](#)
with document-term matrix [452](#)
with ICA algorithm [421](#)
with PCA algorithm [419](#), [420](#)

smart beta funds [7](#)

social sentiment data

about [71](#)
Dataminr [72](#)
RavenPack [72](#)
StockTwits [72](#)

softplus function [557](#)

spaCy

used, for constructing NLP pipeline [445](#)

specific risk factors [644](#)

Splash

using [79](#), [80](#)

spread

computing [292](#)

SRGAN [654](#)

stacked LSTM, for predicting weekly stock price moves and returns [606](#), [607](#)

architecture, defining with Keras Functional API [608](#), [609](#), [610](#), [611](#)

multiple inputs, creating in RNN format [607](#)

returns, predicting instead of directional price moves [611](#)

StackGAN [654](#)

standard error [182](#)

Standard Industrial Classification (SIC) [53](#)

state-value function [685](#)

stationarity

achieving [260](#)

diagnosing [260](#)

stationary time series [260](#)

statistical arbitrage (StatArb) [88](#)

statsmodels

used, for conducting inference [217](#), [218](#)

stochastic control approach [285](#)

stochastic gradient boosting [377](#)

stochastic gradient descent (SGD) [253](#), [526](#), [701](#)

used, for learning baseline model [178](#), [179](#)
using, with sklearn [187](#), [188](#)

stochastic inference

with Markov chain Monte Carlo sampling [302](#)

stochastic techniques

versus deterministic methods [301](#)

stochastic trends

handling [261](#)

stochastic volatility models [322](#), [323](#), [324](#)

stock momentum

stock return prediction, with linear regression [199](#)

model features and forward returns, preparing [199](#)

StockTwits [72](#)

stop order [23](#)

strategy backtest

preparing [290](#)

Street View House Numbers (SVHN) dataset [576](#)

strict stationarity [260](#)

strides [556](#)

subjective priors [299](#)

subsampling [377](#)

supervised learning [149](#)

supervised learning problem, types

binary classification [522](#)

multiclass problems [522](#)

regression problems [522](#)

survivorship bias [224](#)

synthetic data

GANs, using for [650](#)

synthetic time series

with RCGANs [654](#)

T

tag-based FIXML [27](#)

TA-Lib

technical alpha factors, creating [100](#), [102](#)

TA-Lib, technical indicators

Bollinger Bands

normalized average true range (NATR)

percentage price oscillator (PPO)

relative strength index (RSI)

t-distributed Stochastic Neighbor Embedding (t-SNE) [641](#)

t-distributed Stochastic Neighbor Embedding (t-SNE) algorithm

[410](#), [423](#), [425](#)

teacher forcing [595](#)

temporal difference (TD) learning algorithm [683](#), [697](#)

TensorBoard

used, for visualizing word embeddings of financial news [498](#)

using [541](#), [542](#)

TensorFlow 2

skip-gram architecture, implementing [495](#), [496](#)

used, for designing autoencoders [634](#)

used, for implementing TimeGAN [663](#)

used, for training autoencoders [634](#)

using [539](#), [540](#), [541](#)

term frequency (TF) [452](#)

test scores

parameter impact [381](#)

test statistics

distributional characteristics [182](#)

textacy

used, for constructing NLP pipeline [445](#)

TextBlob, with NLP

about [449](#)

sentiment polarity and subjectivity [450](#)

stemming, performing [450](#)

text classifications [456](#)

text data

key challenges [441](#)

parsing [443](#)

tokenizing [443](#)

text data, with ML

about [441](#)

applications [444](#)

TfidfTransformer

using [455](#)

TfidfVectorizer

used, for smoothing documents [455](#)

used, for summarizing news articles [456](#)

using [455](#)

Theano

for Bayesian machine learning [305](#)

tick bars [35](#), [36](#)

time bars [36](#), [39](#)

TimeGAN architecture

components [661](#), [662](#)

joint training, of autoencoder and adversarial network [663](#)

reconstruction loss [663](#)

supervised loss [663](#)

unsupervised loss [663](#)

TimeGAN, implementing with TensorFlow 2 [663](#)

autoencoder, using with real data [667](#), [668](#)

joint training, with real and random data [669](#), [670](#), [671](#)

quality of synthetic time-series data, evaluating [673](#), [674](#)

real and random input series, preparing [665](#), [666](#)

supervised learning, using with real data [669](#)

synthetic time-series data, visualizing with PCA [674](#)

synthetic time-series data, visualizing with t-SNE [674](#), [675](#)

synthetic time series, generating [672](#), [673](#)

TimeGAN model components, creating [666](#), [667](#)

time-series classification performance [675](#), [676](#), [677](#)

time-series prediction model, training on synthetic and real data
[677](#), [678](#)

time indicators

adding, to capture seasonal effects [98](#)

time series [256](#)

transforming, to achieve stationarity [261](#)

time-series approach [285](#)

time series cross-validation

with scikit-learn [168](#)

time series data

used, for building regression tree [332](#), [334](#)

time-series data [165](#)

time-series features

extracting, with seq2seq autoencoders [633](#)

time-series generative adversarial network (TimeGAN) [660](#)

adversarial and supervised training, combining with time-series embedding [661](#)

data generation process, across features [661](#)

data generation process, across time [661](#)

time-series patterns

decomposing [257](#), [258](#)

time-series transformations

in practice [263](#), [264](#), [265](#)

timing of decisions [227](#)

tokens [439](#)

top-down approach [330](#)

topic coherence

used, for evaluating LDA topics [475](#)

topic modeling

approaches [464](#), [466](#)

for earnings calls data [479](#)

for financial news , [482](#)

goals [464](#), [466](#)

trace [303](#)

trade data [41](#)

fields [42](#)

trade execution

 scheduling [138, 139](#)

trade-execution programs [17](#)

trades

 reconstructing [32](#)

trade simulation

 with Zipline [137](#)

trading [23](#)

trading activity [31](#)

TradingCalendar library [243](#)

trading signal quality

 evaluating

trading strategies

 use cases [151](#)

Trading with Python

transaction costs [227](#)

transfer coefficient (TC) [124](#)

transfer learning [558, 567](#)

alternative approaches [567](#)

transfer learning, used for identifying land use with satellite images [574](#)

DenseNet201 [574](#)

EuroSat dataset [574](#)

model training [575](#)

results evaluation [575](#)

transfer learning, with VGG [569](#)

bottleneck features, extracting [569](#), [570](#)

pretrained model, fine-tuning [570](#), [571](#), [572](#)

transformer architecture [596](#)

Transformer model [511](#)

Transmission Control Protocol (TCP) [27](#)

tree pruning [340](#)

tree structure

inspecting [345](#), [347](#)

trend-stationary [261](#)

true positive rates (TPR) [158](#)

two-layer RNN

defining, with single LSTM layer [603](#)

U

ultrafinance

undercomplete [627](#)

underfitting [162](#)

Uniform Manifold Approximation and Projection (UMAP) algorithm [410](#), [425](#), [427](#)

unigram [443](#)

unit root [261](#)

diagnosing [262](#)

unit roots

removing [263](#)

univariate time-series models [265](#)

ARIMA models and extensions [268](#)

autoregressive models [266](#)

features, adding [269](#)

macro fundamentals, forecasting [270](#), [271](#)

moving-average models [267](#)

number of AR and MA terms, identifying [269](#)

seasonal differencing, adding [270](#)

time-series models [272](#)

univariate time-series regression model

evaluating [604](#)

predictions, re-scaling [605](#)

single LSTM layer, used for defining two-layer RNN [603](#)

training [604](#)

used, for predicting S&P 500 index values [601](#), [603](#)

univariate times-series models

designing, guidelines [268](#)

universal approximation theorem [517](#)

unsupervised learning [18](#), [150](#)

unsupervised representation learning

DCGANs, using for [653](#)

use cases, ML for algorithmic trading

asset allocation [19](#)

data mining, for feature extraction [18](#)

data mining, for insights [18](#)

reinforcement learning

supervised learning, using for alpha factor aggregation [18](#)

supervised learning, using for alpha factor creation [18](#)

trade ideas, testing [19](#)

V

validation curves [170](#)

value at risk (VaR) [226](#)

value effects

capturing [89](#)

value factors [88](#)

returns, predicting [89](#)

value function approximation

with NNs [699](#)

value functions [682](#)

value iteration [688](#)

variance [162](#)

variational autoencoders (VAE)

input data, generating [633](#)

variational Bayes [301](#)

variational Bayes (VB) [474](#)

variational inference [301](#)

vector autoregression (VAR) [283](#)

vector autoregressive (VAR) model [278](#), [612](#)

using, for macro fundamentals forecasts [279](#), [280](#), [281](#)

vector error correction model (VECM) [279](#)

vectorized backtest

versus event-driven backtest [230](#), [231](#)

VGGNet [568](#)

visualizations [160](#)

volatility

measuring [92](#)

returns, predicting [91](#)

volatility anomaly [90](#)

volatility model

building, for asset-return series [275](#), [276](#), [277](#)

volume bars [39](#)

volume-weighted average price (VWAP) [35](#), [39](#), [43](#)

Voronoi [433](#)

W

wavelets

noisy signals, preprocessing [106](#), [107](#), [108](#)

weighted least squares (WLS) [184](#)

white noise [257](#)

White standard errors [184](#)

Wiecki, Thomas [317](#)

winner minus loser (WML) factor [85](#)

word2vec models

automating phrase detection [488](#)

hierarchical softmax, using [487](#)

negative sampling (NEG) , using [487](#)

noise contrastive estimation (NCE), using [487](#)

objective [487](#)

phrase embeddings [486](#), [487](#)

softmax function, simplifying [487](#)

using, for trading [501](#)

word embeddings [486](#), [487](#)

word embedding models

characteristics [510](#)

word embeddings

for SEC filings [501](#)

semantics, encoding [485](#)

word representation

GloVe, using for [490](#), [492](#)

WorldQuant

X

XBRL [52](#)

XGBoost [382](#)

Y

Yellowbrick

parameter tuning [170](#)

yfinance [47](#)

end-of-day and intraday prices, downloading [47](#)

option chain, downloading [48](#)

prices, downloading [48](#)

You Only Look Once (YOLO) [576](#)

Z

Zipline [49](#), [50](#), [108](#), [241](#)

backtest
evaluation, with pyfolio
in and out-of-sample strategy backtest, executing
Japanese Equities, ingesting
single-factor strategy, backtesting [108](#)
use, for backtesting long-short trading strategy
using, for portfolio management [137](#)
using, for trade simulation [137](#)

Zipline backtest

pyfolio input, obtaining from [142](#)