6
The Machine Learning Process

In this chapter, we will start to illustrate how you can use a broad range of supervised and unsupervised **machine learning** (**ML**) models for algorithmic trading. We will explain each model's assumptions and use cases before we demonstrate relevant applications using various Python libraries. The categories of models will include:

- Linear models for the regression and classification of cross-section, time series, and panel data
- Generalized additive models, including non-linear tree-based models, such as **decision trees**
- Ensemble models, including random forest and gradient-boosting machines
- Unsupervised linear and nonlinear methods for dimensionality reduction and clustering
- Neural network models, including recurrent and convolutional architectures
- Reinforcement learning models

We will apply these models to the market, fundamental, and alternative data sources introduced in the first part of this book. We will further build on the material covered so far by showing you how to embed these models in an algorithmic trading strategy to generate or combine alpha factors or to optimize the portfolio-management process and evaluate their performance.

There are several aspects that many of these models and their uses have in common. This chapter covers these common aspects so that we can focus on model-specific usage in the following chapters. They include the overarching goal of learning a functional relationship from data by optimizing an objective or loss function. They also include the closely related methods of measuring model performance.

We distinguish between unsupervised and supervised learning and supervised regression and classification problems, and outline use cases for algorithmic trading. We contrast the use of supervised learning for statistical inference of relationships between input and output data with the use for the prediction of future outputs from future inputs. We also illustrate how prediction errors are due to the model's bias or variance, or because of a high noise-to-signal ratio in the data. Most importantly, we present methods to diagnose sources of errors and improve your model's performance.

In this chapter, we will cover the following topics:

- How supervised and unsupervised learning using data works
- How to apply the ML workflow
- How to formulate loss functions for regression and classification
- How to train and evaluate supervised learning models
- How the bias-variance trade-off impacts prediction errors
- How to diagnose and address prediction errors
- How to train a model using cross-validation to manage the bias-variance trade-off
- How to implement cross-validation using scikit-learn
- Why the nature of financial data requires different approaches to out-of-sample testing

If you are already quite familiar with ML, feel free to skip ahead and dive right into learning how to use linear models to produce and combine alpha factors for an algorithmic trading strategy.

Learning from data

There have been many definitions of ML, which all revolve around the automated detection of meaningful patterns in data. Two prominent examples include:

- AI pioneer **Arthur Samuelson** defined ML in 1959 as a subfield of computer science that gives computers the ability to learn without being explicitly programmed.
- **Toni Mitchell**, one of the current leaders in the field, pinned down a well-posed learning problem more specifically in 1998: a computer program learns from experience with respect to a task and a performance measure whether the performance of the task improves with experience.

Experience is presented to an algorithm in the form of training data. The principal difference to previous attempts at building machines that solve problems is that the rules that an algorithm uses to make decisions are learned from the data as opposed to being programmed or hard-coded—this was the case for expert systems prominent in the 1980s.

The key challenge of automated learning is to identify patterns in the training data that are meaningful when generalizing the model's learning to new data. There are a large number of potential patterns that a model could identify, while the training data only constitute a sample of the larger set of phenomena that the algorithm needs to perform the task in the future. The infinite number of functions that could generate the given outputs from the given input make the search process impossible to solve without restrictions on the eligible set of functions.

The types of patterns that an algorithm is capable of learning are limited by the size of its hypothesis space on the one hand and the amount of information contained in the sample data on the other. The size of the hypothesis space varies significantly between algorithms. On the one hand, this limitation enables a successful search and on the other hand, it implies an inductive bias as the algorithm generalizes from the training sample to new data.

Hence, the key challenge becomes a matter of how to choose a model with a hypothesis space large enough to contain a solution to the learning problem, yet small enough to ensure reliable generalization given the size of the training data. With more and more informative data, a model with a larger hypothesis space will be successful.

The **no-free-lunch theorem** states that there is no universal learning algorithm. Instead, a learner's hypothesis space has to be tailored to a specific task using prior knowledge about the task domain in order for the search of meaningful patterns to succeed. We will pay close attention to the assumptions that a model makes about data relationships for a specific task throughout this chapter, and emphasize the importance of matching these assumptions with empirical evidence gleaned from data exploration. The process required to master the task can be differentiated into supervised, unsupervised, and reinforcement learning.

Supervised learning

Supervised learning is the most commonly used type of ML. We will dedicate most of the chapters in this book to learning about the various applications of models in this category. The term *supervised* implies the presence of an outcome variable that guides the learning process—that is, it teaches the algorithm the correct solution to the task that is being learned. Supervised learning aims at generalizing a functional relationship between input and output data that is learned from individual samples and applying it to new data.

The output variable is also, depending on the field, interchangeably called the label, target, outcome, endogenous, or left-hand-side variable. We will use y_i for observations $i = 1, ..., N$, or y in vector notation. Some tasks are represented by several outcomes, also called **multilabel problems**. The input data for a supervised learning problem is also known as features, exogenous, and right-hand-side variables, denoted by an x_i for a vector of features for observations $i = 1, ..., N$, or X in matrix notation.

The solution to a supervised learning problem is a function ($\hat{f}$) that represents what the model learned about the input-output relationship from the sample and approximates the true relationship, represented with $\mathbf{y} \approx \hat{f}(\mathbf{X})$. This function can be used to infer statistical associations or potentially even causal relationships among variables of interest beyond the sample, or it can be used to predict outputs for new input data.

Both goals face an important trade-off: more complex models have more *moving parts* that are capable of representing more nuanced relationships, but they may also be more difficult to inspect. They are also likely to overfit and learn random noise particular to the training sample, as opposed to a systematic signal that represents a general pattern of the input-output relationship. Overly simple models, on the other hand, will miss signals and deliver biased results. This trade-off is known as the **bias-variance trade-off** in supervised learning, but conceptually this also applies to the other forms of ML where overly complex models may perform poorly beyond the training data.

Unsupervised learning

When solving an unsupervised learning problem, we only observe the features and have no measurements of the outcome. Instead of the prediction of future outcomes or the inference of relationships among variables, the task is to find structure in the data without any outcome information to guide the search process.

Often, unsupervised algorithms aim to learn a new representation of the input data that is useful for some other tasks. This includes coming up with labels that identify commonalities among observations, or a summarized description that captures relevant information while requiring data points or features. Unsupervised learning algorithms also differ from supervised learning algorithms in the assumptions they make about the nature of the structure they are aiming to discover.

Applications

There are several helpful uses of unsupervised learning that can be applied to algorithmic trading, including the following:

- Grouping together securities with similar risk and return characteristics (see hierarchical risk parity in this chapter (which looks at portfolio optimization))
- Finding a small number of risk factors driving the performance of a much larger number of securities
- Identifying trading and price patterns that differ systematically and may pose higher risks
- Identifying latent topics in a body of documents (for example, earnings call transcripts) that comprise the most important aspects of those documents

At a high level, these applications rely on methods to identify clusters and methods to reduce the dimensionality of the data.

Cluster algorithms

Cluster algorithms use a measure of similarity to identify observations or data attributes that contain similar information. They summarize a dataset by assigning a large number of data points to a smaller number of clusters so that the cluster members are more closely related to each other than to members of other clusters.

Cluster algorithms primarily differ with respect to the type of clusters that they will produce, which implies different assumptions about the data generation process, listed as follows:

- **K-means clustering**: Data points belong to one of the *k* clusters of equal size that take an elliptical form
- **Gaussian mixture models**: Data points have been generated by any of the various multivariate normal distributions

- **Density-based clusters**: Clusters are of an arbitrary shape and are defined only by the existence of a minimum number of nearby data points
- **Hierarchical clusters**: Data points belong to various supersets of groups that are formed by successively merging smaller clusters

Dimensionality reduction

Dimensionality reduction produces new data that captures the most important information contained in the source data. Rather than grouping existing data into clusters, these algorithms transform existing data into a new dataset that uses significantly fewer features or observations to represent the original information.

Algorithms differ with respect to the nature of the new dataset they will produce, as shown in the following list:

- **Principal component analysis (PCA)**: Finds the linear transformation that captures most of the variance in the existing dataset
- **Manifold learning**: Identifies a nonlinear transformation that produces a lower-dimensional representation of the data
- **Autoencoders**: Uses neural networks to compress data non-linearly with minimal loss of information

We will dive deeper into linear, non-linear, and neural-network-based unsupervised learning models in several of the following chapters, including important applications of **natural language processing** (**NLP**) in the form of topic modeling and Word2vec feature extraction.

Reinforcement learning

Reinforcement learning is the third type of ML. It aims to choose the action that yields the highest reward, given a set of input data that describes a context or environment. It is both dynamic and interactive: the stream of positive and negative rewards impacts the algorithm's learning, and actions taken now may influence both the environment and future rewards.

The trade-off between the exploitation of a course of action that has been learned to yield a certain reward and the exploration of new actions that may increase the reward in the future gives rise to a trial-and-error approach. Reinforcement learning optimizes the agent's learning using dynamical systems theory and, in particular, the optimal control of Markov decision processes with incomplete information.

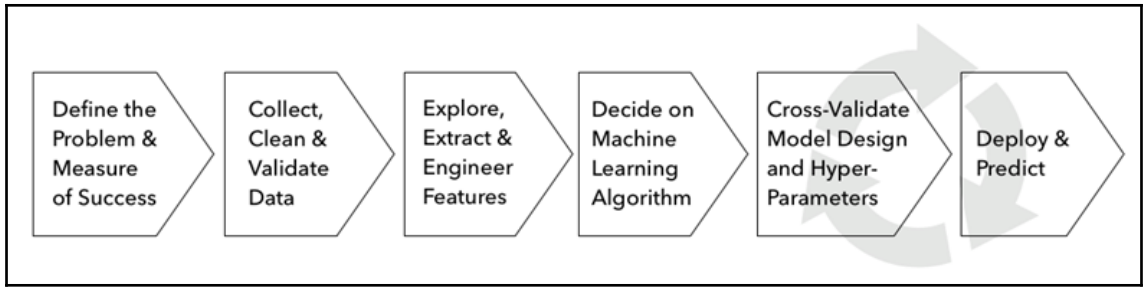
Reinforcement learning differs from supervised learning, where the available training data lays out both the context and the correct decision for the algorithm. It is tailored to interactive settings where the outcomes only become available over time and learning has to proceed in an *online* or continuous fashion as the agent acquires new experience. However, some of the most notable progress in **Artificial Intelligence** (**AI**) involves reinforcement that uses deep learning to approximate functional relationships between actions, environments, and future rewards. It also differs from unsupervised learning because feedback of the consequences will be available, albeit with a delay.

Reinforcement learning is particularly suitable for algorithmic trading because the concept of a return-maximizing agent in an uncertain, dynamic environment has much in common with an investor or a trading strategy that interacts with financial markets. This approach has been successfully applied to game-playing agents, most prominently to the game of Go, but also to complex video games. It is also used in robotics—for example, self-driving cars—or to personalize services such as website offerings based on user interaction. We will introduce reinforcement learning approaches to building an algorithmic trading strategy in `Chapter 21`, *Reinforcement Learning*.

The machine learning workflow

Developing a ML solution for an algorithmic trading strategy requires a systematic approach to maximize the chances of success while economizing on resources. It is also very important to make the process transparent and replicable in order to facilitate collaboration, maintenance, and later refinements.

The following chart outlines the key steps from problem definition to the deployment of a predictive solution:



The process is iterative throughout the sequence, and the effort required at different stages will vary according to the project, but this process should generally include the following steps:

1. Frame the problem, identify a target metric, and define success
2. Source, clean, and validate the data
3. Understand your data and generate informative features
4. Pick one or more machine learning algorithms suitable for your data
5. Train, test, and tune your models
6. Use your model to solve the original problem

We will walk through these steps in the following sections using a simple example to illustrate some of the key points.

Basic walkthrough – k-nearest neighbors

The `machine_learning_workflow.ipynb` notebook in this chapter's folder of the book's GitHub repository contains several examples that illustrate the machine learning workflow using a dataset of house prices.

We will use the fairly straightforward **k-nearest neighbors** (**KNN**) algorithm that allows us to tackle both regression and classification problems.

In its default `sklearn` implementation, it identifies the `k` nearest data points (based on the Euclidean distance) to make a prediction. It predicts the most frequent class among the neighbors or the average outcome in the classification or regression case, respectively.

Frame the problem – goals and metrics

The starting point for any machine learning exercise is the ultimate use case it aims to address. Sometimes, this goal will be statistical inference in order to identify an association between variables or even a causal relationship. Most frequently, however, the goal will be the direct prediction of an outcome to yield a trading signal.

Both inference and prediction use metrics to evaluate how well a model achieves its objective. We will focus on common objective functions and the corresponding error metrics for predictive models that can be distinguished by the variable type of the output: continuous output variables imply a regression problem, categorical variables imply classification, and the special case of ordered categorical variables implies ranking problems.

The problem may be the efficient combination of several alpha factors and could be framed as a regression problem that aims to predict returns, a binary classification problem that aims to predict the direction of future price movements, or a multiclass problem that aims to assign stocks to various performance classes. In the following section, we will introduce these objectives and look at how to measure and interpret related error metrics.

Prediction versus inference

The functional relationship produced by a supervised learning algorithm can be used for inference—that is, to gain insights into how the outcomes are generated—or for prediction—that is, to generate accurate output estimates (represented by $\hat{y}$) for unknown or future inputs (represented by X).

For algorithmic trading, inference can be used to estimate the causal or statistical dependence of the returns of an asset on a risk factor, whereas prediction can be used to forecast the risk factor. Combining the two can yield a prediction of the asset price, which in turn can be translated into a trading signal.

Statistical inference is about drawing conclusions from sample data about the parameters of the underlying probability distribution or the population. Potential conclusions include hypothesis tests about the characteristics of the distribution of an individual variable, or the existence or strength of numerical relationships among variables. They also include point or interval estimates of statistical metrics.

Inference depends on the assumptions about the process that generates the data in the first place. We will review these assumptions and the tools that are used for inference with linear models where they are well established. More complex models make fewer assumptions about the structural relationship between input and output, and instead approach the task of function approximation more openly while treating the data-generating process as a black box. These models, including decision trees, ensemble models, and neural networks, are focused on and often outperform when used for prediction tasks. However, random forests have recently gained a framework for inference that we will introduce later.

Causal inference

Causal inference aims to identify relationships so that certain input values imply certain outputs—for example, a certain constellation of macro variables causing the price of a given asset to move in a certain way, assuming all other variables remain constant.

Statistical inference about relationships among two or more variables produces measures of correlation that can only be interpreted as a causal relationship when several other conditions are met—for example, when alternative explanations or reverse causality has been ruled out. Meeting these conditions requires an experimental setting where all relevant variables of interest can be fully controlled to isolate causal relationships. Alternatively, quasi-experimental settings expose units of observations to changes in inputs in a randomized way to rule out that other observable or unobservable features are responsible for the observed effects of the change in the environment.

These conditions are rarely met so inferential conclusions need to be treated with care. The same applies to the performance of predictive models that also rely on the statistical association between features and outputs, which may change with other factors that are not part of the model.

The non-parametric nature of the KNN model does not lend itself well to inference, so we'll postpone this step in the workflow until we encounter linear models in the next chapter.

Regression problems

Regression problems aim to predict a continuous variable. The **root-mean-square error** (**RMSE**) is the most popular loss function and error metric, not least because it is differentiable. The loss is symmetric, but larger errors weigh more in the calculation. Using the square root has the advantage of measuring the error in the units of the target variable. The same metric in combination with the **RMSE log of the error** (**RMSLE**) is appropriate when the target is subject to exponential growth because of its asymmetric penalty that weights negative errors less than positive errors. You can also log-transform the target first and then use the RMSE, as we do in the example later in this section.

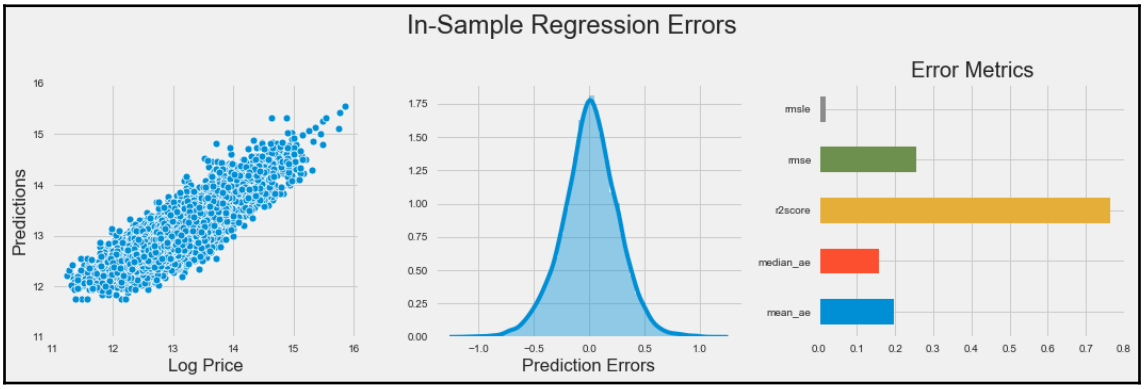
The **mean absolute errors** (**MAE**) and **median absolute errors** (**MedAE**) are symmetric but do not weigh larger errors more. The MedAE is robust to outliers.

The explained variance score computes the proportion of the target variance that the model accounts for and varies between 0 and 1. The R^2 score or coefficient of determination yields the same outcome the mean of the residuals is 0, but can differ otherwise. In particular, it can be negative when calculated on out-of-sample data (or for a linear regression without intercept).

The following table defines the formulas used for calculation and the corresponding `sklearn` function that can be imported from the `metrics` module. The `scoring` parameter is used in combination with automated train-test functions (such as `cross_val_score` and `GridSearchCV`) that we will introduce later in this section, and which are illustrated in the accompanying notebook:

Name	Formula	sklearn	Scoring parameter
Mean squared error	$\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$	`mean_squared_error`	`neg_mean_squared_error`
Mean squared log error	$\frac{1}{N}\sum_{i=1}^{N}(\ln(1+y_i) - \ln(1+\hat{y}_i))^2$	`mean_squared_log_error`	`neg_mean_squared_log_error`
Mean absolute error	$\frac{1}{N}\sum_{i=1}^{N}\lvert y_i - \hat{y}_i\rvert$	`mean_absolute_error`	`neg_mean_absolute_error`
Median absolute error	$\text{median}(\lvert y_1 - \hat{y}_1\rvert, \ldots, \lvert y_n - \hat{y}_n\rvert)$	`median_absolute_error`	`neg_median_absolute_error`
Explained variance	$1 - \frac{\text{var}(y - \hat{y})}{\text{var}(y)}$	`explained_variance_score`	`explained_variance`
R^2 score	$1 - \frac{\sum_{i=1}^{N}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{N}(y_i - \bar{y})^2}$	`r2_score`	`r2`

The following screenshot shows the various error metrics for the house price regression demonstrated in the notebook:



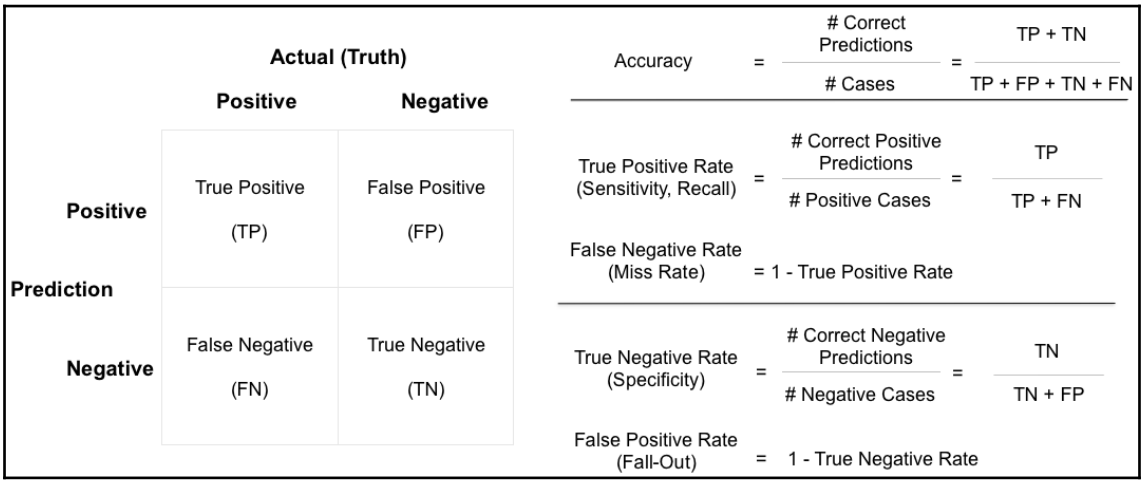
The `sklearn` function also supports multilabel evaluation—that is, assigning multiple outcome values to a single observation; see the documentation referenced on GitHub for more details (`https://github.com/PacktPublishing/Hands-On-Machine-Learning-for-Algorithmic-Trading/tree/master/Chapter06`).

Classification problems

Classification problems have categorical outcome variables. Most predictors will output a score to indicate whether an observation belongs to a certain class. In the second step, these scores are then translated into actual predictions.

In the binary case, where we will label the classes positive and negative, the score typically varies between zero or is normalized accordingly. Once the scores are converted into 0-1 predictions, there can be four outcomes, because each of the two existing classes can be either correctly or incorrectly predicted. With more than two classes, there can be more cases if you differentiate between the several potential mistakes.

All error metrics are computed from the breakdown of predictions across the four fields of the 2 x 2 confusion matrix that associates actual and predicted classes. The metrics listed in the table shown in the following diagram, such as accuracy, evaluate a model for a given threshold:



A classifier does not necessarily need to output calibrated probabilities, but should rather produce scores that are relative to each other in distinguishing positive from negative cases. Hence, the threshold is a decision variable that can and should be optimized, taking into account the costs and benefits of correct and incorrect predictions. A lower threshold implies more positive predictions, with a potentially rising false positive rate, and for a higher threshold, the opposite is likely to be true.

Receiver operating characteristics and the area under the curve

The **receiver operating characteristics** (**ROC**) curve allows us to visualize, organize, and select classifiers based on their performance. It computes all the combinations of **true positive rates** (**TPR**) and **false positive rates** (**FPR**) that result from producing predictions using any of the predicted scores as a threshold. It then plots these pairs on a square, the side of which has a measurement of one in length.

A classifier that makes random predictions (taking into account class imbalance) will on average yield TPR and FPR that are equal so that the combinations will lie on the diagonal, which becomes the benchmark case. Since an underperforming classifier would benefit from relabeling the predictions, this benchmark also becomes the minimum.

The **area under the curve** (**AUC**) is defined as the area under the ROC plot that varies between 0.5 and the maximum of 1. It is a summary measure of how well the classifier's scores are able to rank data points with respect to their class membership. More specifically, the AUC of a classifier has the important statistical property of representing the probability that the classifier will rank a randomly chosen positive instance higher than a randomly chosen negative instance, which is equivalent to the Wilcoxon ranking test. In addition, the AUC has the benefit of not being sensitive to class imbalances.

Precision-recall curves

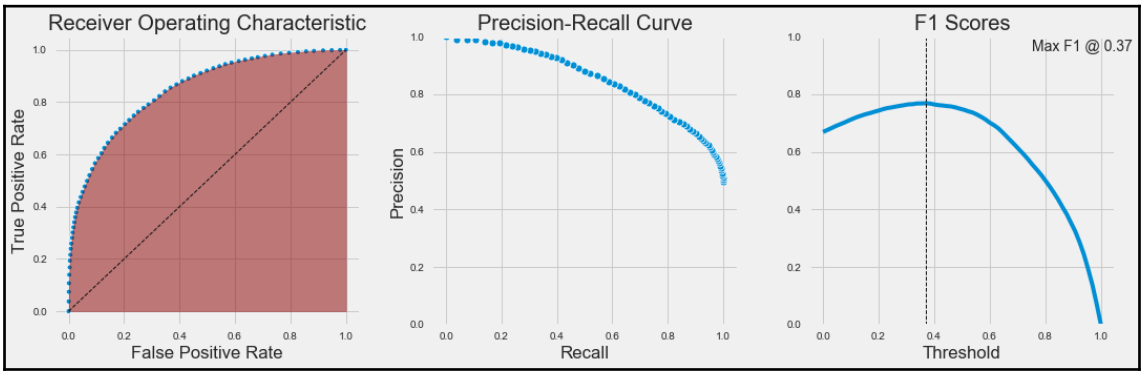
When predictions for one of the classes are of particular interest, precision and recall curves visualize the trade-off between these error metrics for different thresholds. Both measures evaluate the quality of predictions for a particular class. The following list shows how they are applied to the positive class:

- **Recall** measures the share of actual positive class members that a classifier predicts as positive for a given threshold. It originates in information retrieval and measures the share of relevant documents successfully identified by a search algorithm.
- **Precision**, in contrast, measures the share of positive predictions that are correct.

Recall typically increases with a lower threshold, but precision may decrease. Precision-recall curves visualize the attainable combinations and allow for the optimization of the threshold given the costs and benefits of missing a lot of relevant cases or producing lower-quality predictions.

The F1 score is a harmonic mean of precision and recall for a given threshold and can be used to numerically optimize the threshold while taking into account the relative weights that these two metrics should assume.

The following chart illustrates the ROC curve and corresponding AUC alongside the precision-recall curve and the F1 score that, using equal weights for precision and recall, yields an optimal threshold of 0.37. The chart is taken from the accompanying notebook where you can find the code for the KNN classifier that operates on binarized housing prices:



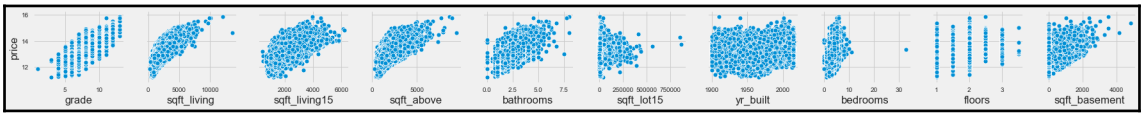
Collecting and preparing the data

We already addressed important aspects of the sourcing of market, fundamental, and alternative data, and will continue to work with various examples of these sources as we illustrate the application of the various models.

In addition to market and fundamental data that we will access through the Quantopian platform, we will also acquire and transform text data as we explore natural language processing and image data when we look at image processing and recognition. Besides obtaining, cleaning, and validating the data to relate it to trading data typically available in a time-series format, it is important to store it in a format that allows for fast access to enable quick exploration and iteration. We have recommended the HDF and parquet formats. For larger data volumes, Apache Spark represents the best solution.

Explore, extract, and engineer features

Understanding the distribution of individual variables and the relationships among outcomes and features is the basis for picking a suitable algorithm. This typically starts with visualizations such as scatter plots, as illustrated in the companion notebook (and shown in the following image), but also includes numerical evaluations ranging from linear metrics, such as the correlation, to nonlinear statistics, such as the Spearman rank correlation coefficient that we encountered when we introduced the information coefficient. It also includes information-theoretic measures, such as mutual information, as illustrated in the next subsection:



Scatter plots

A systematic exploratory analysis is also the basis of what is often the single most important ingredient of a successful predictive model: the engineering of features that extract information contained in the data, but which are not necessarily accessible to the algorithm in their raw form. Feature engineering benefits from domain expertise, the application of statistics and information theory, and creativity.

It relies on an ingenious choice of data transformations that effectively tease out the systematic relationship between input and output data. There are many choices that include outlier detection and treatment, functional transformations, and the combination of several variables, including unsupervised learning. We will illustrate examples throughout but will emphasize that this feature is best learned through experience. Kaggle is a great place to learn from other data scientists who share their experiences with the Kaggle community.

Using information theory to evaluate features

The **mutual information** (**MI**) between a feature and the outcome is a measure of the mutual dependence between the two variables. It extends the notion of correlation to nonlinear relationships. More specifically, it quantifies the information obtained about one random variable through the other random variable.

The concept of MI is closely related to the fundamental notion of entropy of a random variable. Entropy quantifies the amount of information contained in a random variable. Formally, the mutual information—$I(X, Y)$—of two random variables, X and Y, is defined as the following:

$$I(X,Y) = \int_Y \int_X p(x,y) \log\left(\frac{p(x,y}{p(x)p(y)}\right)$$

The `sklearn` function implements `feature_selection.mutual_info_regression` that computes the mutual information between all features and a continuous outcome to select the features that are most likely to contain predictive information. There is also a classification version (see the documentation for more details). The notebook `mutual_information.ipynb` notebook contains an application to the financial data we created in `Chapter 4`, *Alpha Factor Research*.

Selecting an ML algorithm

The remainder of this book will introduce several model families, ranging from linear models, which make fairly strong assumptions about the nature of the functional relationship between input and output variables, to deep neural networks, which make very few assumptions. As mentioned in the introductory section, fewer assumptions will require more data with significant information about the relationship so that the learning process can be successful.

We will outline the key assumptions and how to test them where applicable as we introduce these models.

Design and tune the model

The ML process includes steps to diagnose and manage model complexity based on estimates of the model's generalization error. An unbiased estimate requires a statistically sound and efficient procedure, as well as error metrics that align with the output variable type, which also determines whether we are dealing with a regression, classification, or ranking problem.

The bias-variance trade-off

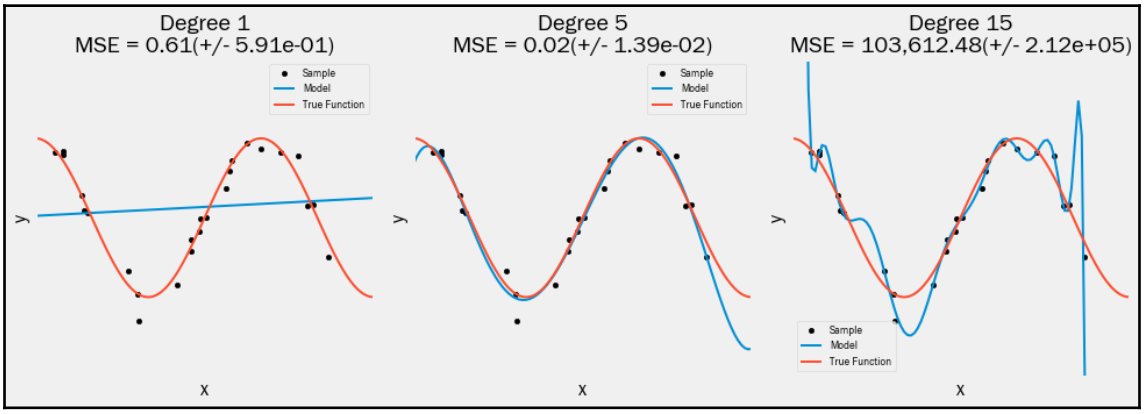
The errors that an ML model makes when predicting outcomes for new input data can be broken down into reducible and irreducible parts. The irreducible part is due to random variation (noise) in the data that is not measured, such as relevant but missing variables or natural variation. The reducible part of the generalization error, in turn, can be broken down into bias and variance. Both are due to differences between the true functional relationship and the assumptions made by the machine learning algorithm, as detailed in the following list:

- **Error due to bias**: The hypothesis is too simple to capture the complexity of the true functional relationship. As a result, whenever the model attempts to learn the true function, it makes systematic mistakes and, on average, the predictions will be similarly biased. This is also called **underfitting**.
- **Error due to variance**: The algorithm is overly complex in view of the true relationship. Instead of capturing the true relationship, it overfits to the data and extracts patterns from the noise. As a result, it learns different functional relationships from each sample, and out-of-sample predictions will vary widely.

Underfitting versus overfitting

The following diagram illustrates overfitting by approximating a cosine function using increasingly complex polynomials and measuring the in-sample error. More specifically, we draw 10 random samples with some added noise ($n = 30$) to learn a polynomial of varying complexity (see the code in the accompanying notebook). Each time, the model predicts new data points and we capture the mean-squared error for these predictions, as well as the standard deviation of these errors.

The left-hand panel in the following diagram shows a polynomial of degree 1; a straight line clearly underfits the true function. However, the estimated line will not differ dramatically from one sample drawn from the true function to the next. The middle panel shows that a degree 5 polynomial approximates the true relationship reasonably well on the [0, 1] interval. On the other hand, a polynomial of degree 15 fits the small sample almost perfectly, but provides a poor estimate of the true relationship: it overfits to the random variation in the sample data points, and the learned function will vary strongly with each sample drawn:

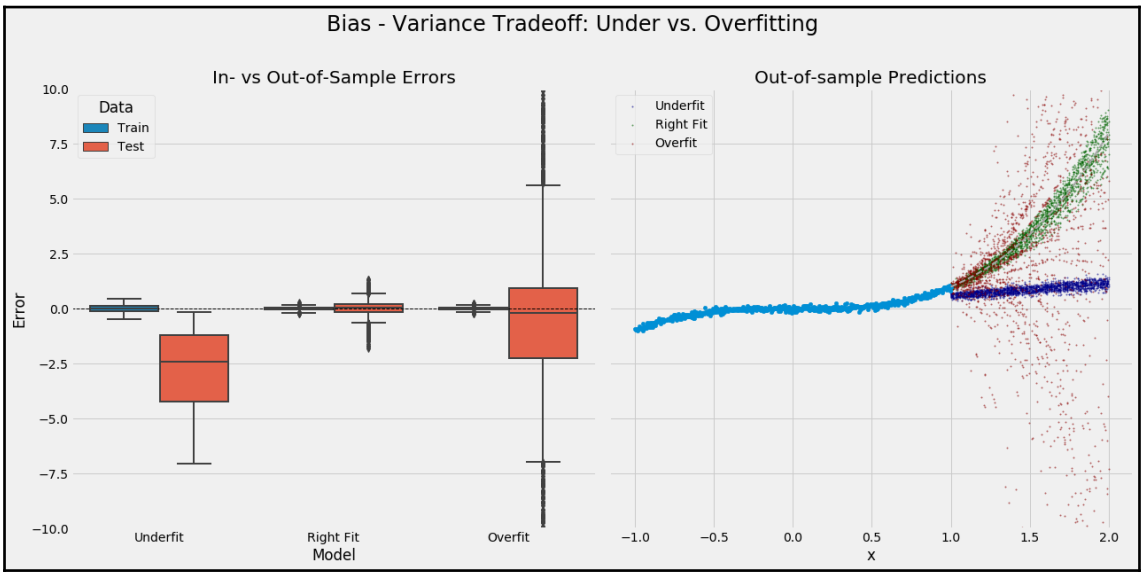


Managing the trade-off

Let's further illustrate the impact of overfitting versus underfitting by trying to learn a Taylor series approximation of the cosine function of ninth degree with some added noise. In the following diagram, we draw random samples of the true function and fit polynomials that underfit, overfit, and provide an approximately correct degree of flexibility. We then predict out-of-sample and measure the RMSE.

The high bias but low variance of a polynomial of degree 3 compares to the low bias but exceedingly high variance of the various prediction errors visible in the first panel. The left-hand panel shows the distribution of the errors that result from subtracting the true function values. The underfit case of a straight line produces a poor in-sample fit and is significantly off target out of sample. The overfit model shows the best fit in-sample with the smallest dispersion of errors, but the price is a large variance out-of-sample. The appropriate model that matches the functional form of the true model performs the best by far out-of-sample.

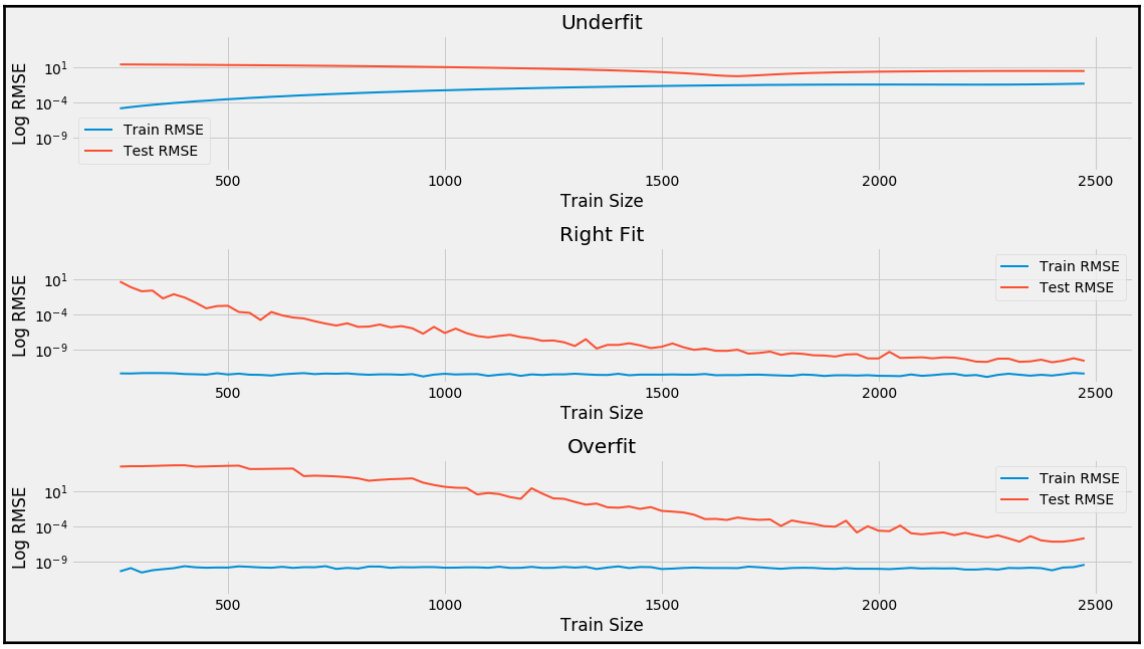
The right-hand panel of the following screenshot shows the actual predictions rather than the errors to demonstrate what the different types of fit look like in practice:



Learning curves

A learning curve plots the evolution of train and test errors against the size of the dataset used to learn the functional relationship. It is a useful tool to diagnose the bias-variance trade-off for a given model because the errors will behave differently. A model with a high bias will have a high but similar training error, both in-sample and out-of-sample, whereas an overfit model will have a very low training error.

The declining out-of-sample error illustrates that overfit models may benefit from additional data or tools to limit the model's complexity, such as regularization, whereas underfit models need to use either more features or otherwise increase the complexity of the model, as shown in the following screenshot:



How to use cross-validation for model selection

When several candidate models (that is, algorithms) are available for your use case, the act of choosing one of them is called the **model selection** problem. Model selection aims to identify the model that will produce the lowest prediction error given new data.

An unbiased estimate of this generalization error requires a test on data that was not part of model training. Hence, we only use part of the available data to train the model and set aside another part of the data to test the model. In order to obtain an unbiased estimate of the prediction error, absolutely no information about the test set may leak into the training set, as shown in the following diagram:

There are several methods that can be used to split the available data, which differ in terms of the amount of data used for training, the variance of the error estimates, the computational intensity, and whether structural aspects of the data are taken into account when splitting the data.

How to implement cross-validation in Python

We will illustrate various options for splitting data into training and test sets by showing how the indices of a mock dataset with ten observations are assigned to the train and test set (see `cross_validation.py` for details), as shown in following code:

```
data = list(range(1, 11))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Basic train-test split

For a single split of your data into a training and a test set, use `sklearn.model_selection.train_test_split`, where the `shuffle` parameter, by default ensures the randomized selection of observations, which in turn can be replicated by setting `random_state`. There is also a `stratify` parameter that, for a classification problem, ensures that the train and test sets will contain approximately the same shares of each class, as shown in the following code:

```
train_test_split(data, train_size=.8)
[[8, 7, 4, 10, 1, 3, 5, 2], [6, 9]]
```

In this case, we train a model using all data except row numbers `6` and `9`, which will be used to generate predictions and measure the errors given on the `know` labels. This method is useful for quick evaluation but is sensitive to the split, and the standard error of the test error estimate will be higher.

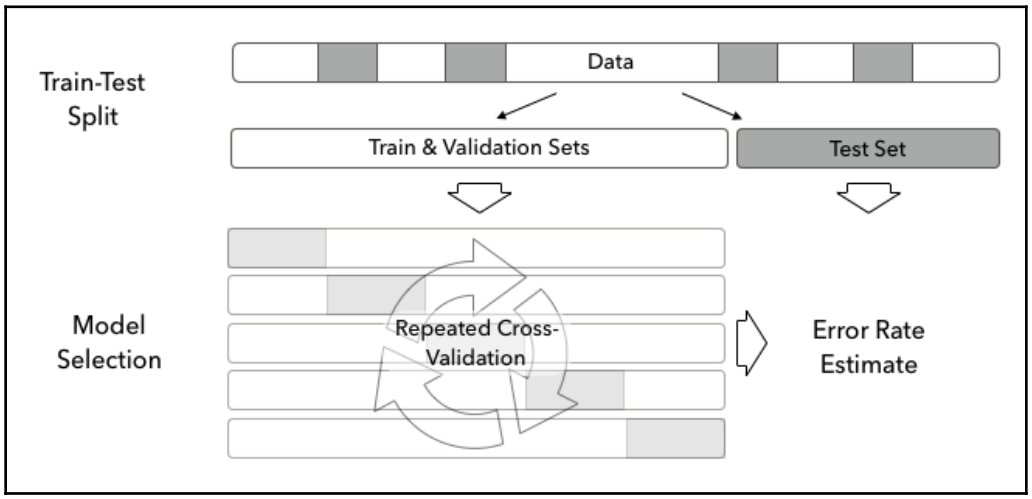
Cross-validation

Cross-validation (**CV**) is a popular strategy for model selection. The main idea behind CV is to split the data one or several times so that each split is used once as a validation set and the remainder as a training set: part of the data (the training sample) is used to train the algorithm, and the remaining part (the validation sample) is used for estimating the risk of the algorithm. Then, CV selects the algorithm with the smallest estimated risk.

While the data-splitting heuristic is very general, a key assumption of CV is that the data is **independently and identically distributed** (**IID**). In the following sections, we will see that, for time series data, this is often not the case and requires a different approach.

Using a hold-out test set

When selecting hyperparameters based on their validation score, be aware that this validation score is biased because of multiple tests, and is no longer a good estimate of the generalization error. For an unbiased estimate of the error rate, we have to estimate the score from a fresh dataset, as shown in the following diagram:



For this reason, we use a three-way split of the data, as illustrated in the preceding diagram: one part is used in cross-validation and is repeatedly split into a training and validation set. The remainder is set aside as a hold-out set that is only used once cross-validation is complete to generate an unbiased test error estimate. We will illustrate this method as we start building ML models in the next chapter.

KFold iterator

The `sklearn.model_selection.KFold` iterator produces several disjunct splits and assigns each of these splits once to the validation set, as shown in the following code:

```
kf = KFold(n_splits=5)
for train, validate in kf.split(data):
    print(train, validate)

[2 3 4 5 6 7 8 9] [0 1]
[0 1 4 5 6 7 8 9] [2 3]
[0 1 2 3 6 7 8 9] [4 5]
[0 1 2 3 4 5 8 9] [6 7]
[0 1 2 3 4 5 6 7] [8 9]
```

In addition to the number of splits, most CV objects take a `shuffle` argument that ensures randomization. To render results reproducible, set the `random_state`, as follows:

```
kf = KFold(n_splits=5, shuffle=True, random_state=42)
for train, validate in kf.split(data):
    print(train, validate)

[0 2 3 4 5 6 7 9] [1 8]
[1 2 3 4 6 7 8 9] [0 5]
[0 1 3 4 5 6 8 9] [2 7]
[0 1 2 3 5 6 7 8] [4 9]
[0 1 2 4 5 7 8 9] [3 6]
```

Leave-one-out CV

The original CV implementation used a leave-one-out method that used each observation once as the validation set, as shown in the following code:

```
loo = LeaveOneOut()
for train, validate in loo.split(data):
    print(train, validate)

[1 2 3 4 5 6 7 8 9] [0]
[0 2 3 4 5 6 7 8 9] [1]
...
```

```
[0 1 2 3 4 5 6 7 9] [8]
[0 1 2 3 4 5 6 7 8] [9]
```

This maximizes the number of models that are trained, which increases computational costs. While the validation sets do not overlap, the overlap of training sets is maximized, driving up the correlation of models and their prediction errors. As a result, the variance of the prediction error is higher for a model with a larger number of folds.

Leave-P-Out CV

A similar version to leave-one-out CV is leave-P-out CV, which generates all possible combinations of `p` data rows, as shown in the following code:

```
lpo = LeavePOut(p=2)
for train, validate in lpo.split(data):
    print(train, validate)

[2 3 4 5 6 7 8 9] [0 1]
[1 3 4 5 6 7 8 9] [0 2]
...
[0 1 2 3 4 5 6 8] [7 9]
[0 1 2 3 4 5 6 7] [8 9]
```

ShuffleSplit

The `sklearn.model_selection.ShuffleSplit` object creates independent splits with potentially overlapping validation sets, as shown in the following code:

```
ss = ShuffleSplit(n_splits=3, test_size=2, random_state=42)
for train, validate in ss.split(data):
    print(train, validate)
[4 9 1 6 7 3 0 5] [2 8]
[1 2 9 8 0 6 7 4] [3 5]
[8 4 5 1 0 6 9 7] [2 3]
```

Parameter tuning with scikit-learn

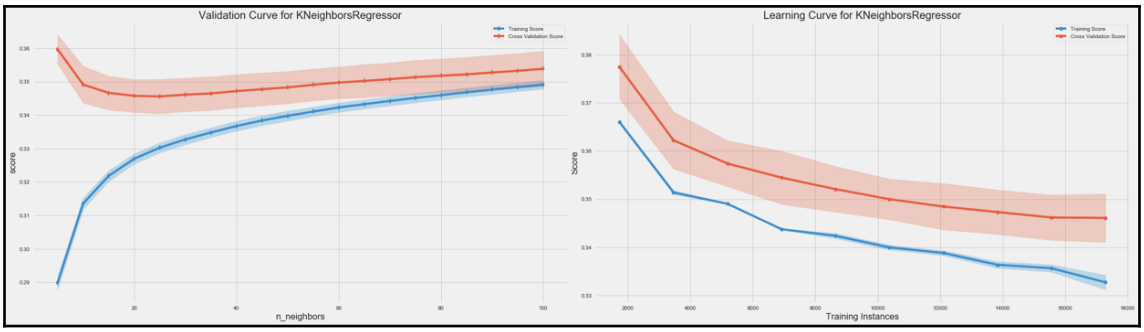
Model selection typically involves repeated cross-validation of the out-of-sample performance of models using different algorithms (such as linear regression and random forest) or different configurations. Different configurations may involve changes to hyperparameters or the inclusion or exclusion of different variables.

The `yellowbricks` library extends the `sklearn` API to generate diagnostic visualization tools to facilitate the model-selection process. These tools can be used to investigate relationships among features, analyze classification or regression errors, monitor cluster algorithm performance, inspect the characteristics of text data, and help with model selection. We will demonstrate validation and learning curves that provide valuable information during the parameter-tuning phase—see the `machine_learning_workflow.ipynb` notebook for implementation details.

Validation curves with yellowbricks

Validation curves (see the left-hand panel in the following graph) visualize the impact of a single hyperparameter on a model's cross-validation performance. This is useful to determine whether the model underfits or overfits the given dataset.

In our example of the **KNeighborsRegressor** that only has a single hyperparameter, we can clearly see that the model underfits for values of *k* above 20, where the validation error drops as we reduce the number of neighbors, thereby making our model more complex, because it makes predictions for more distinct groups of neighbors or areas in the feature space. For values below 20, the model begins to overfit as training and validation errors diverge and average out-of-sample performance quickly deteriorates, as shown in the following graph:



Learning curves

The learning curve (see the right-hand panel of the preceding chart for our house price regression example) helps determine whether a model's cross-validation performance would benefit from additional data and whether prediction errors are more driven by bias or by variance.

If training and cross-validation performance converges, then more data is unlikely to improve the performance. At this point, it is important to evaluate whether the model performance meets expectations, determined by a human benchmark. If this is not the case, then you should modify the model's hyperparameter settings to better capture the relationship between the features and the outcome, or choose a different algorithm with a higher capacity to capture complexity.

In addition, the variation of train and test errors shown by the shaded confidence intervals provide clues about the bias and variance sources of the prediction error. Variability around the cross-validation error is evidence of variance, whereas variability for the training set suggests bias, depending on the size of the training error.

In our example, the cross-validation performance has continued to drop, but the incremental improvements have shrunk and the errors have plateaued, so there are unlikely to be many benefits from a larger training set. On the other hand, the data is showing substantial variance given the range of validation errors compared to that shown for the training errors.

Parameter tuning using GridSearchCV and pipeline

Since hyperparameter tuning is a key ingredient of the machine learning workflow, there are tools to automate this process. The `sklearn` library includes a `GridSearchCV` interface that cross-validates all combinations of parameters in parallel, captures the result, and automatically trains the model using the parameter setting that performed best during cross-validation on the full dataset.

In practice, the training and validation sets often require some processing prior to cross-validation. Scikit-learn offers the `Pipeline` to also automate any requisite feature-processing steps in the automated hyperparameter tuning facilitated by `GridSearchCV`.

You can look at the implementation examples in the included `machine_learning_workflow.ipynb` notebook to see these tools in action.

Challenges with cross-validation in finance

A key assumption for the cross-validation methods discussed so far is the **independent and identical** (**iid**) distribution of the samples available for training.

For financial data, this is often not the case. On the contrary, financial data is neither independently nor identically distributed because of serial correlation and time-varying standard deviation, also known as **heteroskedasticity** (see the next two chapters for more details). The `TimeSeriesSplit` in the `sklearn.model_selection` module aims to address the linear order of time-series data.

Time series cross-validation with sklearn

The time series nature of the data implies that cross-validation produces a situation where data from the future will be used to predict data from the past. This is unrealistic at best and data snooping at worst, to the extent that future data reflects past events.

To address time dependency, the `sklearn.model_selection.TimeSeriesSplit` object implements a walk-forward test with an expanding training set, where subsequent training sets are supersets of past training sets, as shown in the following code:

```
tscv = TimeSeriesSplit(n_splits=5)
for train, validate in tscv.split(data):
    print(train, validate)

[0 1 2 3 4] [5]
[0 1 2 3 4 5] [6]
[0 1 2 3 4 5 6] [7]
[0 1 2 3 4 5 6 7] [8]
[0 1 2 3 4 5 6 7 8] [9]
```

You can use the `max_train_size` parameter to implement walk-forward cross-validation, where the size of the training set remains constant over time, similar to how `zipline` tests a trading algorithm. Scikit-learn facilitates the design of custom cross-validation methods using subclassing, which we will implement in the following chapters.

Purging, embargoing, and combinatorial CV

For financial data, labels are often derived from overlapping data points as returns are computed from prices in multiple periods. In the context of trading strategies, the results of a model's prediction, which may imply taking a position in an asset, may only be known later, when this decision is evaluated—for example, when a position is closed out.

The resulting risks include the leaking of information from the test into the training set, likely leading to an artificially inflated performance that needs to be addressed by ensuring that all data is point-in-time—that is, truly available and known at the time it is used as the input for a model. Several methods have been proposed by Marcos Lopez de Prado in *Advances in Financial Machine Learning* to address these challenges of financial data for cross-validation, as shown in the following list:

- **Purging**: Eliminate training data points where the evaluation occurs after the prediction of a point-in-time data point in the validation set to avoid look-ahead bias.
- **Embargoing**: Further eliminate training samples that follow a test period.
- **Combinatorial cross-validation**: Walk-forward CV severely limits the historical paths that can be tested. Instead, given T observations, compute all possible train/test splits for $N<T$ groups that each maintain their order, and purge and embargo potentially overlapping groups. Then, train the model on all combinations of $N-k$ groups while testing the model on the remaining k groups. The result is a much larger number of possible historical paths.

Prado's *Advances in Financial Machine Learning* contains sample code to implement these approaches; the code is also available via the new library, `timeseriescv`.

Summary

In this chapter, we introduced the challenge of learning from data and looked at supervised, unsupervised, and reinforcement models as the principal forms of learning that we will study in this book to build algorithmic trading strategies. We discussed the need for supervised learning algorithms to make assumptions about the functional relationships that they attempt to learn in order to limit the search space while incurring an inductive bias that may lead to excessive generalization errors.

We presented key aspects of the ML workflow, introduced the most common error metrics for regression and classification models, explained the bias-variance trade-off, and illustrated the various tools for managing the model selection process using cross-validation.

In the following chapter, we will dive into linear models for regression and classification to develop our first algorithmic trading strategies that use ML.