6
The Machine Learning Process

This chapter starts Part 2 of this book, where we'll illustrate how you can use a range of supervised and unsupervised **machine learning (ML)** models for trading. We will explain each model's assumptions and use cases before we demonstrate relevant applications using various Python libraries. The categories of models that we will cover in Parts 2-4 include:

- Linear models for the regression and classification of cross-section, time series, and panel data
- Generalized additive models, including nonlinear 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 build on the material covered so far by demonstrating how to embed these models in a trading strategy that translates model signals into trades, how to optimize portfolio, and how to evaluate strategy performance.

There are several aspects that many of these models and their applications 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'll distinguish between unsupervised and supervised learning and outline use cases for algorithmic trading. We'll contrast supervised regression and classification problems and the use of supervised learning for statistical inference of relationships between input and output data, along with its use for the prediction of future outputs.

We'll 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'll present methods to diagnose sources of errors like overfitting and improve your model's performance.

In this chapter, we will cover the following topics relevant to applying the ML workflow in practice:

- How supervised and unsupervised learning from data works
- Training and evaluating supervised learning models for regression and classification tasks
- How the bias-variance trade-off impacts predictive performance
- How to diagnose and address prediction errors due to overfitting
- Using cross-validation to optimize hyperparameters with a focus on time-series data
- Why financial data requires additional attention when testing out-of-sample

> If you are already quite familiar with ML, feel free to skip ahead and dive right into learning how to use ML models to produce and combine alpha factors for an algorithmic trading strategy. This chapter's directory in the GitHub repository contains the code examples and lists additional resources.

How machine learning from data works

Many definitions of ML 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.
- **Tom 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 of whether the performance of the task improves with experience (Mitchell 1997).

Experience is presented to an algorithm in the form of training data. The principal difference from previous attempts of 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 by humans as was the case, for example, for expert systems prominent in the 1980s.

Recommended textbooks that cover a wide range of algorithms and general applications include James et al (2013), Hastie, Tibshirani, and Friedman (2009), Bishop (2006), and Mitchell (1997).

The challenge – matching the algorithm to the task

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 constitutes a sample of the larger set of phenomena that the algorithm may encounter when performing the task in the future.

The infinite number of functions that could have generated the observed outputs from the given input makes the search process for the true function impossible, without restricting the eligible set of candidates. The types of patterns that an algorithm is capable of learning are limited by the size of its **hypothesis space** that contains the functions it can possibly

represent. It is also limited by the amount of information provided by the sample data.

The size of the hypothesis space varies significantly between algorithms, as we will see in the following chapters. On the one hand, this limitation enables a successful search, and on the other hand, it implies an inductive bias that may lead to poor performance when the algorithm generalizes from the training sample to new data.

Hence, the key challenge becomes 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 learning and generalization given the size of the training data. With more informative data, a model with a larger hypothesis space has a better chance of being 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 for meaningful patterns that generalize well to succeed (Gómez and Rojas 2015).

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.

There are several categories of machine learning tasks that differ by purpose, available information, and, consequently, the learning process itself. The main categories are supervised, unsupervised, and reinforcement learning, and we will review their key differences next.

Supervised learning – teaching by example

Supervised learning is the most commonly used type of ML. We will dedicate most of the chapters in this book to applications 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 at hand. Supervised learning aims to capture a functional input-output relationship from individual samples that reflect this relationship and to apply its learning by making valid statements about new data.

Depending on the field, the output variable is also interchangeably called the label, target, or outcome, as well as the endogenous or left-hand side variable. We will use y_i for outcome observations $i = 1, ..., N$, or y for a (column) vector of outcomes. Some tasks come with several outcomes and are called **multilabel problems**.

The input data for a supervised learning problem is also known as features, as well as exogenous or right-hand side variables. We use x_i for a vector of features with observations $i = 1, ..., N$, or X in matrix notation, where each column contains a feature and each row an observation.

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

The task of learning an input-outcome relationship from data that permits accurate predictions of outcomes for new inputs faces important trade-offs. More complex models have more moving parts that are capable of representing more nuanced relationships. However, they are also more likely to learn random noise particular to the training sample, as opposed to a systematic signal that represents a general pattern. When this happens, we say the model is **overfitting** to the training data. In addition, complex models may also be more difficult to inspect, making it more difficult to understand the nature of the learned relationship or the drivers of specific predictions.

Overly simple models, on the other hand, will miss complex 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 too simple or too complex models may perform poorly beyond the training data.

Unsupervised learning – uncovering useful patterns

When solving an **unsupervised learning** problem, we only observe the features and have no measurements of the outcome. Instead of predicting future outcomes or inferring relationships among variables, unsupervised algorithms aim to identify structure in the input that permits a new representation of the information contained in the data.

Frequently, the measure of success is the contribution of the result to the solution of some other problem. This includes identifying commonalities, or clusters, among observations, or transforming features to obtain a compressed summary that captures relevant information.

The key challenge is that unsupervised algorithms have to accomplish their mission without the guidance provided by outcome information. As a consequence, we are often unable to evaluate the result against a ground truth as in the supervised case, and its quality may be in the eye of the beholder. However, sometimes, we can evaluate its contribution to a downstream task, for example when dimensionality reduction enables better predictions.

There are numerous approaches, from well-established cluster algorithms to cutting-edge deep learning models, and several relevant use cases for our purposes.

Use cases – from risk management to text processing

There are numerous trading use cases for unsupervised learning that we will cover in later chapters:

- Grouping securities with similar risk and return characteristics (see **hierarchical risk parity** in *Chapter 13, Data-Driven Risk Factors and Asset Allocation with Unsupervised Learning*)

- Finding a small number of risk factors driving the performance of a much larger number of securities using **principal component analysis** (*Chapter 13*, *Data-Driven Risk Factors and Asset Allocation with Unsupervised Learning*) or **autoencoders** (*Chapter 19*, *RNN for Multivariate Time Series and Sentiment Analysis*)
- Identifying latent topics in a body of documents (for example, earnings call transcripts) that comprise the most important aspects of those documents (*Chapter 14*, *Text Data for Trading – Sentiment Analysis*)

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

Cluster algorithms – seeking similar observations

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

Cluster algorithms differ in what they assume about how the various groupings were generated and what makes them alike. As a result, they tend to produce alternative types of clusters and should thus be selected based on the characteristics of the data. Some prominent examples are:

- **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 arbitrary shape and 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 – compressing information

Dimensionality reduction produces new data that captures the most important information contained in the source data. Rather than grouping data into clusters while retaining the original data, these algorithms transform the data with the goal of using fewer features to represent the original information.

Algorithms differ with respect to how they transform data and, thus, the nature of the resulting compressed dataset, 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 yields a lower-dimensional representation of the data
- **Autoencoders**: Uses a neural network to compress data nonlinearly with minimal loss of information

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

Reinforcement learning – learning by trial and error

Reinforcement learning (**RL**) is the third type of ML. It centers on an agent that needs to pick an action at each time step, based on information provided by the environment. The agent could be a self-driving car, a program playing a board game or a video game, or a trading strategy operating in a certain security market. You find an excellent introduction in *Sutton and Barto (2018)*.

The agent aims to choose the action that yields the highest reward over time, based on a set of observations that describes the current state of the 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 agent needs to take action right from start and learns in an "online" fashion, one example at a time as it goes along. The learning process follows a trial-and-error approach. This is because the agent needs to manage the trade-off between exploiting a course of action that has yielded a certain reward in the past and exploring new actions that may increase the reward in the future. RL algorithms optimize the agent's learning using dynamical systems theory and, in particular, the optimal control of Markov decision processes with incomplete information.

RL differs from supervised learning, where the 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 must proceed in a continuous fashion as the agent acquires new experience.

However, some of the most notable progress in **artificial intelligence** (**AI**) involves RL, which uses deep learning to approximate functional relationships between actions, environments, and future rewards. It also differs from unsupervised learning because feedback on the actions will be available, albeit with a delay.

RL is particularly suitable for algorithmic trading because the model 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. We will introduce RL approaches to building an algorithmic trading strategy in *Chapter 21*, *Generative Adversarial Networks for Synthetic Time-Series Data*.

The machine learning workflow

Developing an 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:

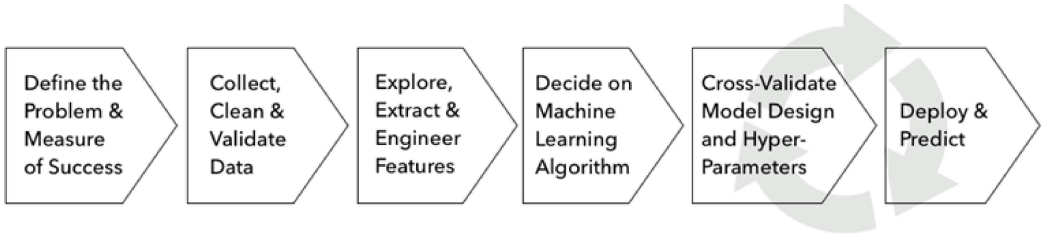

Figure 6.1: Key steps of the machine learning workflow

The process is iterative throughout, and the effort required at different stages will vary according to the project. Generally, however, this process should 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 this 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, which allows us to tackle both regression and classification problems. In its default scikit-learn 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.

The `README` for this chapter on GitHub links to additional resources; see Bhatia and Vandana (2010) for a brief survey.

Framing the problem – from goals to metrics

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

Both inference and prediction tasks rely on metrics to evaluate how well a model achieves its objective. Due to their prominence in practice, we will focus on common objective functions and the corresponding error metrics for predictive models.

We distinguish prediction tasks by the nature of the output: a continuous output variable poses a **regression** problem, a categorical variable implies **classification**, and the special case of ordered categorical variables represents a **ranking** problem.

You can often frame a given problem in different ways. The task at hand may be how to efficiently combine several alpha factors. You could frame this task 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 such as return quintiles.

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. Alternatively, you can use it to predict outputs for unknown inputs.

For algorithmic trading, we can use inference to estimate the statistical association of the returns of an asset with a risk factor. This implies, for instance, assessing how likely this observation is due to noise, as opposed to an actual influence of the risk factor. Prediction, in turn, can be used to forecast the risk factor, which can help predict the asset return and price and 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 the point or interval estimates of metrics.

Inference depends on the assumptions about the process that originally generated the data. 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. Instead, they approach the task of function approximation with fewer restrictions, while treating the data-generating process as a black box.

These models, including decision trees, ensemble models, and neural networks, have gained in popularity because they often outperform on prediction tasks. However, we will see that there have been numerous recent efforts to increase the transparency of complex models. Random forests, for example, have recently gained a framework for statistical inference (Wager and Athey 2019).

Causal inference – correlation does not imply causation

Causal inference aims to identify relationships where 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, while assuming all other variables remain constant.

Statistical inference about relationships among two or more variables produces measures of correlation. Correlation 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. It does this 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 *Chapter 7*, *Linear Models – From Risk Factors to Return Forecasts*.

Regression – popular loss functions and error metrics

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 that we can measure the error in the units of the target variable.

The **root-mean-square of the log of the error** (**RMSLE**) is appropriate when the target is subject to exponential growth. Its asymmetric penalty weighs negative errors less than positive errors. You can also log-trans-

form the target prior to training the model and then use the RMSE, as we'll do in the example later in this section.

The **mean of the absolute errors** (**MAE**) and **median of the absolute errors** (**MedAE**) are symmetric but do not give more weight to larger errors. 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 **R2 score** is also called the coefficient of determination and yields the same outcome if 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 scikit-learn 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`), which we'll will introduce later in this section, and which are illustrated in the accompanying notebook:

Name	Formula	scikit-learn function	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		`median_absolute_error`	`neg_median_absolute_error`
Explained variance		`explained_variance_score`	`explained_variance`
R^2 score		`r2_score`	`r2`

Figure 6.2 shows the various error metrics for the house price regression that we'll compute in the notebook:

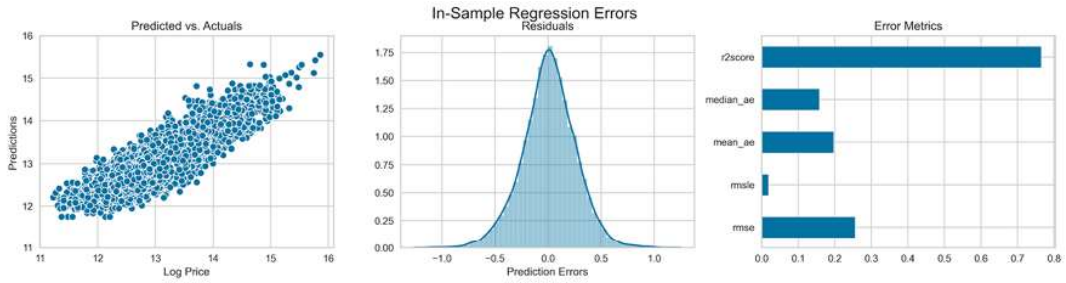

Figure 6.2: In-sample regression errors

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.

Classification – making sense of the confusion matrix

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 using a threshold value.

In the binary case, with a positive and a negative class label, the score typically varies between zero and one or is normalized accordingly. Once the scores are converted into predictions of one class or the other, there

can be four outcomes, since each of the two 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 2×2 confusion matrix that associates actual and predicted classes.

The metrics listed in the following table, such as accuracy, evaluate a model for a given threshold:

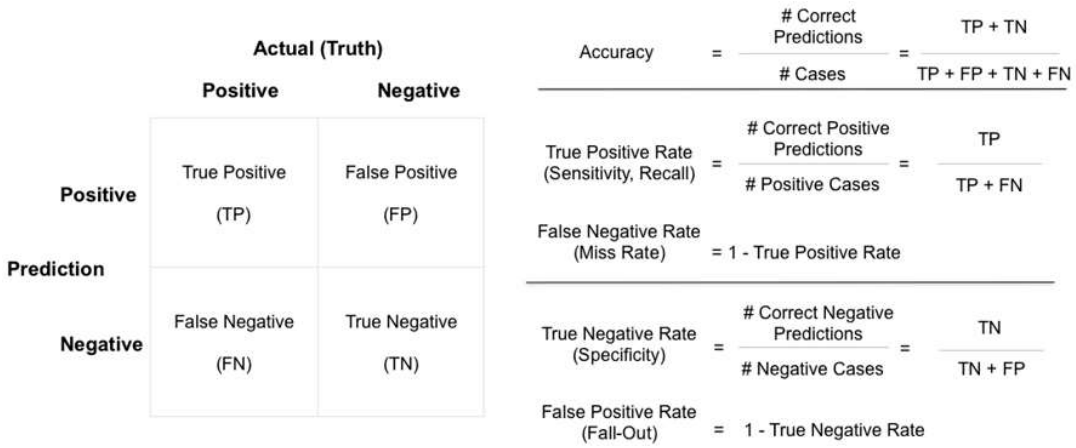

Figure 6.3: Confusion matrix and related error metrics

The classifier usually doesn't output calibrated probabilities. Instead, the threshold used to distinguish positive from negative cases is itself a decision variable that should be optimized, taking into account the costs and benefits of correct and incorrect predictions.

All things equal, a lower threshold tends to imply more positive predictions, with a potentially rising false positive rate, whereas for a higher threshold, the opposite is likely to be true.

Receiver operating characteristics the area under the curve

The **receiver operating characteristics** (**ROC**) curve allows us to visualize, compare, and select classifiers based on their performance. It computes the pairs of **true positive rates** (**TPR**) and **false positive rates**

(**FPR**) that result from using all predicted scores as a threshold to produce class predictions. It visualizes these pairs inside a square with unit side length.

Random predictions (weighted to take into account class imbalance), on average, yield equal TPR and FPR that appear 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 (Fawcett 2006). In addition, the AUC has the benefit of not being sensitive to class imbalances.

Precision-recall curves – zooming in on one class

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 from 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, all while taking into account the relative weights that these two metrics should assume.

Figure 6.4 illustrates the ROC curve and corresponding AUC, alongside the precision-recall curve and the F1 score, which, using equal weights for precision and recall, yields an optimal threshold of 0.37. The chart has been taken from the accompanying notebook, where you can find the code for the KNN classifier that operates on binarized housing prices:

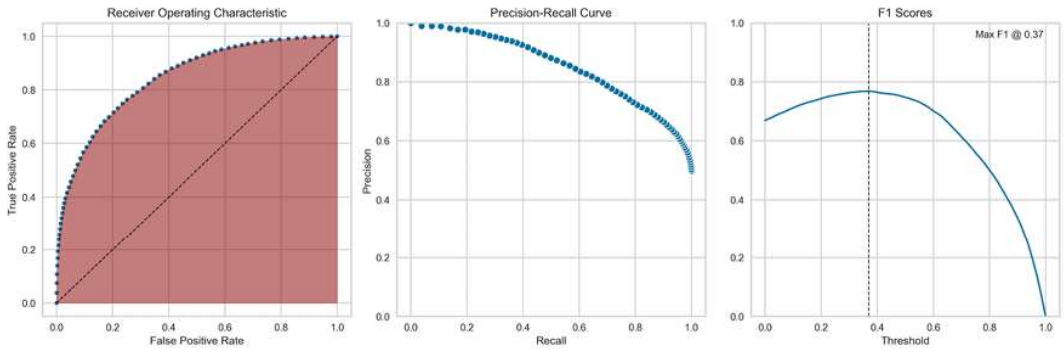

Figure 6.4: Receiver-Operating Characteristics, Precision-Recall Curve, and F1 Scores charts

Collecting and preparing the data

We already addressed important aspects of how to source market, fundamental, and alternative data in *Chapter 2*, *Market and Fundamental Data – Sources and Techniques,* and *Chapter 3*, *Alternative Data for Finance – Categories and Use Cases*. We 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, 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, we may need to assign labels such as sentiment for news articles or timestamps to align it with trading data typically available in a time-series format.

It is also important to store it in a format that enables quick exploration and iteration. We recommend the HDF and parquet formats (see *Chapter 2*, *Market and Fundamental Data – Sources and Techniques*). For data that does not fit into memory and requires distributed processing on several machines, Apache Spark is often the best solution for interactive analysis and machine learning.

Exploring, extracting, and engineering 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 accompanying notebook and shown in *Figure 6.5*:

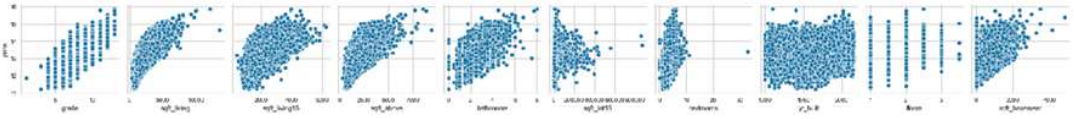

Figure 6.5: Pairwise scatter plots of outcome and features

It also includes **numerical evaluations** ranging from linear metrics like correlation to nonlinear statistics, such as the Spearman rank correlation coefficient that we encountered when we introduced the information coefficient in *Chapter 4*, *Financial Feature Engineering – How to Research Alpha Factors*. There are also information-theoretic measures, such as mutual information, which we'll illustrate in the next subsection.

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 smart 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 central aspect of the ML workflow is best learned through experience. Kaggle is a great place to learn from other data scientists who share their experiences with the 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`, which 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 sklearn documentation for more details). The `mutual_information.ipynb` notebook contains an applica-

tion for the financial data we created in *Chapter 4*, *Financial Feature Engineering – How to Research Alpha Factors*.

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 important goal of the ML process is to obtain an unbiased estimate of this error using a statistically sound and efficient procedure. Key to managing the model design and tuning process is an understanding of how the bias-variance tradeoff relates to under- and overfitting.

The bias-variance trade-off

The prediction errors of an ML model can be broken down into reducible and irreducible parts. The irreducible part is due to random variation (noise) in the data due to, for example, the absence of relevant variables, natural variation, or measurement errors. The reducible part of the generalization error, in turn, can be broken down into errors due to **bias** and **variance**.

Both result from discrepancies 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 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 – a visual example

Figure 6.6 illustrates overfitting by measuring the in-sample error of approximations of a *sine* function by increasingly complex polynomials. More specifically, we draw a random sample with some added noise (n = 30) to learn a polynomial of varying complexity (see the code in the notebook, `bias_variance.ipynb`). The model predicts new data points, and we capture the mean-squared error for these predictions.

The left-hand panel of *Figure 6.6* 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 interval from about $-\pi$ until 2π. 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 as a function of the sample:

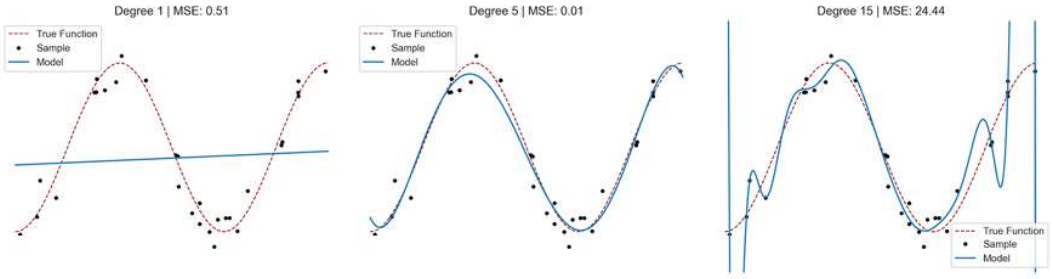

Figure 6.6: A visual example of overfitting with polynomials

How to manage the bias-variance trade-off

To further illustrate the impact of overfitting versus underfitting, we'll try to learn a Taylor series approximation of the *sine* function of the ninth degree with some added noise. *Figure 6.7* shows the in- and-out-of-sample errors and the out-of-sample predictions for polynomials that underfit, overfit, and provide an approximately correct level of flexibility with degrees 1, 15, and 9, respectively, to 100 random samples of the true function.

The left-hand panel shows the distribution of the errors that result from subtracting the true function values from the predictions. The high bias but low variance of an underfit polynomial of degree 1 compares to the low bias but exceedingly high variance of the errors for an overfitting polynomial of degree 15. The underfit polynomial produces a straight line with a poor in-sample fit that 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, on average, by far the best on out-of-sample data.

The right-hand panel of *Figure 6.7* shows the actual predictions rather than the errors to visualize the different types of fit in practice:

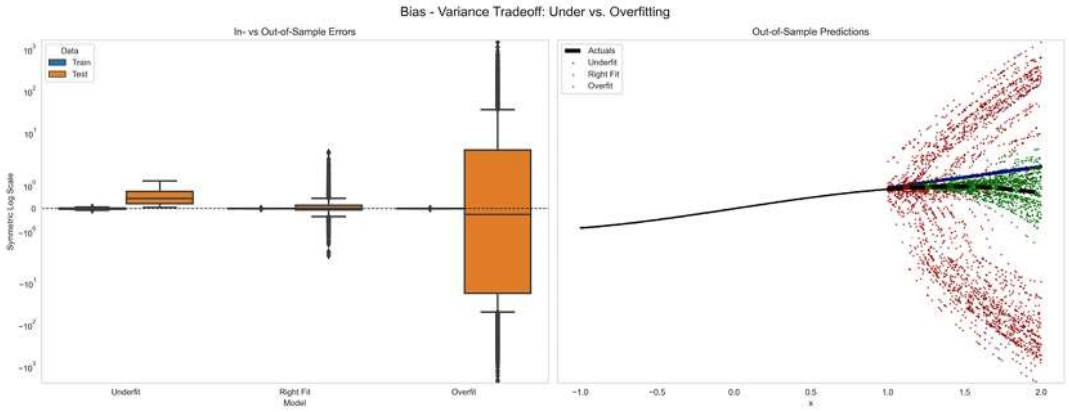

Figure 6.7: Errors and out-of-sample predictions for polynomials of different degrees

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 helps to diagnose the bias-variance trade-off for a given model, and also answer the question of whether increasing the sample size might improve predictive performance. A model with a high bias will have a high but similar training error, both in-sample and out-of-sample. An overfit model will have a very low training but much higher test errors.

Figure 6.8 shows how the out-of-sample error for the overfitted model declines as the sample size increases, suggesting that it may benefit from additional data or tools to limit the model's complexity, such as regularization. Regularization adds data-driven constraints to the model's complexity; we'll introduce this technique in *Chapter 7*, *Linear Models – From Risk Factors to Return Forecasts*.

Underfit models, in contrast, require either more features or need to increase their capacity to capture the true relationship:

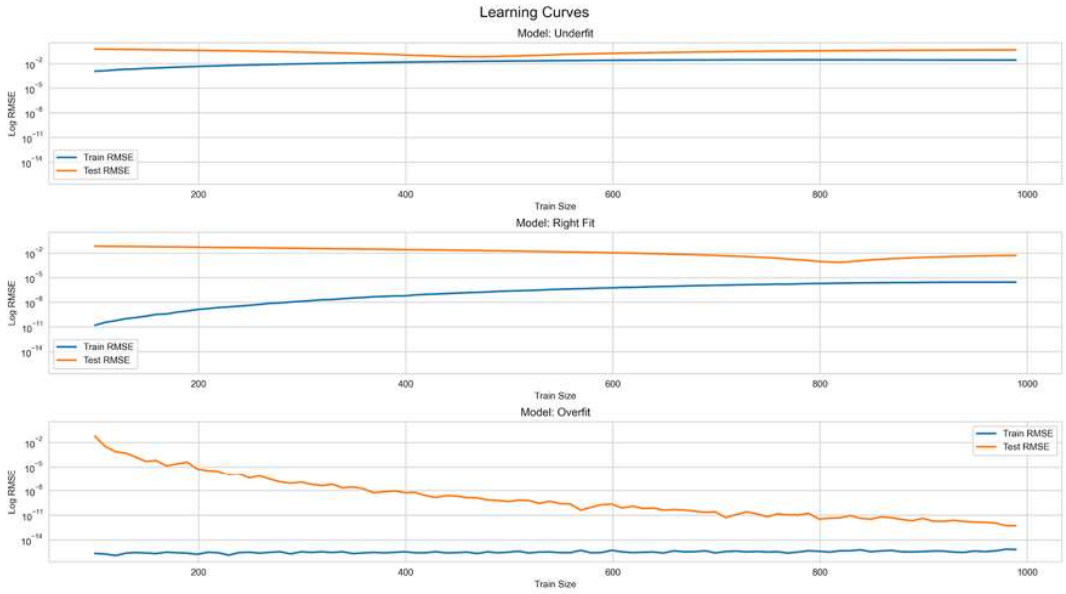

Figure 6.8: Learning curves and bias-variance tradeoff

How to select a model using cross-validation

There are usually several candidate models for your use case, and the task of choosing one of them is known as the **model selection problem**. The goal is to identify the model that will produce the lowest prediction error when given new data.

A good choice requires an unbiased estimate of this generalization error, which, in turn, requires testing the model on data that was not part of model training. Otherwise, the model would have already been able to peek at the "solution" and learn something about the prediction task ahead of time that will inflate its performance.

To avoid this, we only use part of the available data to train the model and set aside another part of the data to validate its performance. The resulting estimate of the model's prediction error on new data will only be unbiased if absolutely no information about the validation set leaks into the training set, as shown in *Figure 6.9*:

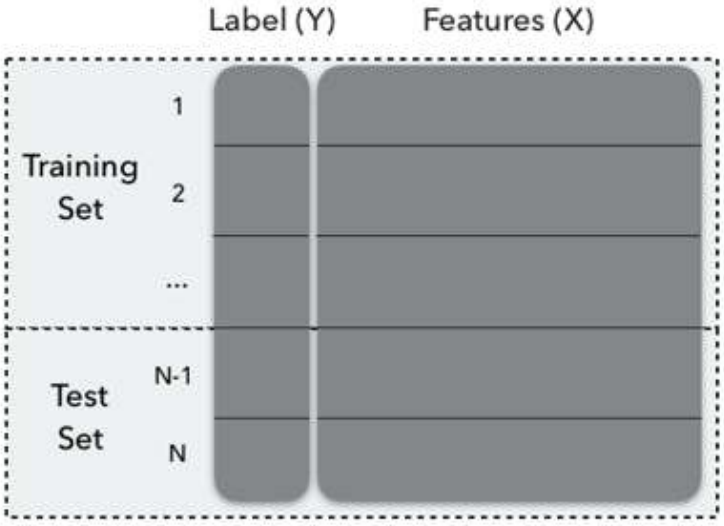

Figure 6.9: Training and test set

Cross-validation (**CV**) is a popular strategy for model selection. The main idea behind CV is to split the data one or several times. This is done 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 to estimate the algorithm's predictive performance. Then, CV selects the algorithm with the smallest estimated error or risk.

Several methods can be used to split the available data. They 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, such as maintaining the ratio between class labels.

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 section and throughout this book, we will emphasize that **time-series data** requires a different approach because it usually does not meet this assumption. Moreover, we need to ensure that splits respect the temporal order to avoid **lookahead bias**. We'll do this by including some information from the future that we aim to predict in the historical training set.

Model selection often involves hyperparameter tuning, which may result in many CV iterations. The resulting validation score of the best-performing model will be subject to **multiple testing bias**, which reflects the sampling noise inherent in the CV process. As a result, it 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.

For this reason, we use a three-way split of the data, as shown in *Figure 6.10*: 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 after, 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:

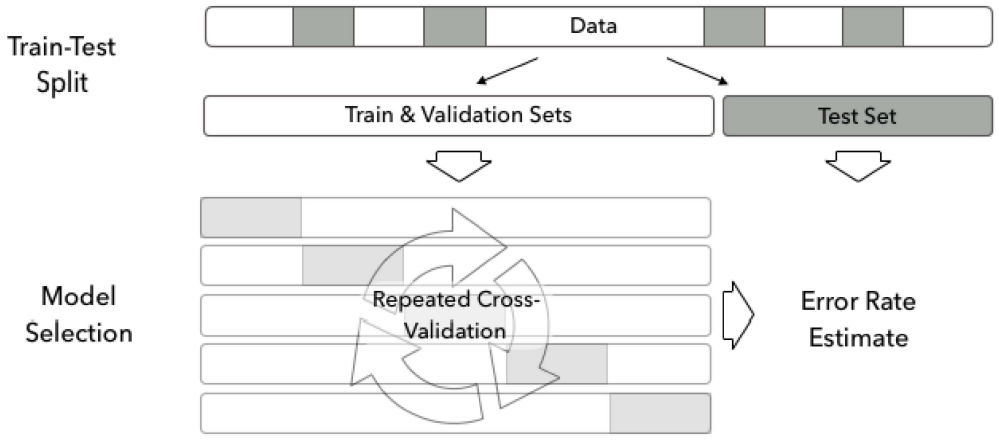

Figure 6.10: Train, validation, and hold-out test set

How to implement cross-validation in Python

We will illustrate various options for splitting data into training and test sets. We'll do this by showing how the indices of a mock dataset with 10 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]
```

Scikit-learn's CV functionality, which we'll demonstrate in this section, can be imported from `sklearn.model_selection`.

For a single split of your data into a training and a test set, use `train_test_split`, where the `shuffle` parameter, by default, ensures the randomized selection of observations. You can ensure replicability by seeding the random number generator by setting `random_state`. There is also a `stratify` parameter, which ensures for a classification problem that the train and test sets will contain approximately the same proportion of each class. The result looks as follows:

```
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 known labels. This method is useful for quick evaluation but is sensitive to the split, and the standard error of the performance measure estimate will be higher.

KFold iterator

The `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 `ShuffleSplit` class 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]
```

Challenges with cross-validation in finance

A key assumption for the cross-validation methods discussed so far is the 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 *Chapter 7, Linear Models – From Risk Factors to Return Forecasts,* and *Chapter 9, Time Series Models for Volatility Forecasts and Statistical Arbitrage*, for more details). `TimeSeriesSplit` in the `sklearn.model_selection` module aims to address the linear order of time-series data.

Time series cross-validation with scikit-learn

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 `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 because returns are computed from prices across multiple periods. In the context of trading strategies, the result of a model's prediction, which may imply taking a position in an asset, can only be known later when this decision is evaluated—for example, when a position is closed out.

The risks include the leakage of information from the test into the training set, which would very likely artificially inflate performance. We need

to address this risk 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. For example, financial disclosures may refer to a certain time period but only become available later. If we include this information too early, our model might do much better in hindsight than it would have under realistic circumstances.

Marcos Lopez de Prado, one of the leading practitioners and academics in the field, has proposed several methods to address these challenges in his book, *Advances in Financial Machine Learning* (2018). Techniques to adapt cross-validation to the context of financial data and trading include:

- **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 Python library, timeseriescv.

Parameter tuning with scikit-learn and Yellowbrick

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 Yellowbrick library extends the scikit-learn 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 – plotting the impact of hyperparameters

Validation curves (see the left-hand panel in *Figure 6.11*) 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 `KNeighborsRegressor`, which only has a single hyperparameter, the number of neighbors is *k*. Note that model complexity increases as the number of neighbors drop because the model can now make predictions for more distinct areas in the feature space.

We can see that the model underfits for values of *k* above 20. The validation error drops as we reduce the number of neighbors and make our model more complex. For values below 20, the model begins to overfit as training and validation errors diverge and average out-of-sample performance quickly deteriorates:

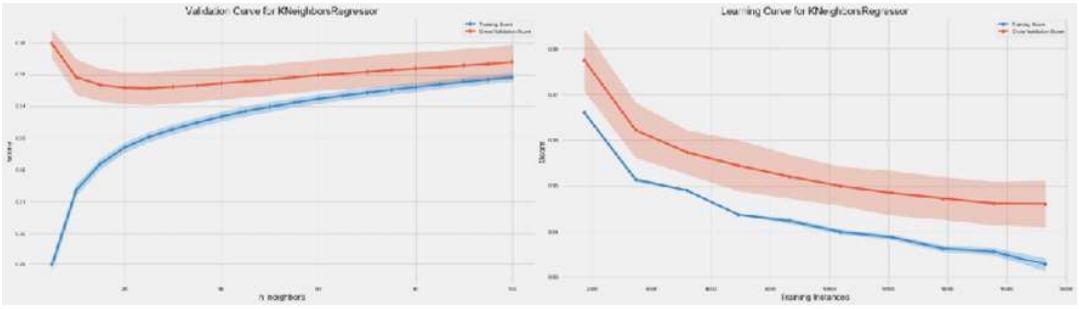

Figure 6.11: Validation and learning curves

Learning curves – diagnosing the bias-variance trade-off

The learning curve (see the right-hand panel of *Figure 6.11* for our house price regression example) helps determine whether a model's cross-validation performance would benefit from additional data, and whether the prediction errors are more driven by bias or by variance.

More data is unlikely to improve performance if training and cross-validation scores converge. 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 provides 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 scikit-learn 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 set often requires some processing prior to cross-validation. Scikit-learn offers the `Pipeline` to also automate any feature-processing steps while using `GridSearchCV`.

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

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. They do this to limit the search space while incurring an inductive bias that may lead to excessive generalization errors.

We presented key aspects of the machine learning 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 machine learning.