

Introduction

Recent advances in **statistics** have been devoted to developing more powerful **automated techniques for predictive modeling**—both **regression** and **classification**. These methods, like those discussed in the previous chapter, are **supervised methods**—they are trained on data where **outcomes are known** and learn to **predict outcomes in new data**. They fall under the umbrella of **statistical machine learning** and are distinguished from **classical statistical methods** in that they are **data-driven** and do **not seek to impose linear or other overall structure on the data**.

The **K-Nearest Neighbors** method, for example, is quite simple: **classify a record in accordance with how similar records are classified**. The most successful and widely used techniques are based on **ensemble learning** applied to **decision trees**. The basic idea of **ensemble learning** is to **use many models to form a prediction**, as opposed to using just a **single model**. **Decision trees** are a **flexible and automatic technique** to learn **rules about the relationships between predictor variables and outcome variables**. It turns out that the **combination of ensemble learning with decision trees** leads to some of the **best performing off-the-shelf predictive modeling techniques**.



The development of many of the techniques in **statistical machine learning** can be traced back to the statisticians **Leo Breiman** (see **Figure 6-1**) at the **University of California at Berkeley** and **Jerry Friedman** at **Stanford University**. Their work, along with that of other researchers at **Berkeley** and **Stanford**, started with the development of **tree models in 1984**. The subsequent development of **ensemble methods of bagging and boosting in the 1990s** established the **foundation of statistical machine learning**.



Machine Learning Versus Statistics

In the context of predictive modeling, what is the difference between machine learning and statistics? There is not a bright line dividing the two disciplines. **Machine learning** tends to be focused more on developing efficient algorithms that scale to large data in order to optimize the predictive model. **Statistics** generally pays more attention to the probabilistic theory and underlying structure of the model. **Bagging, and the random forest** (see “**Bagging and the Random Forest**” on page 259), grew up firmly in the statistics camp. **Boosting** (see “**Boosting**” on page 270), on the other hand, has been developed in both disciplines but receives more attention on the machine learning side of the divide. Regardless of the history, the promise of boosting ensures that it will thrive as a technique in both **statistics and machine learning**.

I. K-Nearest Neighbors

The idea behind K-Nearest Neighbors (KNN) is very simple. For each record to be classified or predicted:

1. Find K records that have similar features (i.e., similar predictor values).
2. For classification, find out what the majority class is among those similar records and assign that class to the new record.

3. For prediction (also called KNN regression), find the average among those similar records, and predict that average for the new record.

Key Terms for K-Nearest Neighbors	
Neighbor	A record that has similar predictor values to another record.
Distance metrics	Measures that sum up in a single number how far one record is from another.
Standardization	Subtract the mean and divide by the standard deviation.
Synonym	Normalization
z-score	The value that results after standardization.
K	The number of neighbors considered in the nearest neighbor calculation.

KNN is one of the **simpler prediction/classification techniques**: there is **no model to be fit** (as in **regression**). This doesn't mean that using **KNN** is an **automatic procedure**. The **prediction results** depend on how the **features are scaled**, how **similarity is measured**, and how **big K is set**. Also, **all predictors must be in numeric form**. We will illustrate how to use the **KNN method** with a **classification example**.

A Small Example: Predicting Loan Default

Table 6-1 shows a few records of personal loan data from LendingClub. LendingClub is a leader in peer-to-peer lending in which pools of investors make personal loans to individuals. The goal of an analysis would be to predict the outcome of a new potential loan: paid off versus default.

```
outcome  payment_inc_ratio  dti
0  target           9.00000  22.50
1  default          5.46933  21.33
2  paid off         6.90294   8.97
3  paid off        11.14800   1.83
4  default          3.72120  10.81
the predicted : ['paid off']
```

Consider a very simple model with just **two predictor variables**: **dti**, which is the **ratio of debt payments (excluding mortgage) to income**, and **payment_inc_ratio**, which is the **ratio of the loan payment to income**. Both ratios are multiplied by 100. Using a **small set of 200 loans**, **loan200**, with **known binary outcomes** (default or no-default, specified in the predictor **outcome200**), and with **K set**

to 20, the **KNN estimate** for a **new loan to be predicted, newloan**, with **dti = 22.5** and **payment_inc_ratio = 9** can be calculated using the **scikit-learn package**, which provides a **fast and efficient implementation of KNN in Python**.

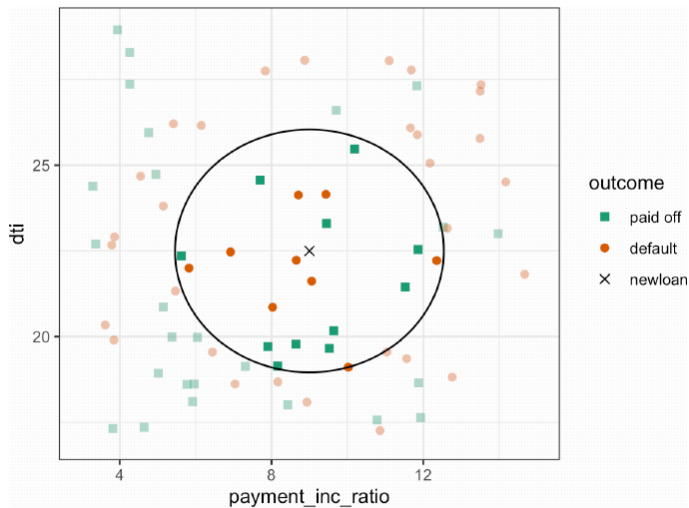


Figure 6-2. KNN prediction of loan default using two variables: debt-to-income ratio and loan-payment-to-income ratio

Figure 6-2 gives a visual display of this example. The new loan to be predicted is the cross in the middle. The squares (paid off) and circles (default) are the training data. The large black circle shows the boundary of the nearest 20 points. In this case, 9 defaulted loans lie within the circle, as compared with 11 paid-off loans. Hence the predicted outcome of the loan is paid off. Note that if we consider only three nearest neighbors, the prediction would be that the loan defaults.

While the output of **KNN for classification** is typically a **binary decision**, such as **default** or **paid off** in the loan data, **KNN routines** usually offer the opportunity to output a **probability (propensity) between 0 and 1**. The **probability** is based on the **fraction of one class in the K nearest neighbors**. In the preceding example, this **probability of default** would have been estimated at **9/20, or 0.45**. Using a **probability score** lets you use **classification rules other than simple majority votes (probability of 0.5)**. This is especially important in problems with **imbalanced classes**; see “Strategies for Imbalanced Data” on page 230. For example, if the goal is to **identify members of a rare class**, the **cutoff would typically be set below 50%**. One common approach is to **set the cutoff at the probability of the rare event**.

I.1 Distance Metrics

Similarity (nearness) is determined using a **distance metric**, which is a **function that measures how far two records** (x_1, x_2, \dots, x_p) and (u_1, u_2, \dots, u_p) are from one another. The **most popular distance metric** between two vectors is **Euclidean distance**. To **measure the Euclidean distance** between two vectors, **subtract one from the other, square the differences, sum them, and take the square root**:

$$\sqrt{(x_1 - u_1)^2 + (x_2 - u_2)^2 + \dots + (x_p - u_p)^2}.$$

Another common distance metric for numeric data is Manhattan distance:

$$|x_1 - u_1| + |x_2 - u_2| + \dots + |x_p - u_p|$$

Euclidean Distance



Manhattan Distance



Euclidean distance corresponds to the straight-line distance between two points (e.g., as the crow flies). Manhattan distance is the distance between two points traversed in a single direction at a time (e.g., traveling along rectangular city blocks). For this reason, Manhattan distance is a useful approximation if similarity is defined as point-to-point travel time.

When measuring the distance between two vectors, variables that are on a **large numerical scale** tend to **dominate the distance calculation**. For example, in the loan data, variables such as **income** and **loan amount**, which are measured in **tens or hundreds of thousands**, would largely determine the distance between two observations. In contrast, **ratio variables** like debt-to-income or payment-to-income ratios, which typically range from 0 to 100, would contribute very little to the distance and be almost ignored.

This imbalance can distort methods like **KNN**, which rely directly on distance. To address this issue, the data are commonly **standardized** so that all variables are placed on a **comparable scale**. Standardization (also called normalization or z-score scaling) ensures that each variable contributes more equally to the distance calculation, preventing large-scale variables from overwhelming smaller-scale but potentially important features.



Other Distance Metrics

There are numerous other metrics for measuring distance between vectors. For numeric data, **Mahalanobis distance** is attractive since it accounts for the correlation between two variables. This is useful since if two variables are highly correlated, Mahalanobis will essentially treat these as a single variable in terms of distance. Euclidean and Manhattan distance do not account for the correlation, effectively placing greater weight on the attribute that underlies those features. **Mahalanobis distance** is the Euclidean distance between the principal components (see “Principal Components Analysis” on page 284). The downside of using Mahalanobis distance is increased computational effort and complexity; it is computed using the **covariance matrix** (see “Covariance Matrix” on page 202).

I.2 One Hot encoder

Table 6-1. A few records and columns for LendingClub loan data

Outcome	Loan amount	Income	Purpose	Years employed	Home ownership	State
Paid off	10000	79100	debt_consolidation	11	MORTGAGE	NV
Paid off	9600	48000	moving	5	MORTGAGE	TN
Paid off	18800	120036	debt_consolidation	11	MORTGAGE	MD
Default	15250	232000	small_business	9	MORTGAGE	CA
Paid off	17050	35000	debt_consolidation	4	RENT	MD
Paid off	5500	43000	debt_consolidation	4	RENT	KS

The loan data in **Table 6-1** includes several **factor (string) variables**. Most **statistical and machine learning models** require this type of variable to be **converted to a series of binary dummy variables** conveying the same information, as in **Table 6-2**. Instead of a **single variable** denoting the **home occupant status** as “owns with a mortgage,” “owns with no mortgage,” “rents,” or “other,” we end up with **four binary variables**.

The first would be “**owns with a mortgage—Y/N**,” the second would be “**owns with no mortgage—Y/N**,” and so on. This **one predictor**, home occupant status, thus yields a **vector with one 1 and three 0s** that can be used in **statistical and machine learning algorithms**. The phrase **one hot encoding** comes from **digital circuit terminology**, where it describes **circuit settings in which only one bit is allowed to be positive (hot)**.

Table 6-2. Representing home ownership factor data in **Table 6-1** as a numeric dummy variable



In linear and logistic regression, one hot encoding causes problems with multicollinearity; see “**Multicollinearity**” on page 172. In such cases, one dummy is omitted (its value can be inferred from the other values). This is not an issue with KNN and other methods discussed in this book.

I.3 Standardization (Normalization, z-score)

In **measurement**, we are often not so much interested in “**how much**” but in “**how different from the average**.”

$$z = \frac{x - \bar{x}}{s}$$

Standardization, also called **normalization**, puts **all variables on similar scales** by **subtracting the mean and dividing by the standard deviation**; in this way, we ensure that a **variable does not overly influence a model** simply due to the **scale of its original measurement**.



Normalization in this statistical context is not to be confused with **database normalization**, which is the removal of redundant data and the verification of data dependencies.

For KNN and a few other procedures (e.g., principal components analysis and clustering), it is essential to consider standardizing the data prior to applying the procedure. To illustrate this idea, KNN is applied to the loan data using `dti` and `payment_inc_ratio` (see “A Small Example: Predicting Loan Default” on page 239) plus two other variables: `revol_bal`, the total revolving credit available to the applicant in dollars, and `revol_util`, the percent of the credit being used.

The new record to be predicted is shown here:

```
newloan
payment_inc_ratio dti revol_bal revol_util
1             2.3932  1      1687         9.4
```

The magnitude of `revol_bal`, which is in dollars, is much bigger than that of the other variables.

Following the model fit, we can use the `kneighbors` method to identify the five closest rows in the training set with scikit-learn.

```
Distances:
[1.55563118  5.64040694  7.13883816  8.84224262  8.97277387]

Nearest neighbors:
      payment_inc_ratio  dti  revol_bal  revol_util
35536             1.47212  1.46      1686         10.0
33651             3.38178  6.37      1688           8.4
25863             2.36303  1.39      1691           3.5
42953             1.28160  7.14      1684           3.9
43599             4.12244  8.98      1684           7.2
```

The value of `revol_bal` in these neighbors is very close to its value in the new record, but the other predictor variables are all over the map and essentially play no role in determining neighbors.

“Prepare training data (exclude the new loan)” means You do NOT use the loan you want to predict as part of the data used to train or compare against itself.

```
Distances to the 5 nearest neighbors:
[0.05750762  0.09802148  0.09887142  0.10540438  0.11645144]

Predictor values of the 5 nearest neighbors:
      payment_inc_ratio  dti  revol_bal  revol_util
2080             2.61091  1.03      1218           9.7
1438             2.34343  0.51       278           9.9
30215            2.71200  1.34      1075           8.5
28542            2.39760  0.74      2917           7.4
44737            2.34309  1.37       488           7.2
```

The five nearest neighbors are much more alike in all the variables, providing a more sensible result. Note that the results are displayed on the original scale, but KNN was applied to the scaled data and the new loan to be predicted.



Using the z-score is just one way to rescale variables. Instead of the mean, a more robust estimate of location could be used, such as the median. Likewise, a different estimate of scale such as the inter-quartile range could be used instead of the standard deviation. Sometimes, variables are “squashed” into the 0–1 range. It’s also important to realize that scaling each variable to have unit variance is somewhat arbitrary. This implies that each variable is thought to have the same importance in predictive power. If you have subjective knowledge that some variables are more important than others, then these could be scaled up. For example, with the loan data, it is reasonable to expect that the payment-to-income ratio is very important.



Normalization (standardization) does not change the distributional shape of the data; it does not make it normally shaped if it was not already normally shaped (see “Normal Distribution” on page 69).

I.4. Choosing K

The choice of K is very important to the performance of KNN. The simplest choice is to set $K = 1$, known as the 1-nearest neighbor classifier. The prediction is intuitive: it is based on finding the data record in the training set most similar to the new record to be predicted. Setting $K = 1$ is rarely the best choice; you’ll almost always obtain superior performance by using $K > 1$ -nearest neighbors.

In KNN, the value of **K** determines how much detail the model pays attention to.

When **K is too small**, the model focuses on only a few nearby points. This makes it very sensitive to noise, so random or unusual observations can strongly influence the result. In this case, the model fits the training data too closely, leading to **overfitting**.

For example, a single borrower who defaulted due to an exceptional situation could cause a new, otherwise safe loan to be classified as risky.

As **K increases**, the model considers more neighbors, which smooths the decision boundary. This averaging effect reduces the influence of noise and lowers the risk of overfitting.

However, if **K becomes too large**, the model starts to average over many distant points. Important local patterns are lost, and the model can no longer capture the local structure of the data — one of KNN's key strengths.

For example, a low-risk borrower may be misclassified because many unrelated high-risk borrowers are included in the decision.

In short, **small K overfits, large K oversmooths**, and the best K is a balance between the two.

The K that best balances between overfitting and oversmoothing is typically determined by accuracy metrics and, in particular, accuracy with holdout or validation data. There is no general rule about the best K—it depends greatly on the nature of the data. For highly structured data with little noise, smaller values of K work best.

Borrowing a term from the **signal processing** community, this type of data is sometimes referred to as having a **high signal-to-noise ratio (SNR)**.

Examples of data with a typically **high SNR** are data sets for **handwriting** and **speech recognition**.

For **noisy data** with **less structure** (data with a **low SNR**), such as the **loan data**, **larger values of K** are appropriate.

Typically, values of **K fall in the range 1 to 20**. Often, an **odd number** is chosen to **avoid ties**.

① What is signal and noise?

- **Signal** = the useful information you want to detect or learn from.
- **Noise** = random errors, irrelevant details, or disturbances that make it harder to see the signal.

Think of it like listening to someone talk:

- If you're in a quiet room, you hear their words clearly → **high signal, low noise**.
- If you're in a loud party, it's hard to hear → **signal is drowned in noise**, low SNR.

② High SNR (clear, structured data)

- The useful patterns **stand out clearly**.
- Small K works well because each point is reliable and not affected much by random noise.

Examples:

- **Handwriting recognition** → letters follow clear shapes
- **Speech recognition** → words have predictable patterns

③ Low SNR (noisy, unstructured data)

- Patterns are **less obvious** because noise is strong.
- Large K helps smooth out the random variations by averaging many neighbors.

Example:

- **Loan data** → borrowers' financial situations vary a lot, some unusual events happen (like sudden default).
- A single point is unreliable, so you need more neighbors to get a stable prediction.



Bias-Variance Trade-off Explained

In model building, there's always a tension between **overfitting** and **oversmoothing**—this is called the **bias-variance trade-off**.

- **Variance** is the error caused by sensitivity to the training data.
Example: If you train a KNN model on one set of loan data and then train it on a slightly different set, the predictions might change a lot. That's a high variance.
- **Bias** is the error caused by oversimplifying the real-world problem.
Example: If your model assumes all borrowers are the same and ignores financial features, it will make consistently wrong predictions. This error won't improve by adding more data.

When a model is very flexible (like $K = 1$ in KNN), it may **overfit**, capturing noise in the training data. This increases **variance**. You can reduce variance by simplifying the model (like using a larger K), but then **bias** may increase because the model can no longer capture fine details.

The key is finding a balance, often using cross-validation to pick a model that generalizes well to new data. "Cross-Validation" on page 155 for more details.

I.5. KNN as Feature Engine

KNN became popular because it's **simple and intuitive** — you just look at the closest neighbors to classify a new point. However, on its own, KNN usually **doesn't perform as well** as more advanced classification methods.

In practice, KNN is often used to **add "local knowledge"** to other models in a staged approach:

1. **Run KNN first:** For each record, KNN provides a classification or a **quasi-probability** for each class. *Example:* For a new loan, KNN might say there's a 70% chance it's "safe" based on its 5 nearest neighbors.
2. **Use KNN results as a feature:** This output is added as a new variable to the dataset. Then, another classification method (like logistic regression or a decision tree) is applied, using both the **original predictors** and the **KNN-derived feature**. *Example:* The logistic regression now knows both the original financial ratios **and** the local KNN probability, improving its prediction.

At first you might wonder whether this process, since it uses some **predictors twice**, causes a problem with **multicollinearity** (see "Multicollinearity" on page 172). This is **not an issue**, since the information


being incorporated into the **second-stage model** is **highly local**, derived only from a **few nearby records**, and is therefore **additional information** and **not redundant**.

For example, think of **real estate pricing**: a realtor sets a home's price by looking at **similar homes recently sold**, called "**comps**". This is basically a **manual version of KNN** — finding the closest neighbors and using their values to make a prediction.

We can mimic this in a statistical model by using **KNN to create a new feature**:

- Existing predictors: location, square footage, type of structure, lot size, bedrooms, bathrooms, etc.
- KNN predicts a value (the sale price) based on the **average of the K nearest homes**.
- This predicted value becomes a **new predictor variable** in the model, just like the realtor's comps.

Since the target is numeric, we use **KNN regression** (averaging neighbors) instead of classification (majority vote).

- Count of votes:
 - Safe = 3 votes
 - Default = 2 votes  **Majority vote result:** Safe → the new loan is classified as "Safe."

Coming back to 'load_data.csv'. Similarly, for the loan data, we can create features that represent different aspects of the loan process. The result is a feature that predicts the likelihood a borrower will default based on his credit history.

Metric	Value	Meaning
count	45342	There are 45,342 borrowers in the dataset
mean	0.499	On average, the predicted probability of default is about 50%
std	0.129	Most scores are spread within ~0.13 around the mean (moderate variation)
min	0.05	The lowest risk borrower has a 5% predicted probability of default
25%	0.4	25% of borrowers have a predicted risk \leq 40%
50% (median)	0.5	Half the borrowers have a predicted risk \leq 50%
75%	0.6	75% of borrowers have a predicted risk \leq 60%
max	1	The highest risk borrower has a predicted probability of 100%

```
count    45342.000000
mean      0.498902
std       0.128736
min       0.050000
25%      0.400000
50%      0.500000
75%      0.600000
max       1.000000
```

Key Ideas

- **K-Nearest Neighbors (KNN)** classifies a record by assigning it to the class that similar records belong to.
- **Similarity (distance)** is determined by Euclidian distance or other related metrics.
- The number of nearest neighbors to compare a record to, K , is determined by how well the algorithm performs on training data, using different values for K .
- Typically, the predictor variables are standardized so that variables of large scale do not dominate the distance metric.
- **KNN** is often used as a first stage in predictive modeling, and the predicted value is added back into the data as a **predictor** for second-stage (non-KNN) modeling.

II. Tree Models

Tree models, also called **Classification and Regression Trees (CART)**, **decision trees**, or just **trees**, are an **effective** and **popular classification (and regression) method** initially developed by **Leo Breiman** and others in **1984**. Tree models, and their more powerful descendants **random forests** and **boosted trees** (see “Bagging and the Random Forest” on page 259 and “Boosting” on page 270), form the basis for the **most widely used** and **powerful predictive modeling tools** in **data science** for **regression** and **classification**.

Key Terms for Trees

Recursive partitioning

Repeatedly dividing and subdividing the data with the goal of making the outcomes in each final subdivision as **homogeneous** as possible.

Split value

A predictor value that divides the records into those where that predictor is less than the split value, and those where it is more.

Node

In the **decision tree**, or in the set of corresponding branching rules, a node is the graphical or rule representation of a split value.

Leaf

The end of a set of **if-then rules**, or branches of a tree—the rules that bring you to that leaf provide one of the classification rules for any record in a tree.

Loss

The number of misclassifications at a stage in the splitting process; the more losses, the more impurity.

Impurity

The extent to which a mix of classes is found in a subpartition of the data (the more mixed, the more impure).

Synonym

Heterogeneity

Antonyms

Homogeneity, purity

Pruning

The process of taking a fully grown tree and progressively cutting its branches back to reduce overfitting.

II.1. A Simple Example

The `sklearn.tree.DecisionTreeClassifier` provides an implementation of a decision tree. The `dmdba` package provides a convenience function to create a visualization inside a Jupyter notebook.

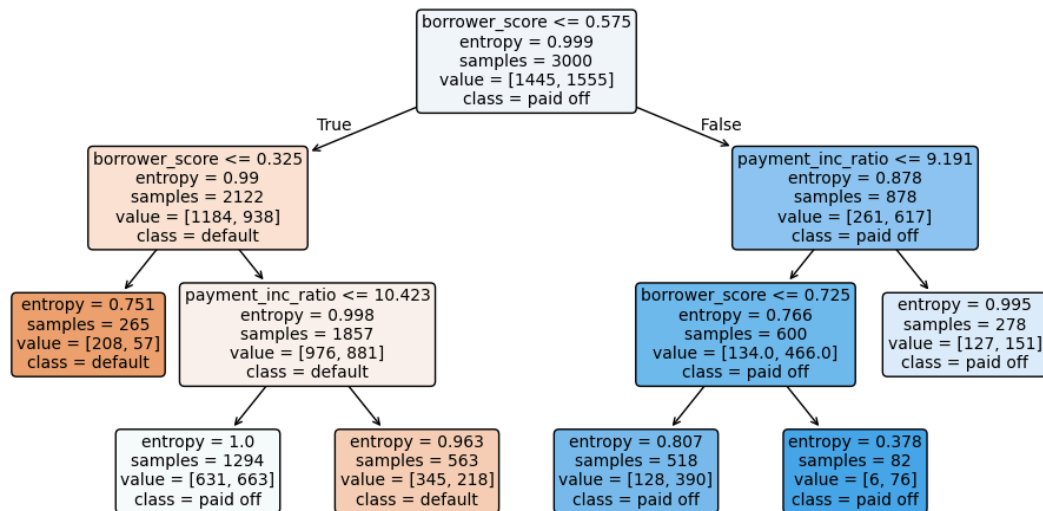


Figure 6-3. The rules for a simple tree model fit to the loan data

These **classification rules** are determined by traversing through a **hierarchical tree**, starting at the **root** and moving **left if the node is true** and **right if not**, until a **leaf** is reached. Typically, the **tree** is plotted **upside-down**, so the **root** is at the **top** and the **leaves** are at the **bottom**. For example, if we get a **loan** with **borrower_score** of 0.6 and a **payment_inc_ratio** of 8.0, we end up at the **leftmost leaf** and **predict** the **loan will be paid off**.

1 Tree Depth and Indentation

- In a **decision tree**, each **split** creates a **new level** in the hierarchy.
- The **indent** in the tree printout shows how **deep a node is**: the more indented, the further down the node is in the tree.
- **Why it matters**: deeper nodes represent **smaller, more specific partitions** of the data.

Example:

node=0

node=1

node=2

- **Node 0** → root (all data)
- **Node 1** → first split (subset of data)
- **Node 2** → deeper split (smaller subset)

2 Nodes and Provisional Classification

- Each **node** makes a **temporary prediction** (provisional classification) based on the **majority class** of records in that node.
- It doesn't look at new data yet — it just sees what's in the current partition.

Example:

- Node 2 has 878 loans, 617 are "paid off," 261 are "default."
- **Provisional classification:** "paid off" (the majority)
- **Loss:** 261 (number of misclassifications in that node)

3 Loss and Misclassifications

- **Loss** = number of records in that node that **don't match the provisional class**.
- **Why it matters:** a node with a **high loss** is less pure; a node with **zero loss** is perfectly homogeneous (all records belong to the same class).

Example:

- Node 2 → 261 misclassifications out of 878 → ~30% of records are "wrong" for this provisional class.

4 Class Proportions in Parentheses

- The **numbers in parentheses** show the **distribution of classes** in that node.
- Format: `[proportion_paid_off, proportion_default]`

Example:

- Node 13 → predicts **default**, values: `(0.38, 0.62)` → 62% of loans are actually defaulting.
- This shows that even though the node predicts default, there's still a portion of loans that are paid off (38%).

5 Putting it all together

Think of a decision tree like a **flowchart for decisions**:

1. Start at the **root** (all borrowers).
2. Ask a **yes/no question** (e.g., `borrower_score <= 0.575`).
3. Move **left or right** depending on the answer.
4. Keep splitting until you reach a **leaf node** (smallest group).
5. The **leaf node's majority class** is the prediction, but the **loss** and **class proportions** tell you **how confident** the prediction is.

✓ In short:

- **Depth** → how specific the node is
- **Node** → temporary prediction based on majority class
- **Loss** → misclassified records in that node

II.2. The Recursive Partitioning Algorithm

The algorithm to construct a decision tree, called recursive partitioning, is straightforward and intuitive. The data is repeatedly partitioned using predictor values that do the best job of separating the data into relatively homogeneous partitions. Figure 6-4 shows the partitions created for the tree in Figure 6-3.

The first rule, depicted by rule 1, is $\text{borrower_score} \geq 0.575$ and segments the right portion of the plot. The second rule is $\text{borrower_score} < 0.375$ and segments the left portion.

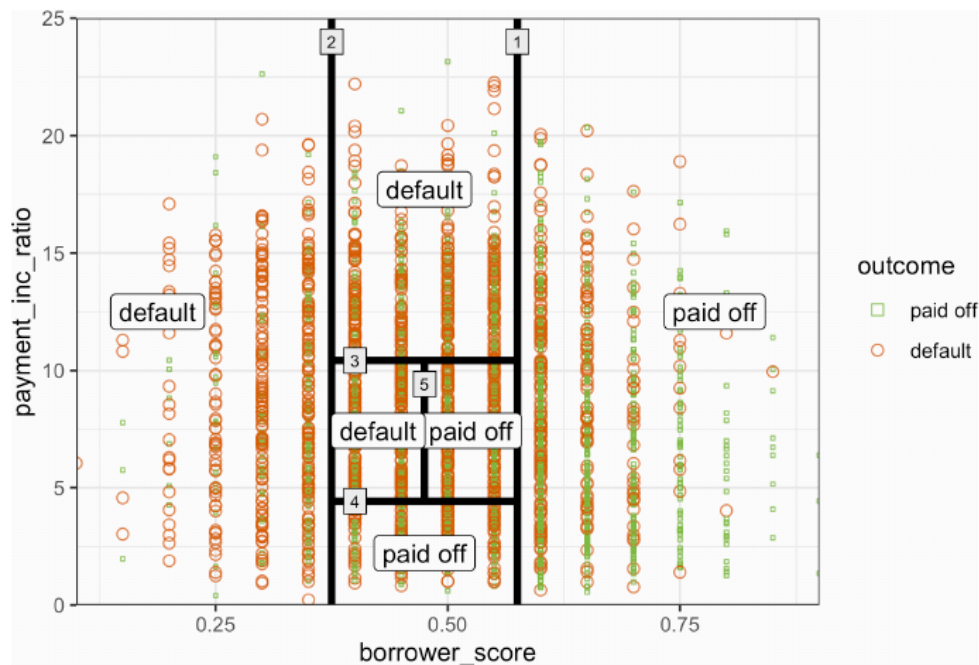


Figure 6-4. The first three rules for a simple tree model fit to the loan data

- **x-axis:** `borrower_score`
- **y-axis:** `payment_inc_ratio`

Each **rule** corresponds to a **vertical or horizontal cut** that partitions the plot into regions. Inside each region, the tree predicts either **paid off** or **default**.

`borrower_score ≥ 0.575`

- This is the **first split at the root of the tree**.
- Graphically, it is the **vertical black line** at `borrower_score = 0.575`.

What it does:

- Everything **to the right of this line** becomes one region.
- The tree immediately classifies most of these loans as **paid off**.

#=====

Rule 2: `borrower_score < 0.375`

- This is applied **only to the left side** of Rule 1.
- It creates another **vertical cut**, further left.

What it does:

- Everything **to the left of 0.375** becomes its own region.
- This region is mostly classified as **default**.

Why this makes sense:

- Very **low borrower scores** indicate high risk.
- The left side of the plot contains **many orange points (defaults)**.

#=====

What remains in the middle?

After Rules 1 and 2:

- The **middle band** ($0.375 \leq \text{borrower_score} < 0.575$) is still mixed.
- That's why the tree adds **more rules** (horizontal cuts using `payment_inc_ratio`) to refine predictions there.

Steps logic behind The Recursive Partitioning Algorithm

Step 1: Start with the full dataset

Suppose:

- **Y** = loan outcome (0 = default, 1 = paid off)
- **Predictors:**
 - `borrower_score`
 - `payment_inc_ratio`

We begin with **A = all loans**.

At this point, A is mixed: some loans default, some are paid off.

Step 2: Try all possible splits (this is the key idea)

Example: trying `borrower_score`

The algorithm:

1. Takes `borrower_score`
2. Tries **many possible cut points** (values of `sj`), for example: 0.30, 0.40, 0.50, 0.60

For each split:

- Left group: `borrower_score < sj`
- Right group: `borrower_score ≥ sj`

Then it asks:

“After this split, are the two groups more homogeneous than before?”

Example split

Try `borrower_score < 0.575`:

- **Left side:** mixed defaults and paid-off loans
- **Right side:** mostly paid-off loans

This split produces **high homogeneity**, especially on the right.

So this is a **good candidate**.

Step 3: Compare all predictors

The algorithm does the same for:

- `payment_inc_ratio`
- Any other predictor `Xj`

Each variable is tested with many possible split values.

👉 The algorithm then **chooses the variable and split value** that produces the **best overall homogeneity**.

Step 4: Recursive part (repeat on subgroups)

Now the dataset is split into:

- **A1:** `borrower_score < 0.575`
- **A2:** `borrower_score ≥ 0.575`

Apply the same logic again:

- On **A2**: Maybe it's already very pure → stop splitting.
- On **A1**: Still mixed → try splitting again.

Example:

- Try `borrower_score < 0.375` on A1
- This isolates very risky borrowers → mostly defaults

Step 5: Continue until stopping condition

The algorithm keeps splitting until:

- No split significantly improves homogeneity, or
- A minimum node size or impurity threshold is reached

At that point, the node becomes a **leaf**.

Final result as in figure 6-4

- The data is divided into **regions (partitions)**
- Each region predicts:
 - **0 (default)** or **1 (paid off)**
- The prediction is based on the **majority vote** within that partition

Example:

- A region with 80% defaults → predict *default*
- A region with 70% paid-off → predict *paid off*



In addition to a binary 0/1 prediction, tree models can produce a probability estimate based on the number of 0s and 1s in the partition. The estimate is simply the sum of 0s or 1s in the partition divided by the number of observations in the partition:

$$\text{Prob}(Y = 1) = \frac{\text{Number of 1s in the partition}}{\text{Size of the partition}}$$

The estimated $\text{Prob}(Y = 1)$ can then be converted to a binary decision; for example, set the estimate to 1 if $\text{Prob}(Y = 1) > 0.5$.

II.3. Measuring Homogeneity or Impurity

When a decision tree splits data, it asks one key question: “After this split, are the groups more uniform in their outcome?” → That uniformity is called homogeneity (or purity).

Intuition first

- A node is **pure** if almost everyone in it has the **same outcome**.
- A node is **impure** if outcomes are **mixed**.
- The tree prefers splits that **increase homogeneity**.

Example (loan data):

- Node A: 95% *paid off*, 5% *default* → **high homogeneity**
- Node B: 50% *paid off*, 50% *default* → **high impurity**

Tree models recursively create partitions (sets of records), A , that predict an outcome of $Y = 0$ or $Y = 1$. You can see from the preceding algorithm that we need a way to measure homogeneity, also called class purity, within a partition. Or equivalently, we need to measure the impurity of a partition. The accuracy of the predictions is the proportion p of misclassified records within that partition, which ranges from 0 (perfect) to 0.5 (purely random guessing).

It turns out that **accuracy is not a good measure for impurity**. Instead, two common measures for impurity are the **Gini impurity** and **entropy of information**. While these (and other) **impurity measures apply to classification problems with more than two classes**, we focus on the **binary case**.

The Gini impurity for a set of records A is: $I(A) = p(1 - p)$

The entropy measure is given by: $I(A) = -p \log_2(p) - (1 - p) \log_2(1 - p)$

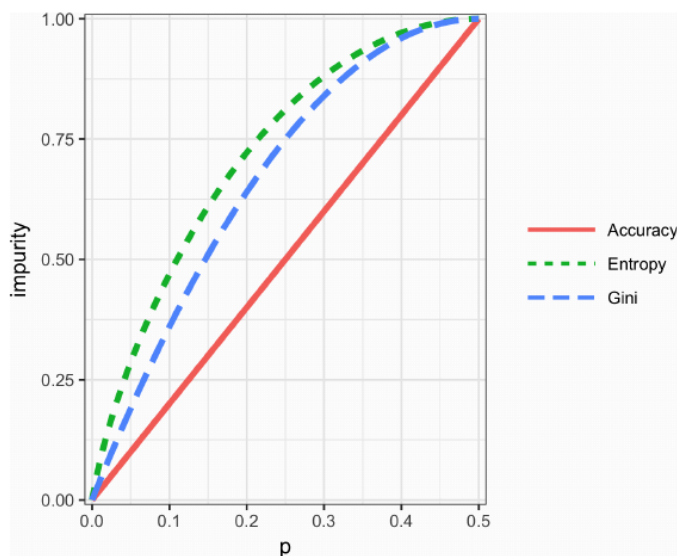


Figure 6-5. Gini impurity and entropy measures

Figure 6-5 shows that Gini impurity (rescaled) and entropy measures are similar, with entropy giving higher impurity scores for moderate and high accuracy rates.



Gini Coefficient

Gini impurity is not to be confused with the Gini coefficient. They represent similar concepts, but the Gini coefficient is limited to the binary classification problem and is related to the AUC metric (see "AUC" on page 226).

1 Gini impurity (most common)

Measures how often a random record would be misclassified.

- **Gini = 0** → perfectly pure
- Higher Gini → more mixed

2 Entropy (used in your code)

Measures uncertainty (from information theory).

- **Entropy = 0** → perfectly pure
- Higher entropy → more uncertainty

For each proposed partition of the data, impurity is measured for each of the partitions that result from the split. A weighted average is then calculated, and whichever partition (at each stage) yields the lowest weighted average is selected.

II.4. Stopping the Tree from Growing

As the tree grows bigger, the splitting rules become more detailed, and the tree gradually shifts from identifying “**big**” rules that capture real and reliable relationships in the data to “**tiny**” rules that reflect only noise. A fully grown tree results in **completely pure leaves** and, hence, **100% accuracy on the training data**. This accuracy is, of course, **illusory**—we have **overfit the data, fitting noise in the training data rather than the true signal**, which leads to **poor performance on new, unseen data**.

We need some way to determine when to stop growing a tree at a stage that will generalize to new data. There are various ways to stop splitting in Python:

- Avoid splitting a partition if a resulting subpartition is **too small**, or if a **terminal leaf contains too few samples**. In Python’s **DecisionTreeClassifier**, this behavior is controlled using the parameters **min_samples_split** (which specifies the **minimum number of samples required to split a node**) and **min_samples_leaf** (which specifies the **minimum number of samples required in a leaf node**). These parameters help **reduce overfitting** by preventing the tree from learning **overly specific rules based on very small sample sizes**.
- Don’t split a partition if the new split does not **significantly reduce impurity**. In Python’s **DecisionTreeClassifier**, this behavior is controlled by the parameter **min_impurity_decrease**, which **prevents a split unless it produces a sufficient weighted reduction in impurity**. Smaller values of **min_impurity_decrease** allow more splits, resulting in more complex trees, while larger values restrict tree growth, helping to **reduce overfitting**.

These methods involve **arbitrary rules** and can be useful for **exploratory work**, but we **can’t easily determine optimum values** (i.e., values that **maximize predictive accuracy on new data**). We need to **combine cross-validation** with either **systematically changing the model parameters** or **modifying the tree through pruning**.

Controlling tree complexity in Python

Neither the **complexity parameter** nor **pruning** is available in **scikit-learn’s DecisionTree** implementation. The solution is to use **grid search** over **combinations of different parameter values**.

For example, we can vary `max_depth` in the range 5 to 30 and `min_samples_split` between 20 and 100. The `GridSearchCV` method in scikit-learn is a **convenient way to combine exhaustive search with cross-validation**. An **optimal parameter set** is then selected based on **cross-validated model performance**.

II.5. Predicting a Continuous Value

Predicting a **continuous value (also termed regression)** with a tree follows the same logic and procedure, except that **impurity is measured by squared deviations from the mean (squared errors)** in each subpartition, and **predictive performance is judged by the square root of the mean squared error (RMSE)** (see “Assessing the Model” on page 153) in each partition.

scikit-learn has the `sklearn.tree.DecisionTreeRegressor` method to train a decision tree regression model.

How Trees Are Used ?

A major challenge for predictive modelers is the “**black box**” **perception** of many algorithms, which can lead to **resistance within organizations**. **Decision trees** offer two key advantages in this context. First, they provide a **visual tool** to explore the data, revealing which **variables are important** and how they **interact**, including **nonlinear relationships**. Second, trees generate a **clear set of rules** that can be **communicated to non-specialists**, useful both for **implementation** and for “**selling**” a data mining project.

For prediction, however, **using multiple trees usually outperforms a single tree**. Methods like **random forests** and **boosted trees** generally deliver **better accuracy and performance**, though the **interpretability and simplicity** of a single tree are lost.

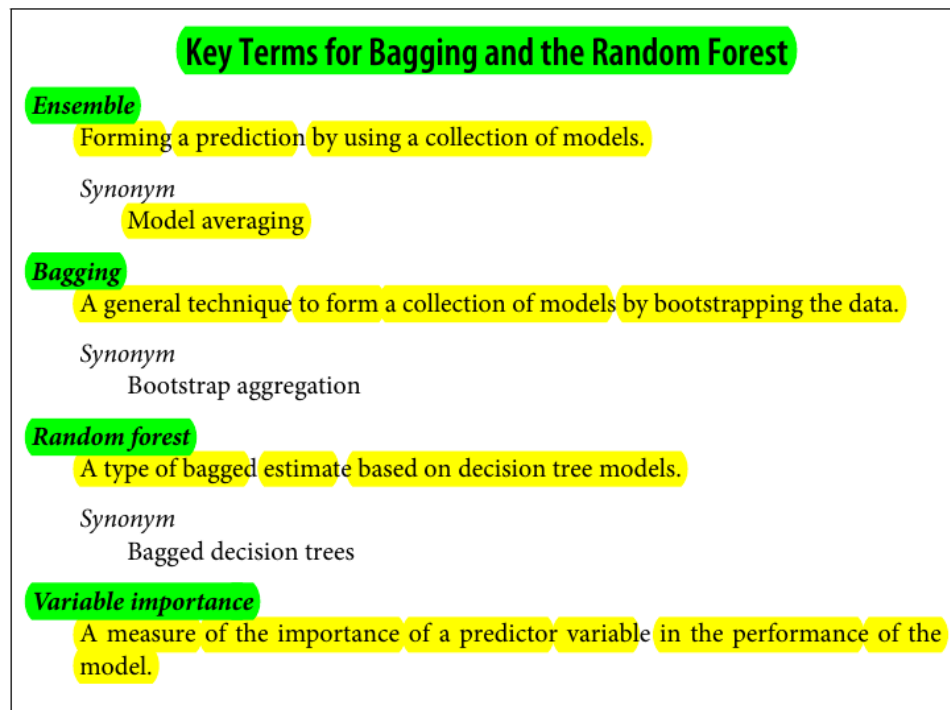
Key Ideas

- Decision trees produce a set of rules to classify or predict an outcome.
- The rules correspond to successive partitioning of the data into subpartitions.
- Each partition, or split, references a specific value of a predictor variable and divides the data into records where that predictor value is above or below that split value.
- At each stage, the tree algorithm chooses the split that minimizes the outcome impurity within each subpartition.
- When no further splits can be made, the tree is fully grown and each terminal node, or leaf, has records of a single class; new cases following that rule (split) path would be assigned that class.
- A fully grown tree overfits the data and must be pruned back so that it captures signal and not noise.
- Multiple-tree algorithms like random forests and boosted trees yield better predictive performance, but they lose the rule-based communicative power of single trees.

“but they lose the rule-based communicative power of single trees” means that while **multiple-tree algorithms** (like random forests or boosted trees) are **more accurate**, they **aren’t as easy to interpret or explain**.

III. Bagging and the Random Forest

This principle applies to predictive models as well: averaging (or taking majority votes) of multiple models—an ensemble of models—turns out to be more accurate than just selecting one model.



The **ensemble approach** has been applied to and across many different modeling methods, most publicly in the **Netflix Prize**, in which Netflix offered a **\$1 million prize** to any contestant who came up with a model that produced a **10% improvement** in predicting the rating that a Netflix customer would award a movie. The simple version of ensembles is as follows:

1. **Develop a predictive model** and **record the predictions** for a given data set.
2. **Repeat for multiple models** on the same data.
3. For each record to be predicted, take an **average, weighted average, or majority vote** of the predictions.

Ensemble Tree Models (e.g., Random Forest, Boosted Trees)

Structure: Many trees are trained on the data (using different samples or weighting), and their predictions are **combined**.

Ensemble methods work especially well with **decision trees**, making it possible to build **accurate predictive models with relatively little effort**. There are two main types of ensemble approaches: **bagging** and **boosting**. When applied to trees, these are called **random forests**

(bagging) and boosted trees (boosting). This section focuses on **bagging**, while **boosting** is discussed separately (see page 270).

III.1 Bagging

Suppose we have a response Y and P predictor variables $\mathbf{X} = X_1, X_2, \dots, X_P$ with N records.

Bagging (Bootstrap Aggregating) is a simple but powerful ensemble technique. Instead of training all models on the same data, **each model is trained on a bootstrap resample**—a random sample **with replacement** from the training set.

The algorithm works as follows:

1. **Decide** how many models (M) to train and the size of each resample ($n < \text{total records } N$).
2. **Draw a bootstrap sample** of n records from the training data to form a bag.
3. **Train a model** on this bag to generate a set of decision rules.
4. **Repeat** the process M times, creating M models, each trained on a slightly different subset of the data.

In the case where f_m predicts the probability $Y = 1$; the bagged estimate is given by:

$$\hat{f} = \frac{1}{M} (\hat{f}_1(\mathbf{X}) + \hat{f}_2(\mathbf{X}) + \dots + \hat{f}_M(\mathbf{X}))$$

III.2 Random Forest

A **random forest** builds on **bagging with decision trees**, with one important extension: it **samples both records and variables**. In a **traditional decision tree**, to create a subpartition of a partition A , the algorithm selects the **variable and split point** by minimizing a criterion such as **Gini impurity** (see “Measuring Homogeneity or Impurity” on page 254).

In a **random forest**, at each split, the choice of variable is limited to a **random subset of variables**, adding **diversity among trees** and reducing correlation. Compared to the basic tree algorithm (see “The Recursive Partitioning Algorithm” on page 252), the random forest adds:

1. **Bagging**—training each tree on a **bootstrap sample** (see “Bagging and the Random Forest” on page 259).
2. **Random variable sampling**—choosing splits from a **subset of variables** at each node.

Example: If a dataset has 10 features, a random forest might consider only 3 randomly selected features at each split, making the trees **less similar** and the ensemble **more robust and accurate**.

This is the **procedure for building a random forest tree:**

1. **Bootstrap sampling:** Take a **random sample with replacement** from the training records. This creates a “bag” of data for one tree.
2. **Variable sampling at a split:** For the first split, randomly select **p features out of P total features** (without replacement).
3. **Evaluate splits:** For each sampled feature X_j :
 - a. Consider possible **split values** s_j .
 - b. Split the current partition into two groups: **records with $X_j < s_j$** and **records with $X_j \geq s_j$**
 - c. Measure how **homogeneous the resulting subpartitions are** (e.g., using Gini impurity).
 - d. Choose the **split value s_j that maximizes homogeneity**.
4. **Choose the best split:** Among all sampled features, pick the **feature and split value** that gives the **highest homogeneity**.
5. **Repeat splits:** Move to the next node and **repeat steps 2–4** until the tree is fully grown (or stopping rules are met).
6. **Build more trees:** Go back to step 1, take a **new bootstrap sample**, and repeat the process to build **additional trees**.

Example:

- Dataset: 1000 records, 10 features.
- For each split, randomly pick **3 features**, test their possible split points, choose the best, then move to the next node.
- Repeat for 500 trees in the forest.
- Each tree sees a **different subset of records and variables**, creating a **diverse ensemble** that improves accuracy.

Types of nodes in a simple Tree

1. **Root node:**
 - This is the **very first node** at the top of the tree.
 - It contains **all the training data**.
 - The first split is applied here.
2. **Internal (or decision) nodes:**
 - These are the **nodes created after the root node** when a split happens.
 - Each node represents a **subset of the data** defined by the split above it.
 - At each internal node, the algorithm can make **another split** based on one of the features.

3. Leaf (terminal) nodes:

- These are the **end points of the tree**, where no further splitting occurs.
- They **hold the prediction** (e.g., a class label for classification or a value for regression).

How many variables to sample at each step?

When building a **random forest**, at each split, the algorithm **doesn't consider all predictor variables**—it only considers a **random subset of them**.

- A **rule of thumb** is:
 - For **classification**, use \sqrt{P} **features**, where **P** is the total number of predictor variables.
 - For **regression**, a common choice is **P/3 features**.
- In **Python**, the `sklearn.ensemble.RandomForestClassifier` controls this with the parameter:
 - **max_features**: the number of features to randomly select at each split.
 - Default for classification: \sqrt{P}
 - Default for regression: **all features**

In Python, we use the method `sklearn.ensemble.RandomForestClassifier`.

By default, the random forest is trained using **500 trees**, which helps stabilize predictions by averaging over many models. In this case, the predictor set contains **only two variables**, so at each split the algorithm **randomly selects just one variable** to consider. In other words, each split is based on a **bootstrap subsample of size 1 from the predictors**.

For **classification**, the **default rule** is:

$\text{max_features} = \sqrt{P}$, where **P** is the number of predictor variables.

- Number of predictors: **P = 2**
- **sqrt (p) = 1.41**
- scikit-learn **rounds this down to 1**

➡ **So at each split, only one variable is randomly selected and tested.**

The **out-of-bag (OOB) error** is a built-in way to estimate how well a **random forest** performs, without needing a separate validation set.

Because each tree is trained on a **bootstrap sample** of the data, about **one-third of the observations are left out** of that tree's training set. These left-out observations are called **out-of-bag samples**. For each record, the model predicts its outcome using **only the trees where that record was not used for training**, and the resulting error is the **OOB error**.

As more trees are added to the forest, the OOB error typically **decreases and then stabilizes**, showing how performance improves with the number of trees. This is why the **OOB error can be plotted against**

the number of trees—to verify that the forest has enough trees and that adding more no longer yields meaningful gains.

```
# Optional: check out-of-bag score
print("OOB score:", rf.oob_score_)
```

OOB score = 0.5753 means:

- The random forest correctly predicts the outcome for about **57.5% of the observations**
- Using **only out-of-bag data** (data not seen by the trees that made the prediction)
- This is an estimate of **how the model would perform on new, unseen data**

The result is shown in Figure 6-6. The error rate rapidly decreases from over 0.44 before stabilizing around 0.385.

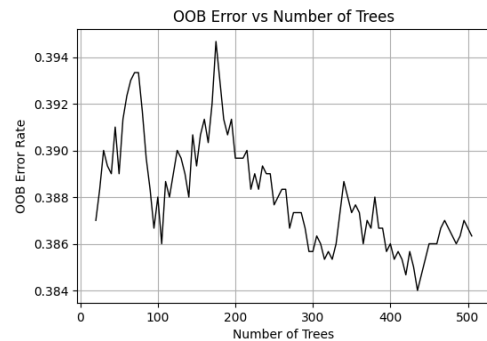
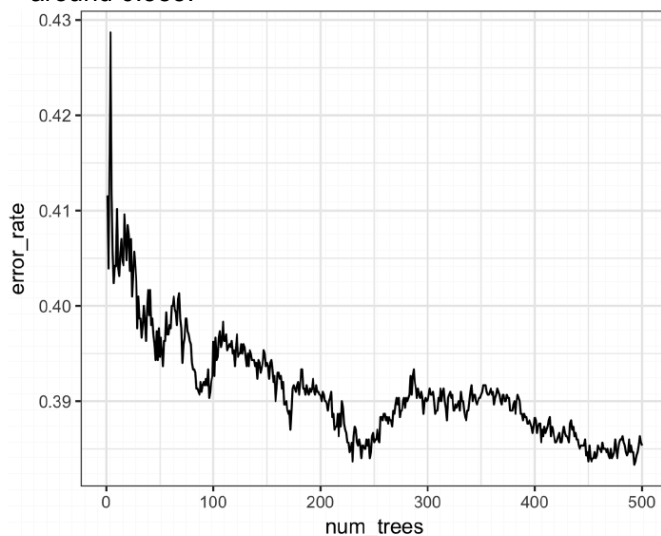
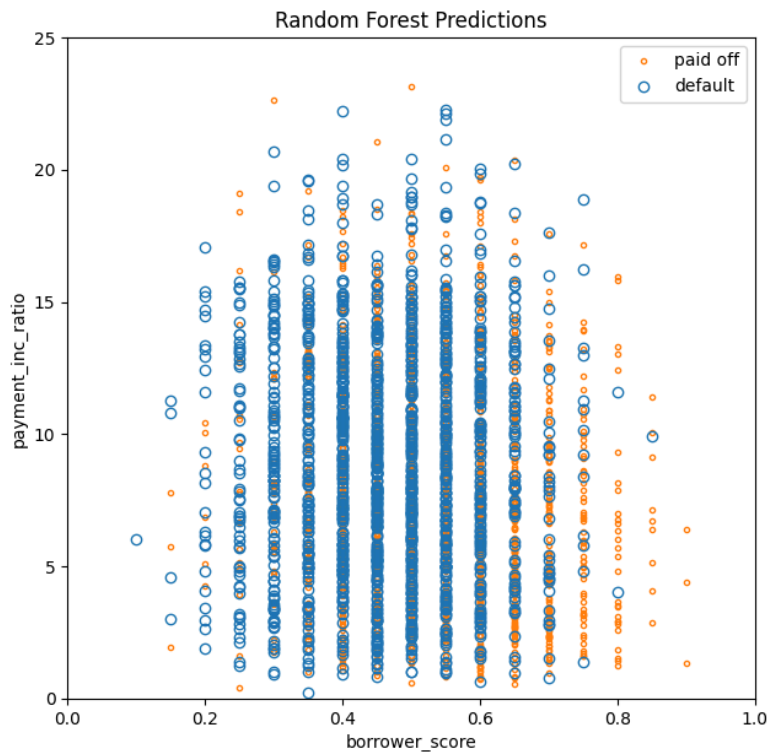


Figure 6-6. An example of the improvement in accuracy of the random forest with the addition of more trees



The plot, shown in Figure 6-7, is quite revealing about the nature of the **random forest**.

The **random forest method** is a “**black box**” method. It produces **more accurate predictions** than a simple tree, but the simple tree’s **intuitive decision rules are lost**. The **random forest predictions are also somewhat noisy**: note that **some borrowers with a very high score, indicating high creditworthiness, still end up with a prediction of default**. This is a result of **some unusual records in the data** and demonstrates the **danger of overfitting by the random forest** (see “**Bias-Variance Trade-off**” on page 247).

III.3. Variable Importance

The **power of the random forest algorithm** shows itself when you **build predictive models for data with many features and records**. It has the ability to **automatically determine which predictors are important** and **discover complex relationships between predictors corresponding to interaction terms** (see “**Interactions and Main Effects**” on page 174).

In Python, the `RandomForestClassifier` collects information about feature importance during training and makes it available with the field `feature_importances_`:

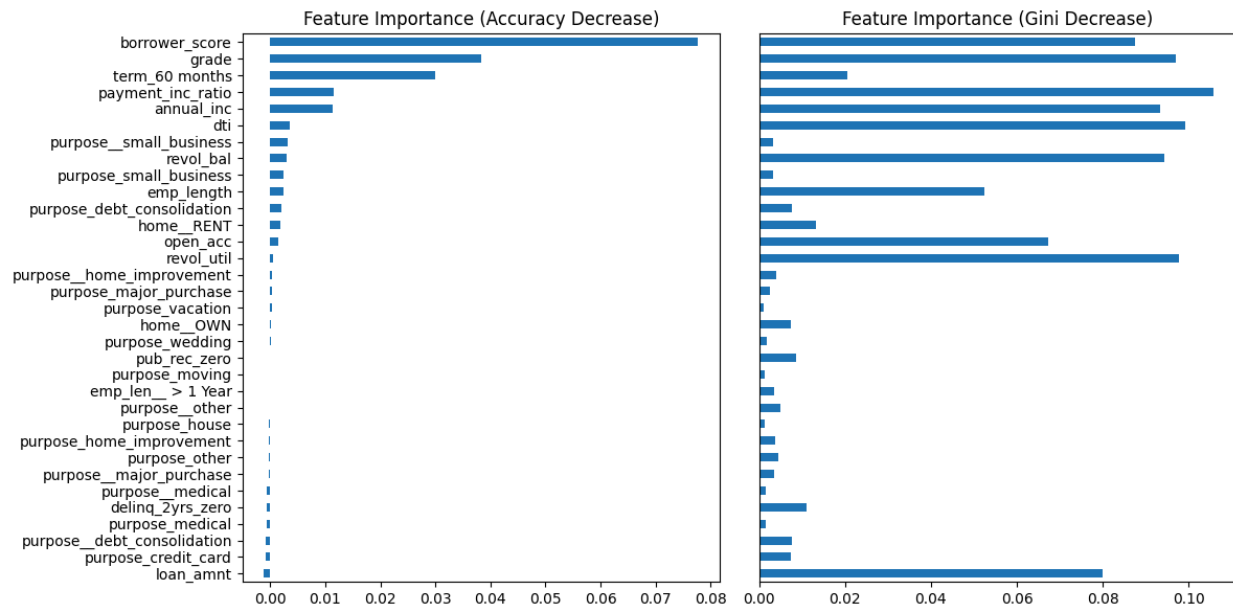


Figure 6-8. The importance of variables for the full model fit to the loan data

When we use a random forest, we often want to know **which variables are really driving the predictions**. There are **two main ways to measure this “variable importance.”**

1 Accuracy decrease (type 1)

Imagine you have a model predicting whether a borrower will default. If you **randomly shuffle the values of one variable**, say `annual_inc`, you destroy its predictive power. Then you check how much the model’s accuracy drops.

- **Example:** Before shuffling, the model predicts 90% correctly. After shuffling `annual_inc`, accuracy drops to 80%. That 10% drop tells you that `annual_inc` is quite important.
- This method uses **out-of-bag data**, which is like a built-in cross-validation, so it gives a reliable estimate.
- **Pros:** Reflects the real contribution of a variable to model performance.
- **Cons:** Slightly slower because it requires recomputing predictions after shuffling.

2 Gini decrease (type 2)

Random forests also track how much **splitting on a variable improves node purity** in each tree. This is the mean decrease in **Gini impurity**.

- **Example:** If splitting on `borrower_score` often creates very “pure” nodes (mostly defaults or non-defaults), the Gini decrease is high.
- This measure is based on the **training set**, so it can sometimes overestimate importance, especially if the model overfits.
- **Pros:** Quick to compute, gives a sense of which variables help separate the classes.
- **Cons:** Less reliable than type 1 because it doesn’t use out-of-bag evaluation.

In short:

- Use **accuracy decreases** to see the real impact of a variable on predictions.
- Use **Gini decrease** for a fast, rough estimate of importance.

why we still use Gini decrease

1. **Computational efficiency** — Gini is calculated automatically while building the forest, while accuracy decrease requires extra computation (shuffling + predicting).
2. **Insight into tree structure** — Gini shows which variables the forest uses for its splitting rules, information that is otherwise hidden in a large ensemble.

In production scenarios, where thousands of models may need to run, making the faster Gini measure attractive.

III.4. Hyperparameters

The **random forest** is often called a **black-box algorithm**: it gives powerful predictions, but it's not immediately obvious how it makes them. Luckily, we can adjust “**knobs**” to control how it works — these are called **hyperparameters**. Unlike regular model parameters, **hyperparameters must be set before training** and are **not learned from the data**.

While in traditional models you might choose which predictors to include, in a random forest, **hyperparameters are more critical**, especially to prevent **overfitting** (making trees that fit the training data too closely).

1 nodesize / min_samples_leaf

- This controls the **minimum number of samples in a terminal node** (the leaves of the tree).
- If the number is too small, the tree can overfit; if too large, the tree may underfit.
- **Defaults:** Python (scikit-learn): 1 for both (classification or regression)
- **Example:** If you set `min_samples_leaf=5`, a leaf must have at least 5 borrowers before the tree can make a decision there.

2 maxnodes / max_leaf_nodes

- This sets the **maximum number of nodes or leaves in a tree**.
- By default, there's no limit, and trees grow until they hit the `nodesize` limit.
- The parameters are related: $\text{maxnodes} = 2 \times \text{max_leaf_nodes} - 1$
- **Example:** If `max_leaf_nodes=10`, the tree can have up to $2 \times 10 - 1 = 19$ nodes.

It may be tempting to **ignore these parameters** and simply go with the **default values**. However, using the defaults may lead to **overfitting** when you apply the random forest to **noisy data**. When you **increase nodesize/min_samples_leaf** or set **maxnodes/max_leaf_nodes**, the algorithm will fit **smaller trees** and is **less likely to create spurious predictive rules**. **Cross-validation** (see “Cross-Validation” on page 155) can be used to **test the effects of setting different values for hyperparameters**.

Key Ideas

- Ensemble models improve model accuracy by combining the results from many models.
- Bagging is a particular type of ensemble model based on fitting many models to bootstrapped samples of the data and averaging the models.
- Random forest is a special type of bagging applied to decision trees. In addition to resampling the data, the random forest algorithm samples the predictor variables when splitting the trees.
- A useful output from the random forest is a measure of variable importance that ranks the predictors in terms of their contribution to model accuracy.
- The random forest has a set of hyperparameters that should be tuned using cross-validation to avoid overfitting.

IV. Boosting

Ensemble models have become a **standard tool** for predictive modeling. **Boosting** is a **general technique** to create an **ensemble of models**. It was developed around the same time as **bagging** (see “Bagging and the Random Forest” on page 259). Like bagging, boosting is most commonly used with **decision trees**.

Despite their similarities, **boosting** takes a very different approach—one that comes with many more **bells and whistles**. As a result, while **bagging** can be done with relatively little tuning, **boosting** requires much **greater care** in its application.

If these two methods were cars, **bagging** could be considered a **Honda Accord** (reliable and steady), whereas **boosting** could be considered a **Porsche** (powerful but requires more care)

In **linear regression models**, the **residuals** are often examined to see if the **fit can be improved** (see “Partial Residual Plots and Nonlinearity” on page 185). **Boosting** takes this concept much further and fits a **series of models**, in which each **successive model seeks to minimize the error of the previous model**. Several **variants of the algorithm** are commonly used: **Adaboost**, **gradient boosting**, and **stochastic gradient boosting**. The latter, **stochastic gradient boosting**, is the **most general and widely used**. Indeed, with the **right choice of parameters**, the algorithm can **emulate the random forest**.

Key Terms for Boosting

Ensemble

Forming a prediction by using a collection of models.

Synonym

Model averaging

Boosting

A general technique to fit a sequence of models by giving more weight to the records with large residuals for each successive round.

Adaboost

An early version of boosting that reweights the data based on the residuals.

Gradient boosting

A more general form of boosting that is cast in terms of minimizing a cost function.

Stochastic gradient boosting

The most general algorithm for boosting that incorporates resampling of records and columns in each round.

Regularization

A technique to avoid overfitting by adding a penalty term to the cost function on the number of parameters in the model.

Hyperparameters

Parameters that need to be set before fitting the algorithm.

IV.1. Boosting Algorithm

There are various boosting algorithms, and the basic idea behind all of them is essentially the same. The easiest to understand is **Adaboost**, which proceeds as follows:

Think of **boosting** as training a team of models **one after another**, where each new model focuses more on the mistakes made so far.

Step 1: Start simple

You first decide **how many models** you're willing to train (M). Every observation starts with the **same weight** $w_i=1/N$, meaning all data points are equally important. The ensemble model starts at zero — it knows nothing yet.

Step 2: Train the first model

Using the current weights, you train a model that tries to **minimize weighted error**. Misclassifying an observation with a high weight is penalized more than misclassifying one with a low weight.

Example:

If a borrower who defaults is misclassified and has a high weight, the model is “punished” more than if it misclassifies an easy, low-weight case.

Step 3: Add the model to the ensemble

The model is added to the ensemble with a weight α_m , which depends on how well it performed.

- **Lower error** → **larger α_m**
- Better models get **more influence** in the final prediction.

Step 4: Reweight the data

Now comes the key idea: **increase the weights of misclassified observations**. This forces the next model to **pay more attention to the hard cases**. The worse the mistake, the bigger the weight increase.

Example:

Borrowers that were repeatedly misclassified become “louder” in the dataset, pushing future models to learn patterns that explain them.

Step 5: Repeat

Increase the counter and repeat the process until you reach M . Each new model corrects the errors of the previous ones.

The boosted estimate is given by:
$$\hat{F} = \alpha_1 \hat{f}_1 + \alpha_2 \hat{f}_2 + \dots + \alpha_M \hat{f}_M$$

By **increasing the weights for the observations that were misclassified**, the algorithm **forces the models to train more heavily on the data for which it performed poorly**. The factor α_m ensures that **models with lower error have a bigger weight**.

#=====

Gradient boosting is closely related to **AdaBoost**, but it reframes the problem as **optimizing a cost (loss) function**. Instead of changing observation weights, it looks at **how wrong the model currently is** and fits the next model to the **pseudo-residuals** — essentially the **direction and size of the error** left to fix.

Example:

If a model underestimates a house price by \$50k, that large residual becomes a strong signal. The next tree is trained to specifically correct that \$50k mistake, while smaller errors get less attention.

This naturally means that **larger residuals receive more focus**, just like in AdaBoost, but through **optimization rather than reweighting**.

Stochastic gradient boosting adds another layer of robustness by introducing **randomness**, much like a random forest. At each stage, it:

- Trains on a **random subset of observations**, and

- Considers a **random subset of predictors**.

Example:

One tree might be trained on 70% of borrowers and a subset of financial variables, while the next tree sees a different subset. This reduces overfitting and improves generalization.

In short:

- Gradient boosting = **learn from residuals via loss optimization**
- Stochastic gradient boosting = **gradient boosting + randomness**, borrowing stability from random forests

#=====

IV.2 XGBoost (XGBoost is an implementation of *stochastic gradient boosting*)

The **most widely used public-domain software for boosting** is **XGBoost**, an implementation of **stochastic gradient boosting** originally developed by **Tianqi Chen and Carlos Guestrin** at the **University of Washington**. A **computationally efficient implementation with many options**, it is available as a **package for most major data science software languages**.

The method **xgboost** has **many parameters that can, and should, be adjusted** (see “Hyperparameters and Cross-Validation” on page 279). Two **very important parameters** are **subsample**, which **controls the fraction of observations sampled at each iteration**, and **eta**, a **shrinkage factor applied to α_m in the boosting algorithm** (see “The Boosting Algorithm” on page 271).

Using **subsample** makes **boosting act like the random forest**, except that the **sampling is done without replacement**. The **shrinkage parameter eta** is **helpful to prevent overfitting by reducing the change in the weights** (a **smaller change in the weights** means the algorithm is **less likely to overfit to the training set**).

In **Python**, **xgboost** has **two different interfaces**: a **scikit-learn API** and a **more functional interface like in R**. To be **consistent with other scikit-learn methods**, **some parameters were renamed**. For example, **eta** is **renamed to learning_rate**; using **eta** will not fail, but it will not have the desired effect.

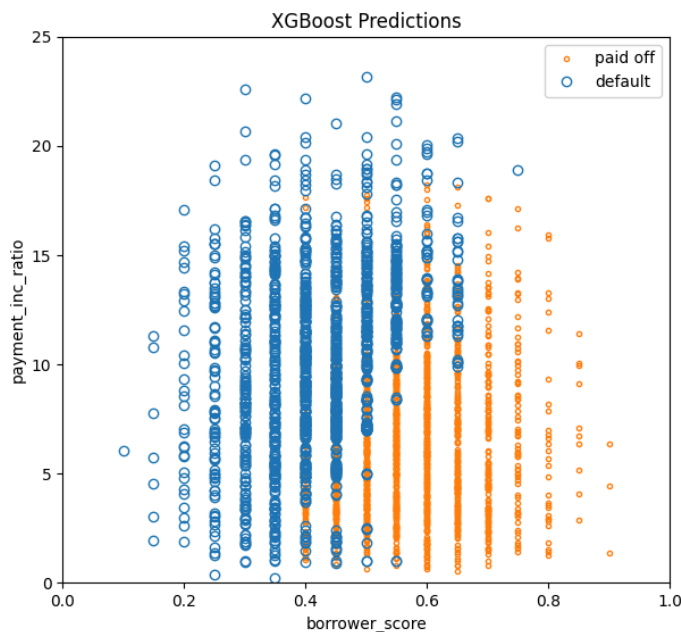


Figure 6-9. The predicted outcomes from XGBoost applied to the loan default data

The result is shown in Figure 6-9. Qualitatively, this is similar to the predictions from the random forest; see Figure 6-7. The predictions are somewhat noisy in that some borrowers with a very high borrower score still end up with a prediction of default.

IV.3. Regularization: Avoiding Overfitting

Blind application of xgboost can lead to unstable models as a result of overfitting to the training data. The **problem with overfitting** is twofold:

- The accuracy of the model on new data not in the training set will be degraded.

Example: Suppose you train XGBoost on borrowers from 2022. It achieves 98% accuracy on that data. But when applied to 2023 borrowers, accuracy falls to 75% because it overfit to specific patterns in 2022 that don't hold anymore.

- The predictions from the model are highly variable, leading to unstable results.

Example: Train two XGBoost models on slightly different sets of 3,000 borrowers. One predicts a certain high-score borrower will default; the other predicts they will pay off. This variability makes the model less trustworthy.

Any **modeling technique** is potentially prone to **overfitting**. For example, if **too many variables are included** in a **regression equation**, the model may end up with **spurious predictions**. However, for most **statistical techniques**, **overfitting can be avoided by a judicious selection of predictor**

variables. Even the **random forest** generally produces a **reasonable model without tuning the parameters**.

This, however, is not the case for xgboost. Fit xgboost to the loan data for a training set with all of the variables included in the model.

We use the function **train_test_split** in Python to split the data set into training and test sets.

```
Regularization avoids overfitting  
Default error rate: 0.3622  
(array) <class 'xgboost.core.XGBModel'>
```

1) What the error rate is

You calculated:

```
error_default = abs(valid_y - (pred_default > 0.5).astype(int))  
np.mean(error_default)
```

- `pred_default > 0.5` → converts predicted probabilities into class predictions:
 - 1 if the model predicts default probability > 0.5
 - 0 otherwise
- `abs(valid_y - prediction)` → 1 if prediction is wrong, 0 if correct
- `np.mean(error_default)` → fraction of misclassified borrowers

So **0.3622** means:

About 36.2% of borrowers in the validation set were misclassified by this XGBoost model.

Example

- Suppose we have 10,000 validation borrowers:
 - $36.2\% \times 10,000 \approx \mathbf{3,620}$ borrowers were misclassified
 - 6,380 borrowers were correctly classified
- For instance:
 - High-score borrower predicted to “paid off” but actually defaulted → counts as 1 error
 - Low-score borrower predicted to “default” but actually paid off → counts as 1 error

In this scenario, your dataset is split into:

- **Training set** → all records except 10,000 randomly sampled ones

- **Test set** → the 10,000 sampled records

When we train the **boosting model**:

- The **training error is very low (13.3%)**, meaning the model fits the training data almost perfectly.
- The **test error is much higher (35.3%)**, meaning the model struggles on new, unseen data.

This difference illustrates **overfitting**:

- **Boosting is very powerful** and can capture nearly all patterns in the training data, including **noise or random quirks**.
- However, the patterns it learned **do not generalize well** to the test set.

Example:

- Suppose in the training set, borrowers with a very specific combination of **borrower_score**, **dti**, and **payment_inc_ratio** all happened to default. Boosting may create a rule that says “all borrowers with that combination will default.”
- In the test set, a borrower has the same combination but actually paid off. The model **wrongly predicts default**, contributing to the high test error.

Boosting provides several **parameters to avoid overfitting**, including the parameters **eta** (or **learning_rate**) and **subsample** (see “XGBoost” on page 272). Another approach is **regularization**, a technique that **modifies the cost function** in order to **penalize the complexity of the model**. **Decision trees** are fit by minimizing **cost criteria** such as **Gini’s impurity score** (see “Measuring Homogeneity or Impurity” on page 254). In **XGBoost**, it is possible to **modify the cost function** by **adding a term that measures the complexity of the model**.

There are **two parameters in XGBoost to regularize the model**: **alpha** and **lambda**, which correspond to **Manhattan distance (L1-regularization)** and **squared Euclidean distance (L2-regularization)**, respectively (see “Distance Metrics” on page 241). **Increasing these parameters will penalize more complex models and reduce the size of the trees that are fit**. For example, see what happens if we set **lambda** to **1,000**. In the **scikit-learn API**, the parameters are called **reg_alpha** and **reg_lambda**.

```
Penalty error rate: 0.3351
```

Now the training error is only slightly lower than the error on the test set.

In Python, we can call the `predict_proba` method with the `ntree_limit` argument, that forces only the first `i` trees to be used in the prediction.

	iterations	default train	penalty train	default test	penalty test
0	1	0.339228	0.338606	0.3531	0.3513
1	2	0.330344	0.336653	0.3474	0.3467
2	3	0.324967	0.337842	0.3434	0.3477
3	4	0.321544	0.334333	0.3435	0.3442
4	5	0.319903	0.335465	0.3405	0.3453

We can use the `pandas plot` method to create the line graph.



Figure 6-10. The error rate of the default XGBoost versus a penalized version of XGBoost

The result, displayed in Figure 6-10, shows how the default model steadily improves the accuracy for the training set but actually gets worse for the test set. The penalized model does not exhibit this behavior.

Ridge regression and the lasso

Back in the 1970s, statisticians discovered that adding a **penalty on model complexity** helps prevent overfitting. In **ordinary least squares (OLS) regression**, we minimize the **residual sum of squares (RSS)**:

$$RSS = \sum_{i=1}^n (Y_i - (b_0 + b_1X_{i1} + \dots + b_pX_{ip}))^2$$

But when there are **many predictors or noisy data**, OLS can produce very large coefficients that **overfit** the training data.

① Ridge Regression (L2 regularization)

- Adds a penalty on the **squared size of coefficients**:

$$RSS + \lambda \sum_{j=1}^p b_j^2$$

- λ controls how much the coefficients are penalized:
 - Small $\lambda \rightarrow$ model is close to OLS
 - Large $\lambda \rightarrow$ coefficients shrink toward 0, producing a **simpler model**
- **Example:** In a loan dataset, if **annual_income** is highly variable, Ridge prevents its coefficient from dominating predictions.

② Lasso Regression (L1 regularization)

- Adds a penalty on the **absolute value of coefficients**:

$$RSS + \alpha \sum_{j=1}^p |b_j|$$

- Effects:
 - Shrinks coefficients
 - Can set some coefficients exactly to 0 \rightarrow **automatic feature selection**
- **Example:** If **pub_rec_zero** is barely predictive, Lasso may remove it entirely from the model.

③ Connection to XGBoost

- XGBoost's **reg_lambda** acts like **Ridge (L2)**, penalizing large coefficients to reduce overfitting.
- XGBoost's **reg_alpha** acts like **Lasso (L1)**, shrinking coefficients and potentially zeroing out weak predictors.
- This allows XGBoost to **control tree complexity** in a way similar to Ridge and Lasso in regression.

IV.4. Hyperparameters and Cross-Validation

XGBoost is extremely powerful, but it comes with a **daunting array of hyperparameters**—things like `max_depth`, `learning_rate`, `subsample`, `reg_alpha`, `reg_lambda`, and many more. The problem is that the **specific choice of these hyperparameters can dramatically affect how well the model fits the data**. So how do we choose the “best” combination? This is where **cross-validation (CV)** comes in.

1) What is cross-validation?

- Cross-validation **splits the data into K folds** (groups).
- For each fold:
 1. Train the model on the **other K-1 folds**.
 2. Test the model on the **current fold** (the out-of-sample data).
- Repeat this process for all folds.
- Finally, **average the errors across all folds** to get a reliable measure of how the model performs on unseen data.

CV gives a **robust estimate of out-of-sample performance**. It prevents choosing hyperparameters that work well on one part of the data but **fail elsewhere**.

3) Example

Suppose we want to tune `max_depth` and `learning_rate` for a loan default prediction:

1. Try `max_depth = 3, 4, 5` and `learning_rate = 0.1, 0.05`.
2. For each combination, perform **5-fold CV**:
 - Fold 1 → train on folds 2–5, test on fold 1
 - Fold 2 → train on folds 1, 3–5, test on fold 2
 - ... repeat for all 5 folds
3. Compute the **average test error** for each combination.
4. Pick the combination with the **lowest average error** → this is your best set of hyperparameters.

To illustrate the technique, we apply it to parameter selection for **XGBoost**. We focus on two key parameters: **eta** (the learning rate, which controls how much the model adapts at each step) and **max_depth** (the maximum depth of a tree from root to leaf). By default, `max_depth` is 6. Limiting tree depth helps control **overfitting**, because deeper trees can capture more complex patterns but may fit the noise in the data.

We begin by setting up the **cross-validation folds** and defining a **list of parameter values** to explore. For example, we might test `eta` values of 0.01, 0.1, and 0.3, and `max_depth` values of 3, 6, and 9. Then, using cross-validation, we evaluate combinations of these parameters to identify the ones that give the best performance while avoiding overfitting.

In python code: we apply the preceding algorithm to compute the error for each model and each fold using five folds. We use the function `itertools.product` from the Python standard library to create all possible combinations of the two hyperparameters.

```
bst.update(dtrain, iteration=i, fobj=obj)
{'eta': 0.9, 'max_depth': 9, 'avg_error': np.float64(0.38175068104025867)}
eta  max_depth  avg_error
0  0.1          3   0.329204
1  0.1          6   0.337136
2  0.1          9   0.346048
3  0.5          3   0.337573
4  0.5          6   0.370704
5  0.5          9   0.373682
6  0.9          3   0.353108
7  0.9          6   0.386724
8  0.9          9   0.381751
```

You have this: we print last evaluation is not the best. (✅ {eta: 0.1, max_depth: 3} IS the best model)

```
bst.update(dtrain, iteration=i, fobj=obj)
```

```
{'eta': 0.9, 'max_depth': 9, 'avg_error': np.float64(0.38175068104025867)}
```

❶ `bst.update(dtrain, iteration=i, fobj=obj)`

This is an **internal XGBoost call**.

- `bst` = your booster (model)
- `dtrain` = training data
- `iteration=i` = XGBoost is updating the model **at iteration i** (one tree added at a time)
- `fobj=obj` = custom objective function (here, usually default logistic)

What this means: XGBoost is building/updating the model. Seeing this line in the console just shows that Python is inside XGBoost's training loop. Nothing to worry about.

❷ **The dictionary:**

```
{'eta': 0.9, 'max_depth': 9, 'avg_error': 0.38175}
```

This comes from **your cross-validation loop**:

- `eta: 0.9` → the **learning rate**.
 - High `eta` = faster learning, bigger steps per tree.
 - Low `eta` = slower, more stable learning.

- `max_depth`: 9 → **maximum depth of each tree**.
 - High depth = more complex tree, can overfit.
 - Low depth = simpler tree, may underfit.
- `avg_error`: 0.38175 → the **average 0-1 classification error** across all 5 folds of CV.
 - Formula: `mean(abs(y_true - y_pred > 0.5))`
 - Here it's **≈ 38.2% misclassification**, meaning the model is **correct ~61.8% of the time** on unseen fold data.

✓ How to read it in plain English

“When training XGBoost with a learning rate of 0.9 and maximum tree depth 9, the model misclassifies about 38% of the samples on average during 5-fold cross-validation.”

Final verdict: Cross-validation suggests that using shallower trees with a smaller value of `eta/learning_rate` yields more accurate results. Since these models are also more stable, the best parameters to use are `eta=0.1` and `max_depth=3` (or possibly `max_depth=6`).

XGBoost Hyperparameters

The **hyperparameters** for `xgboost` are primarily used to balance overfitting with the accuracy and computational complexity. For a complete discussion of the parameters, refer to the `xgboost` documentation.

`eta/learning_rate`

The shrinkage factor between 0 and 1 applied to α in the boosting algorithm. The default is 0.3, but for noisy data, smaller values are recommended (e.g., 0.1). In *Python*, the default value is 0.1.

`nrounds/n_estimators`

The number of boosting rounds. If `eta` is set to a small value, it is important to increase the number of rounds since the algorithm learns more slowly. As long as some parameters are included to prevent overfitting, having more rounds doesn't hurt.

`max_depth`

The maximum depth of the tree (the default is 6). In contrast to the random forest, which fits very deep trees, boosting usually fits shallow trees. This has the advantage of avoiding spurious complex interactions in the model that can arise from noisy data. In *Python*, the default is 3.

`subsample` and `colsample_bytree`

Fraction of the records to sample without replacement and the fraction of predictors to sample for use in fitting the trees. These parameters, which are similar to those in random forests, help avoid overfitting. The default is 1.0.

`lambda/reg_lambda` and `alpha/reg_alpha`

The regularization parameters to help control overfitting (see “Regularization: Avoiding Overfitting” on page 274). Default values for *Python* are `reg_lambda=1` and `reg_alpha=0`. In *R*, both values have default of 0.

Key Ideas

- Boosting is a class of ensemble models based on fitting a sequence of models, with more weight given to records with large errors in successive rounds.
- Stochastic gradient boosting is the most general type of boosting and offers the best performance. The most common form of stochastic gradient boosting uses tree models.
- XGBoost is a popular and computationally efficient software package for stochastic gradient boosting; it is available in all common languages used in data science.
- Boosting is prone to overfitting the data, and the hyperparameters need to be tuned to avoid this.
- Regularization is one way to avoid overfitting by including a penalty term on the number of parameters (e.g., tree size) in a model.
- Cross-validation is especially important for boosting due to the large number of hyperparameters that need to be set.

Summary

This chapter has described two classification and prediction methods that “learn” flexibly and locally from data, rather than starting with a structural model (e.g., a linear regression) that is fit to the entire data set. *K*-Nearest Neighbors is a simple process that looks around at similar records and assigns their majority class (or average value) to the record being predicted. Trying various cutoff (split) values of predictor variables, tree models iteratively divide the data into sections and subsections that are increasingly homogeneous with respect to class. The most effective split values form a path, and also a “rule,” to a classification or prediction. Tree models are a very powerful and popular predictive tool, often outperforming other methods. They have given rise to various ensemble methods (random forests, boosting, bagging) that sharpen the predictive power of trees.