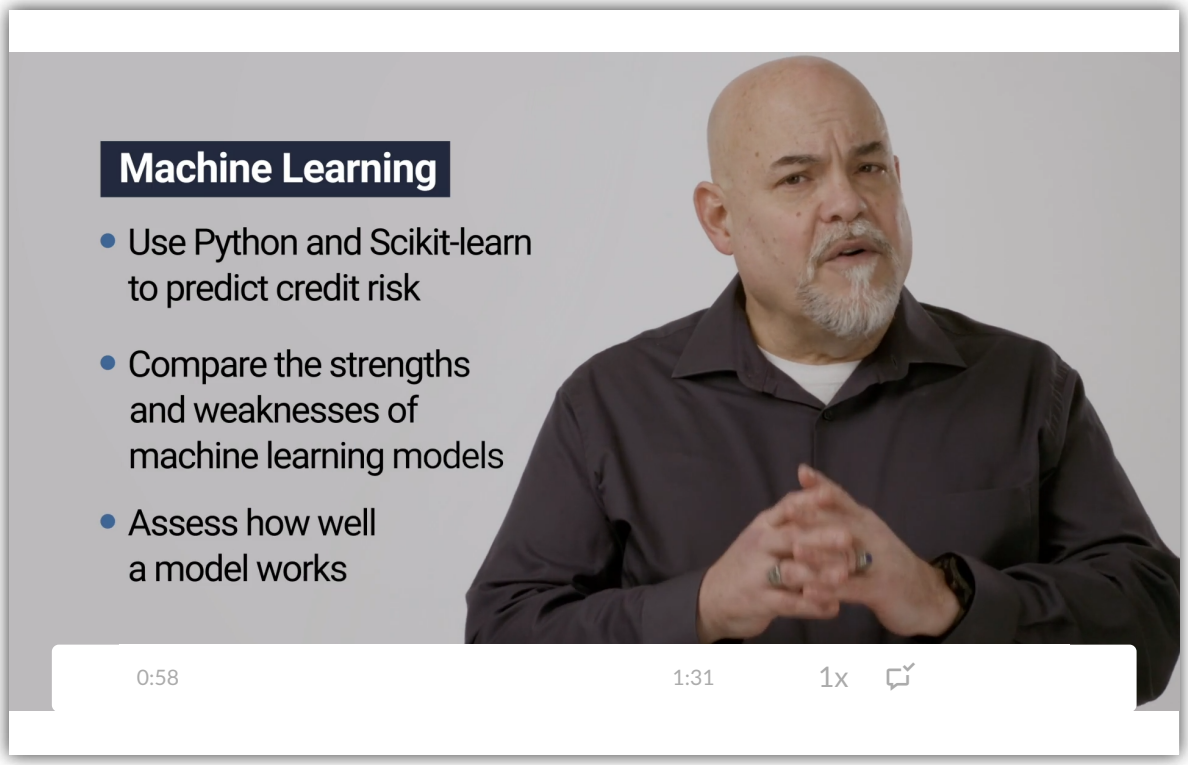
17.0.1 Predicting Credit Risk

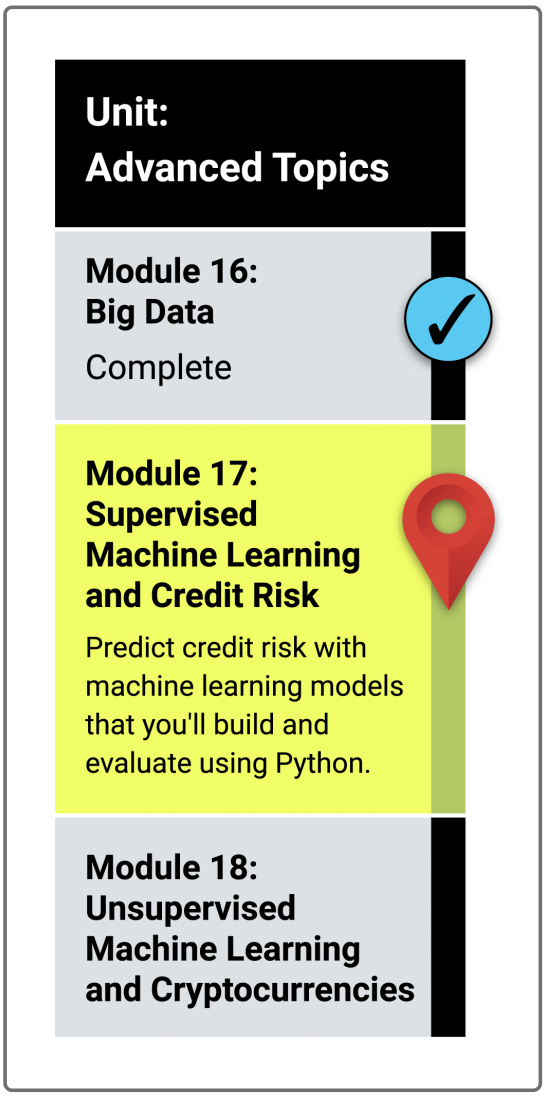


17.0.2 Module 17 Roadmap

Looking Ahead

In 2019, more than 19 million Americans had at least one unsecured personal loan. That's a record-breaking number! Personal lending is growing faster than credit card, auto, mortgage, and even student debt. With such incredible growth, FinTech firms are storming ahead of traditional loan processes. By using the latest machine learning techniques, these FinTech firms can continuously analyze large amounts of data and predict trends to optimize lending.

In this module, you'll use Python to build and evaluate several machine learning models to predict credit risk. Being able to predict credit risk with



machine learning algorithms can help banks and financial institutions predict anomalies, reduce risk cases, monitor portfolios, and provide recommendations on what to do in cases of fraud.

What You Will Learn

By the end of this module, you will be able to:

- Explain how a machine learning algorithm is used in data analytics.
- Create training and test groups from a given data set.
- Implement the logistic regression, decision tree, random forest, and support vector machine algorithms.
- Interpret the results of the logistic regression, decision tree, random forest, and support vector machine algorithms.
- Compare the advantages and disadvantages of each supervised learning algorithm.
- Determine which supervised learning algorithm is best used for a given data set or scenario.
- Use ensemble and resampling techniques to improve model performance.

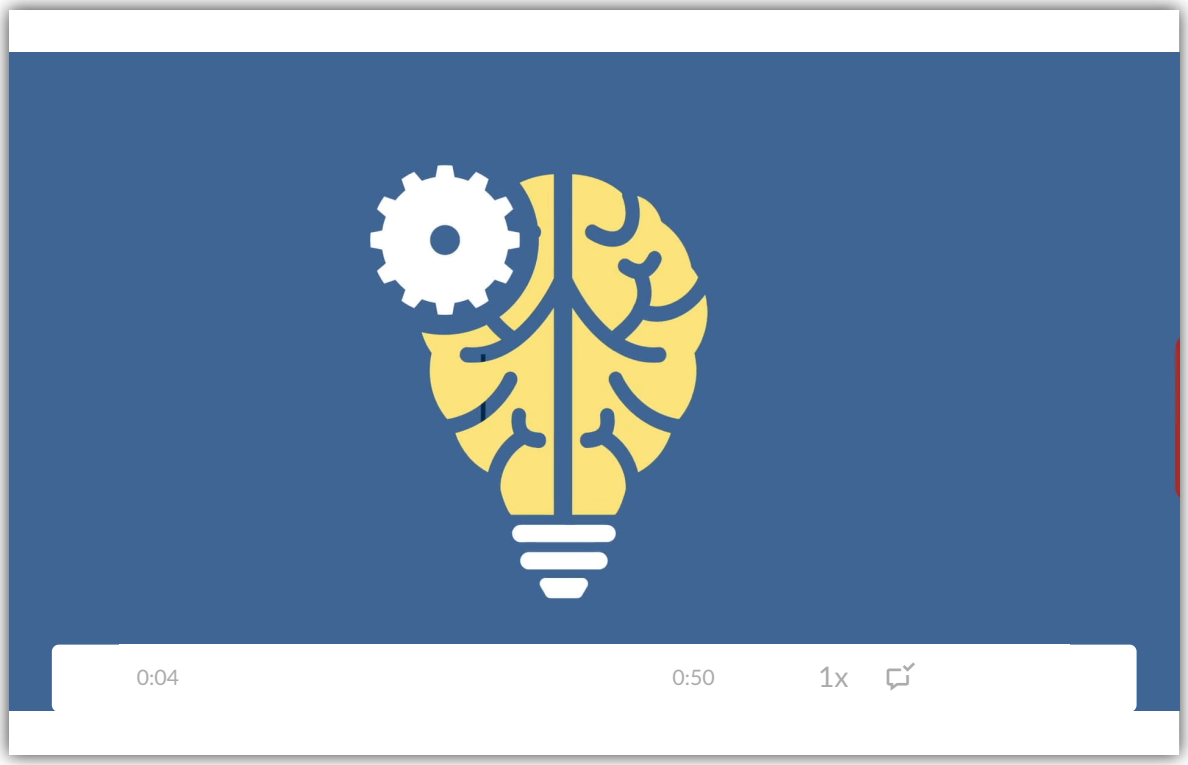
Planning Your Schedule

Here's a quick look at the lessons and assignments you'll cover in this module. You can use the time estimates to help pace your learning and plan your schedule:

- Introduction to Module 17 (15 minutes)
- Machine Learning Environment (30 minutes)
- Supervised Learning (1 hour)
- Logistic Regression (1 hour)

- Classification Model Validation (1 hour)
- Support Vector Machines (1 hour)
- Data Preprocessing in Machine Learning (1 hour)
- Decision Trees (1 hour)
- Ensemble Learning and Random Forests (1 hour)
- Bagging and Boosting (2 hours)
- Techniques to Resolve Class Imbalance (2 hours)
- Application (5 hours)

17.0.3 Welcome to Predicting Credit Risk



17.1.1 Create a Machine Learning Environment

The lead data scientist, Jill, has asked you to create the same development environment that she is using. This step will help you run the code smoothly without conflicts.

Your new virtual environment will use Python 3.7 and accompanying Anaconda packages. After creating the new virtual environment, you'll install the imbalanced-learn library in that environment.

NOTE

> Consult the **imbalanced-learn documentation (https://imbalanced-learn.readthedocs.io/en/stable/)** for additional information about the imbalanced-learn library.

---

macOS Setup

Before we create a new environment in macOS, we'll need to update the global conda environment:

1. If your PythonData environment is activated when you launch the command line, deactivate the environment.

REWIND

To deactivate an active environment, type `conda deactivate`.

2. Update the global conda environment by typing `conda update conda` and press Enter.
3. After all the packages are collected, you'll see the prompt `Proceed ([y]/n)?`. Press the "Y" key (for "yes") and press Enter.
4. In the command line, type `conda create -n mlenv python=3.7 anaconda`. The name of your new environment is mlenv.
5. After all the packages are collected, you'll see the prompt `Proceed ([y]/n)?`. Press the "Y" key (for "yes") and press Enter.
6. Activate your mlenv environment by typing `conda activate mlenv` and press Enter.

Check Dependencies for the imbalanced-learn Package

Before we install the imbalanced-learn package, we need to confirm that all of the package dependencies are satisfied in our mlenv environment:

- NumPy, version 1.11 or later

- SciPy, version 0.17 or later
- Scikit-learn, version 0.21 or later

What is one method you'd use to confirm you met the following three dependencies in your mlenv environment?

- NumPy, version 1.11 or later
- SciPy, version 0.17 or later
- Scikit-learn, version 0.21 or later

( ) Type `conda list` and press Enter to check the version number of each dependency.

( ) Type the dependency `<dependency> --version` and press Enter to check the version number.

Check Answer

Finish ▸

On the command line, you can check all packages that begin with `numpy`, `scipy`, and `scikit-learn` when you type `conda list | grep` and press Enter. The `grep` command will search for patterns of the text `numpy` in our conda list. For example, when we type `conda list | grep numpy` and press Enter, the output should be as follows:

```
numpy                     1.17.4           py37h890c691_0
numpy-base                1.17.4           py37h6575580_0
numpydoc                  0.9.1                      py_0
```

As you can see, our `numpy` dependency meets the installation requirements for the imbalanced-learn package.

Additionally, you can type `python` followed by the command argument `-c`, and then `"import `package_name`;print(`package_name`.__version__)"` to verify which version of a package is installed in an environment, where `` `package_name` `` is the name of the package you want to verify.

Type `python -c "import numpy ;print(numpy.__version__)"` and then press Enter to see the version of `numpy` in your mlenv environment.

Windows Setup

Before we create a new environment in Windows, we'll need to update the global conda environment:

1. Launch the Anaconda Prompt, or open your `PythonData` Anaconda Prompt and deactivate this environment.

REWIND

To deactivate an active environment, type `conda deactivate`.

2. Update the global conda environment by typing `conda update conda` and press Enter
3. After all the packages are collected, you'll see the prompt `Proceed ([y]/n)?`. Press the "Y" key (for "yes") and press Enter.
4. In the command line, type `conda create -n mlenv python=3.7 anaconda`.
5. After all the packages are collected, you'll see the prompt `Proceed ([y]/n)?`. Press the "Y" key (for "yes") and press Enter.

6. Activate your mlenv environment by typing `conda activate mlenv` and press Enter, or open your Anaconda Prompt (mlenv).

Check Dependencies for the imbalanced-learn Package

Before we install the imbalanced-learn package, we need to confirm that all of the package dependencies are satisfied in our mlenv environment:

- NumPy, version 1.11 or later
- SciPy, version 0.17 or later
- Scikit-learn, version 0.21 or later

What is one method you'd use to confirm you met the following three dependencies in your mlenv environment?

- Numpy, version 1.11 or later
- SciPy, version 0.17 or later
- Scikit-learn, version 0.21 or later

( ) Type `conda list` and press Enter to check the version number of each dependency.

( ) Type the dependency `<dependency> --version` and press Enter to check the version number.

Check Answer

Finish ▸

In the Anaconda Prompt, you can check all packages that begin with `numpy`, `scipy`, and `scikit-learn` when you type `conda list | findstr` and press Enter. The `findstr` command will search for patterns of the text in our conda list. For example, when we type `conda list | findstr numpy` and press Enter, the output should be as follows:

```
numpy                     1.16.5           py37h19fb1c0_0
numpy-base                1.16.5           py37hc3f5095_0
numpydoc                  0.9.1            py_0
```

From the output, we can see that our `numpy` dependency meets the installation requirements for the imbalanced-learn package.

Additionally, you can type `python` followed by the command argument `-c`, and then `"import `package_name`;print(`package_name`.__version__)"` to verify which version of a package is installed in an environment, where `` `package_name` `` is the name of the package you want to verify:

Type `python -c "import numpy;print(numpy.__version__)"` and press Enter to see the version of `numpy` in your mlenv environment.

SKILL DRILL

Determine what version of `scipy` and `scikit-learn` you have in your mlenv environment.

Hint: For Windows, you may need to quit the mlenv Anaconda Prompt and launch it again to use `python -c "import <package_name>;print(<package_name>.__version__)"`.

NOTE

> By updating our global conda environment, the `numpy`, `scipy`, and `scikit-learn` dependencies should meet the requirements for installing the imbalanced-learn package.

NOTE

> If you encounter an error while starting Jupyter Notebook, you may need to install the `environment_kernels` module with `pip install environment_kernels`

Install the imbalanced-learn Package

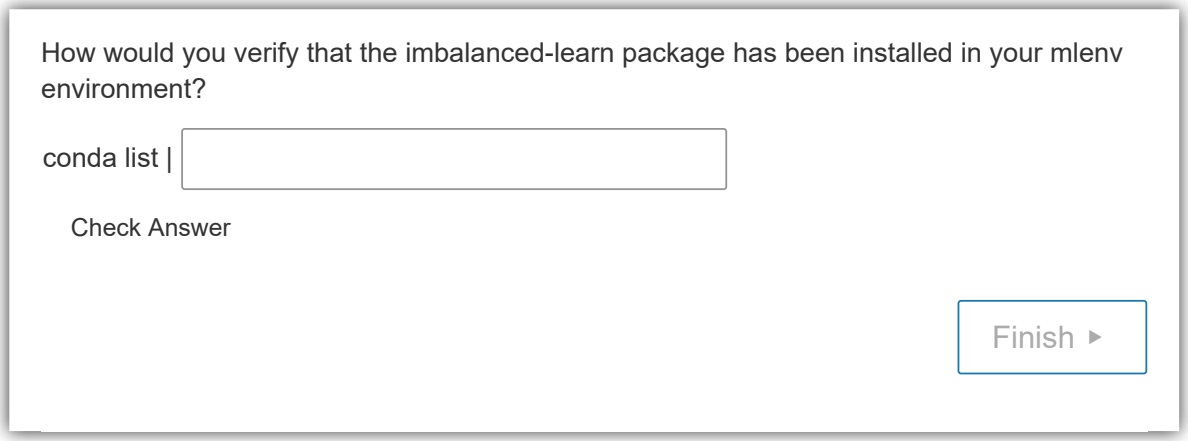
Now that our dependencies have been met, we can install the imbalanced-learn package in our mlenv environment.

With the mlenv environment activated, either in the Terminal in macOS or in the Anaconda Prompt (mlenv) in Windows, type the following:

```
conda install -c conda-forge imbalanced-learn
```

Then press Enter.

After all the packages are collected, you'll see the prompt `Proceed ([y]/n)?`. Press the "Y" key (for "yes") and press Enter.



Add the Machine Learning Environment to Jupyter Notebook

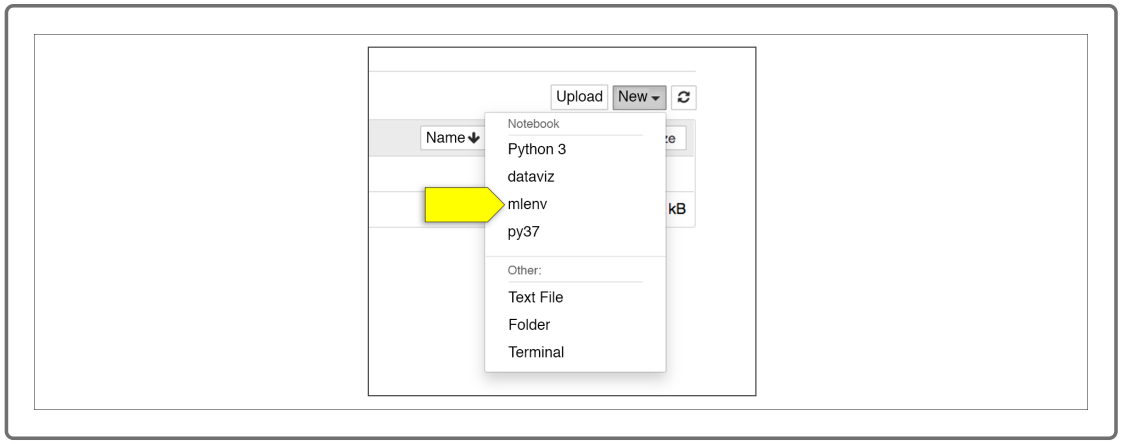
To use the mlenv environment we just created in the Jupyter Notebook, we need to add it to the kernels. In the command line, type `python -m`

`ipykernel install --user --name mlenv` and press Enter.

REWIND

The command `python -m ipykernel install --user --name mlenv` tells Python to use the IPython kernel to install the mlenv environment in the Jupyter kernels. A **kernel** is a computer program that runs and examines the Python code. A kernel interfaces between the application and your computer memory.

To check if the mlenv is installed, launch the Jupyter Notebook and click the "New" dropdown menu:



Now we can begin our machine learning journey.

17.2.1 Overview of Machine Learning

You're excited to get started on your machine learning journey. However, before using machine learning to solve problems, let's look at the most common methods employed.

Machine learning is the use of statistical algorithms to perform tasks such as learning from data patterns and making predictions. There are many different models—a model is a mathematical representation of something that happens in the real world—and you'll learn about several this week.

Broadly speaking, machine learning can be divided into three learning categories: supervised, unsupervised, and deep. For our purposes, we'll only discuss supervised and unsupervised learning.

Supervised Learning

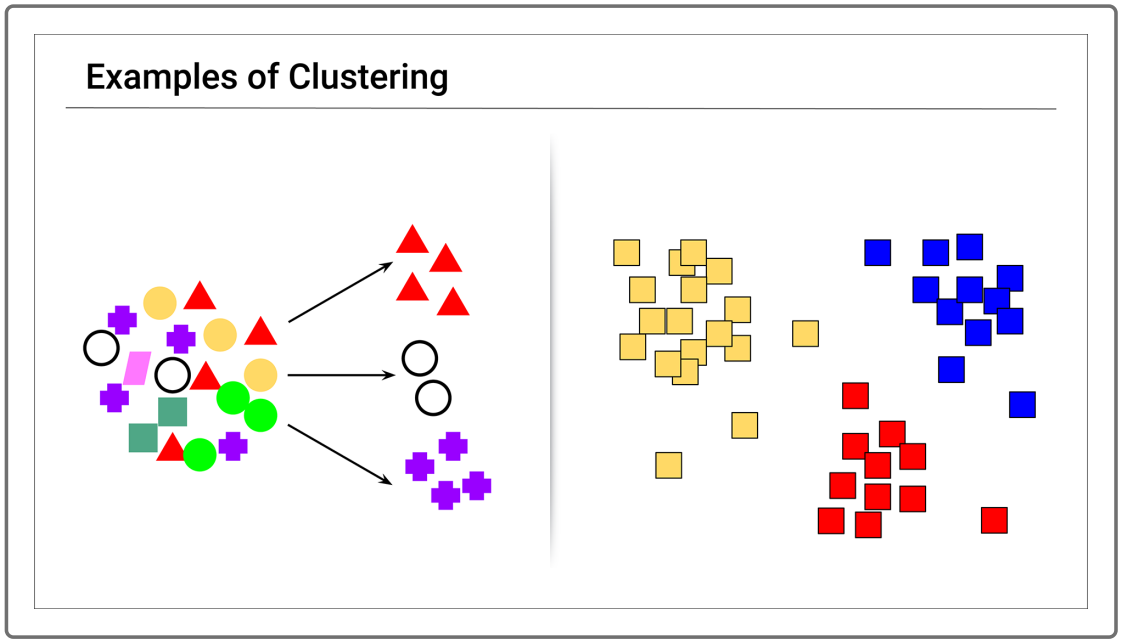
Supervised learning deals with labeled data. An example of supervised learning might be to predict, based on data from previous patients, whether a new patient has diabetes.

Patient ID	Age	Body Mass Index	Has Diabetes
1	31	24	No
2	68	39	Yes
3	57	35	?

In the simplified example above, we know whether or not the first two patients have diabetes. That is, the dataset is labeled. Each row represents a patient, and the "Has Diabetes" column is the label that informs whether or not the patient has diabetes. The status of the third patient is unknown, and based on this patient's age and BMI, the goal is to predict whether or not this person is diabetic.

Unsupervised Learning

In **unsupervised learning**, by contrast, machine learning algorithms work with datasets without labeled outcomes. In supervised learning, the labels provide the correct answers. In unsupervised learning, such correct answers, or labels, aren't provided. An example of unsupervised learning might be to task a machine learning algorithm with grouping a bag of objects as it sees fit. The algorithm isn't given labels, so it's on its own to find patterns. In this case, it might group them based on shapes, colors, or perhaps both:



A common application of unsupervised learning is to group customers by purchasing patterns.

You'll learn more about unsupervised learning in the next module.

Based on a user's previous feedback, a spam filter classifies incoming mail as spam or not spam. Is this an example of supervised or unsupervised learning?

- Supervised learning
- Unsupervised learning

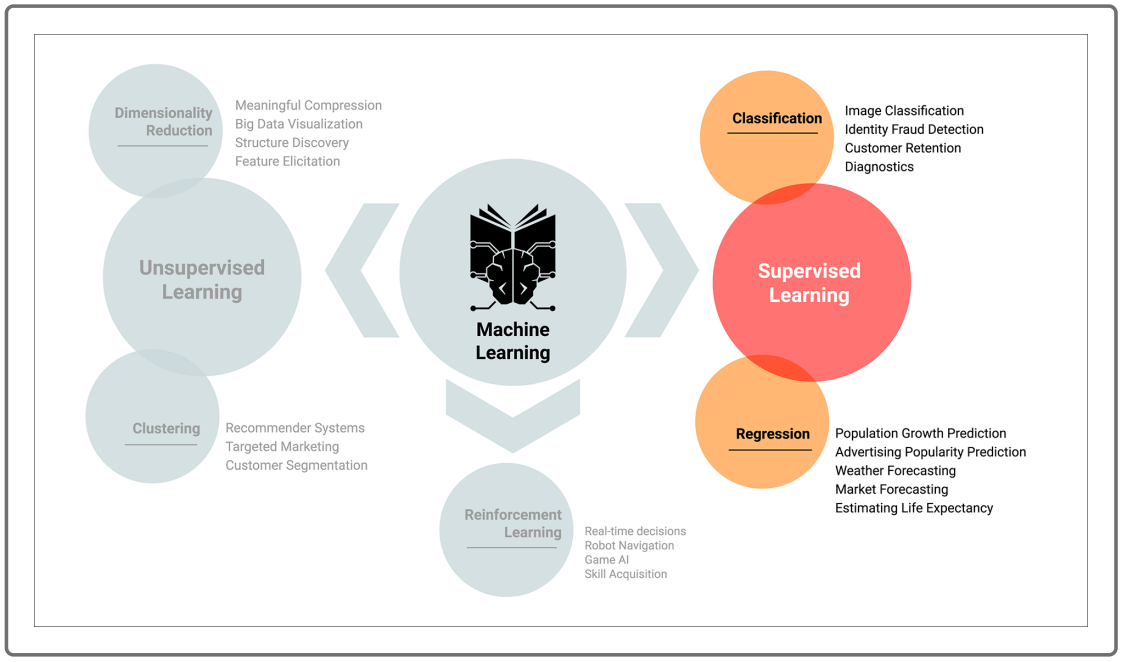
Check Answer

Finish ▸

17.2.2 Supervised Learning: Regression and Classification

After getting an overview of supervised learning and unsupervised learning, you would like to find out more about supervised learning. Jill, your data scientist boss, asks you to research the difference between regression and classification models, then report back to her. Here is what you find out.

Supervised learning can be broadly divided into regression and classification. In this module, we will focus mainly on using supervised machine learning for classification:



Regression

Recall that **regression** is used to predict continuous variables. For example, let's say that we're interested in predicting a person's weight based on factors like height, dietary preferences, and exercise patterns. To accomplish this task, we would collect data on a number of people. The regression model's algorithms would attempt to learn patterns that exist among these factors. If presented with the data of a new person, the model would make a prediction of that person's weight, based on previously learned patterns from the dataset.

REWIND

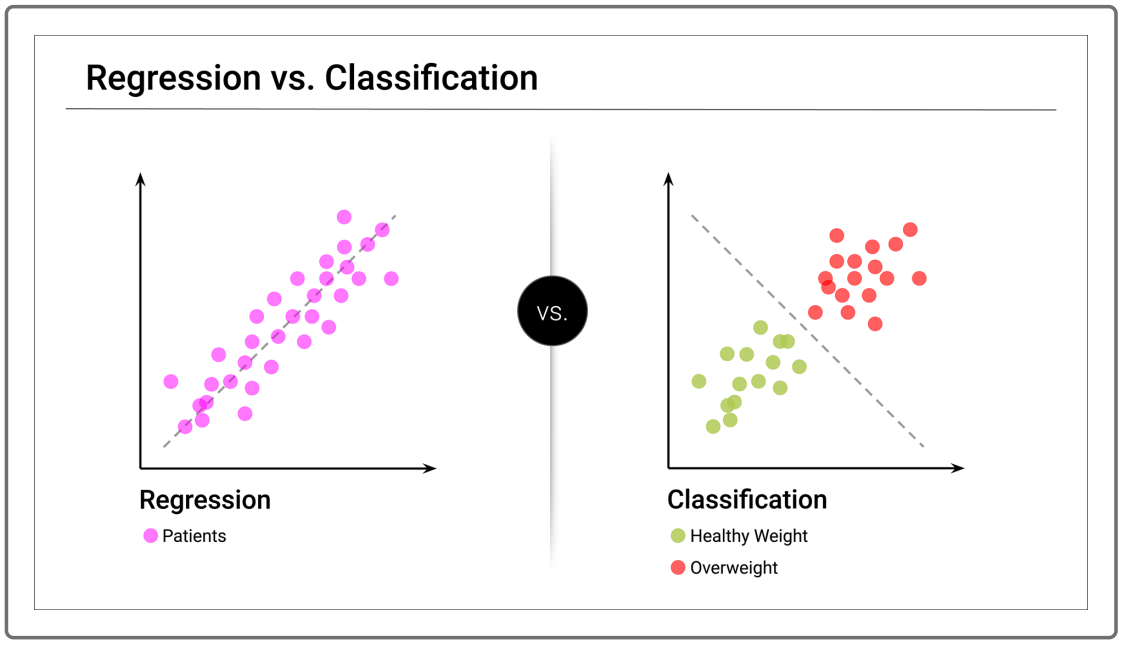
Regression is used to predict continuous variables.

Classification

Classification, on the other hand, is used to predict discrete outcomes. For example. Let's say that we are interested in using a person's traits, such as age, sex, income, and geographic location, to predict how she or he will vote on a particular issue. The outcome, in this case, is whether the person will vote "Yes" or "No." The classification model's algorithms would attempt to learn patterns from the data, and if the model is successful, gain the ability to make accurate predictions for new voters.

Regression vs. Classification

There is a major difference between regression and classification models. In our regression example, the target variable, or what we're trying to predict, is weight. Weight is a continuous variable—a person's weight can be any numerical value within a certain range. In our classification example, on the other hand, the target variable only has two possible values; whether a person votes "Yes" or "No." When the classification model encounters new data, it would attempt to predict whether the votes cast by people will be "Yes" or "No."



NOTE

> In both classification and regression problems, a dataset is divided into features and target. **Features** are the variables used to make a prediction. **Target** is the predicted outcome.

You want to create an algorithm that can read the text of an email and determine whether it is spam or not. Would you use a regression or a classification model?

- ○ Classification
- ○ Regression

Check Answer

Based on factors like the number of a customer's previous purchases, as well as demographics, an online retailer would like to predict how much the customer will spend in the next 12 months. Would a regression or a classification model be more appropriate?

- ○ Classification
- ○ Regression

Check Answer

Finish ▸

In the examples above, we saw a basic pattern:

- A machine learning model is presented with a dataset.
- The model algorithms analyze the data and attempt to identify patterns.
- Based on these patterns, the model makes predictions on new data.

This pattern applies whether we're using regression or classification. Let's look at an example of creating and using a machine learning model.

17.2.3 Linear Regression Example

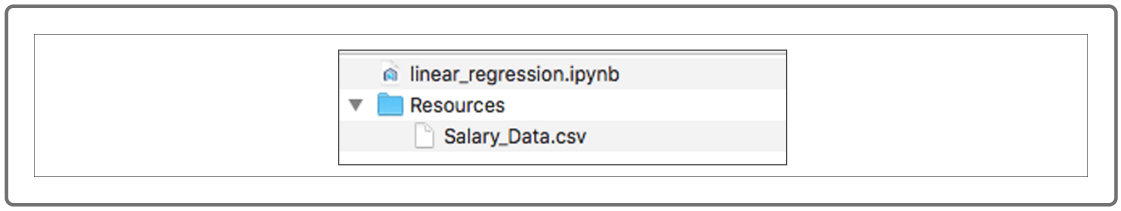
Jill is impressed with your clear explanation of the two main uses of supervised learning: regression and classification. She would like you to now learn how to implement a machine learning model in Python. You will use Scikit-learn, a Python machine learning library. Since you are already familiar with using linear regression, Jill suggests that you implement a linear regression model with Scikit-learn.

To get started, download the CSV file with salary data.

Download Salary_Data.csv (https://courses.bootcampspot.com/courses/138/files/22552/download?wrap=1) 
(https://courses.bootcampspot.com/courses/138/files/22552/download?wrap=1)

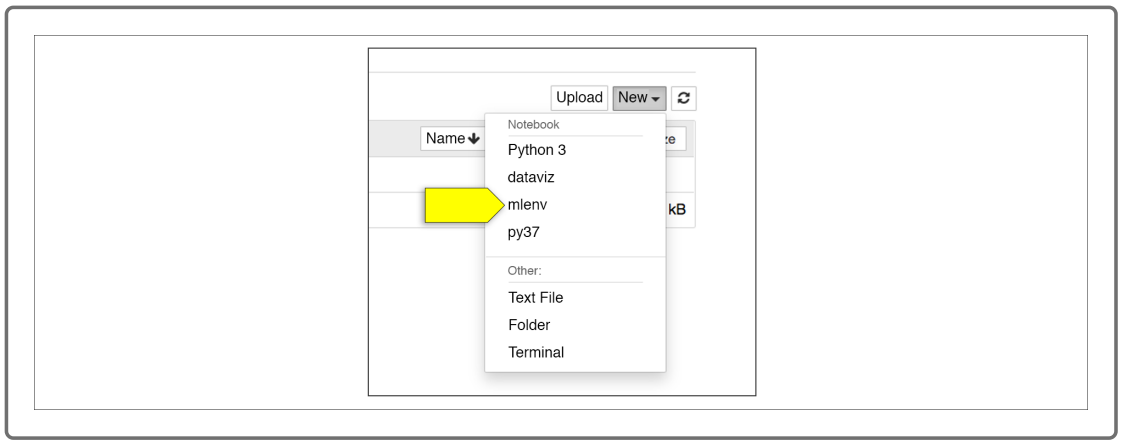
As we go through the following example, you'll learn the basic procedure for implementing a supervised learning model: create a model, train the model, and then create predictions.

Create a directory named “linear_regression_salary.” In this directory, create another directory called “Resources,” into which you’ll copy the CSV file that you downloaded:



Navigate to the directory and start the Jupyter Notebook.

Create a new notebook, and be sure to select the mlenv kernel for the notebook:



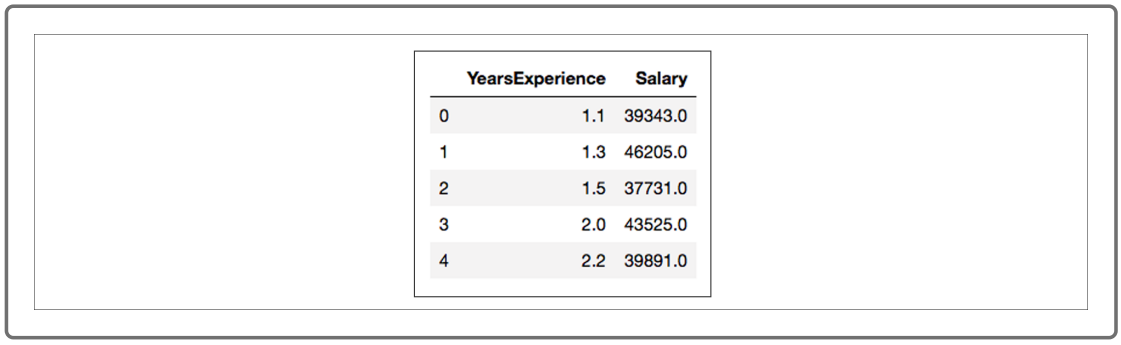
Paste the following code into the first cell and run it. (Each code block from this point on will be a separate cell in your notebook.)

```
import pandas as pd
from pathlib import Path
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
```

`sklearn` is the Scikit-learn machine learning library for Python. It has many modules, including one for linear regression, which we use here.

In the next cell, load the CSV file as a Pandas DataFrame and preview the DataFrame:

```
df = pd.read_csv(Path('./Resources/Salary_Data.csv'))
df.head()
```



	YearsExperience	Salary
0	1.1	39343.0
1	1.3	46205.0
2	1.5	37731.0
3	2.0	43525.0
4	2.2	39891.0

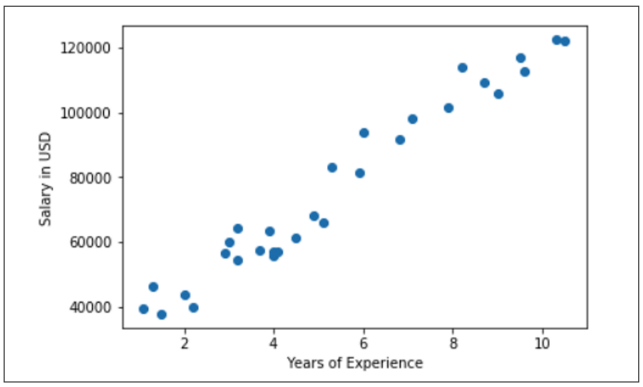
Use the Path library to locate the data file. Its use is not required, but it does make it easier to identify the file path whether you use a Mac or a Windows machine.

Each row in the DataFrame represents an employee and has two columns containing that employee's years of work experience and salary.

The target variable is `Salary`, meaning that the goal of the linear regression model is to predict a person's salary based on years of work experience.

First, let's visually inspect the relationship between `Years of Experience` and `Salary`:

```
plt.scatter(df.YearsExperience, df.Salary)
plt.xlabel('Years of Experience')
plt.ylabel('Salary in USD')
plt.show()
```



What is the relationship between the two variables, salary and experience?

- There is a strong relationship between the two variables, where the salary decreases as the amount of experience increases.
- There is a weak relationship between the two variables, where the salary increases as the amount of experience increases.
- There is a strong relationship between the two variables, where the salary increases as the amount of experience increases.
- There is a weak relationship between the two variables, where the salary decreases as the amount of experience increases.

Check Answer

Finish ▸

However, we want to do better than merely establish that there's a relationship. We want to use the available data to make the most accurate predictions possible.

The next line of code formats the data to meet the requirements of the Scikit-learn library:

```
X = df.YearsExperience.values.reshape(-1, 1)
```

17.3.1 Overview of Logistic Regression

Jill now believes that you are ready to try your hand at solving a classification problem with machine learning. The first model you will use is logistic regression, a popular classification model. She explains that despite its name, logistic regression is actually not a regression model. It is a classification model. With logistic regression, it is possible to try to answer questions such as whether a credit card holder is likely to miss a payment in the next month.

Let's walk through an example of building a logistic regression model in Python. First, download the following file.

Download 17-3-1-logistic_regression_demo.zip (https://courses.bootcampspot.com/courses/138/files/22471/download?wrap=1)

In the previous linear regression example, the model attempted to predict a person's salary based on that person's years of work experience. In this model, is salary a categorical or a continuous variable?

- A categorical variable, which can take on one of a limited, and usually fixed, number of possible values.
- A continuous variable, which can take on any value, usually in a specified range.

Check Answer

Finish ▸

Logistic regression predicts binary outcomes, meaning that there are only two possible outcomes. An example of logistic regression might be to decide, based on personal information, whether to approve a credit card application. Multiple variables, such as an applicant's age and income, are assessed to arrive at one of two answers: to approve or to deny the application.

In other words, a logistic regression model analyzes the available data, and when presented with a new sample, mathematically determines its probability of belonging to a class. If the probability is above a certain cutoff point, the sample is assigned to that class. If the probability is less than the cutoff point, the sample is assigned to the other class.

In this section, we'll first walk through an example of logistic regression. Then we'll learn how the algorithm works in greater detail.

Practice Logistic Regression

Open `logistic_regression.ipynb`. In the first cell, Matplotlib and Pandas libraries are imported. In the second cell, a synthetic dataset is generated with Scikit-learn's `make_blobs` module. You do not need to focus on the details; the sole purpose here is to create two clusters of data named X and y:

```
import matplotlib.pyplot as plt
import pandas as pd

from sklearn.datasets import make_blobs
X, y = make_blobs(centers=2, random_state=42)

print(f"Labels: {y[:10]}")
print(f"Data: {X[:10]}")
```

```
print(f"Labels: {y[:10]}")
print(f"Data: {X[:10]}")
   [ 5.72293008  3.02697174]
   [-3.05358035  9.12520872]
   [ 5.461939    3.86996267]
   [ 4.86733877  3.28031244]
   [-2.14780202 10.55232269]
   [ 4.91656964  2.80035293]
   [ 3.08921541  2.04173266]
   [-2.90130578  7.55077118]
   [-3.34841515  8.70507375]
```

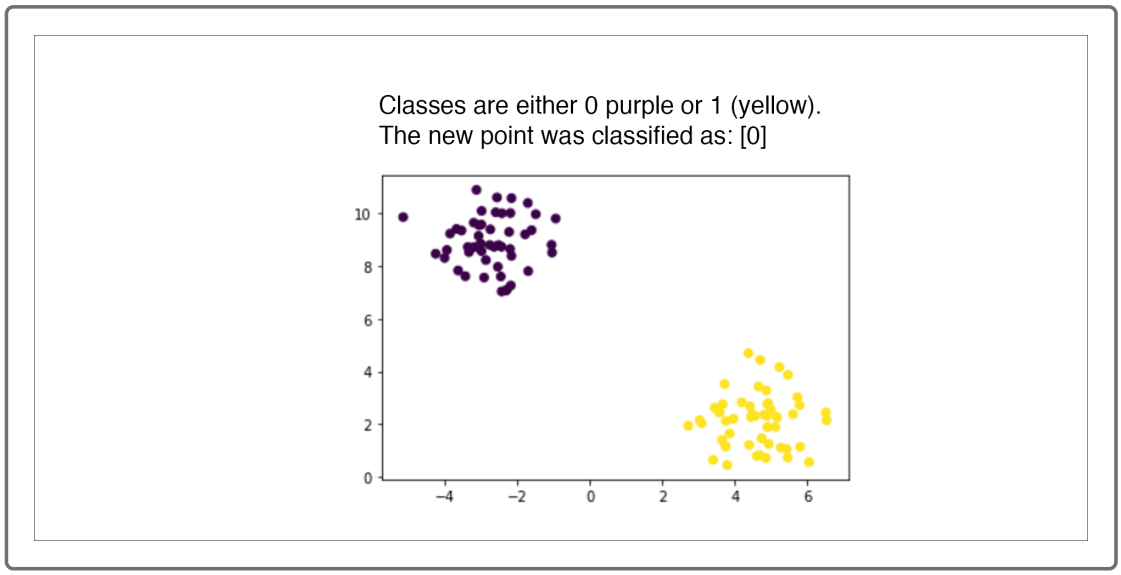
The `centers` argument specifies the number of clusters in the dataset; in this case there are two clusters. The `random_state` ensures reproducibility of this dataset: even though the numbers in this dataset are generated pseudo-randomly, specifying 42 as the `random_state` argument will generate the identical dataset in the future.

Notice also that the dataset is split into X and y arrays. X contains the coordinates of each data point, and y contains information on the class of each data point. That is, each point belongs to either class 0 or 1. Purple dots belong to class 0, while yellow dots belong to class 1. The term “labels” is used here as a synonym for target variable.

In the next cell, the dataset is visualized. We see a scatter plot with two clearly separated groups of data points. The data points to the left of 0 on

the x-axis are purple, while those to its right are colored yellow. The logistic regression model will be trained on this data, and it will be able to categorize a new data point as one of the two classes (yellow or purple):

```
plt.scatter(X[:, 0], X[:, 1], c=y)
```



Split the Dataset Into Train and Test Sets

In the next cell, we split the dataset into two: train and test datasets. Imagine that you are studying for a major exam and have a test bank of 100 questions available. A good strategy might be to use 75 of the questions as you study for the exam. These questions will help pinpoint important topics and patterns of information you might encounter on the actual exam.

Then, a few days before the exam, you sit down to answer the remaining 25 questions in a mock test. Since you haven't seen these questions before, how well you perform on them will give you a good idea of how you will do on the actual exam.

Similarly, a dataset is split into training and testing sets in supervised learning. The model uses the **training dataset** to learn from it. It then uses

the **testing dataset** to assess its performance. If you use your entire dataset to train the model, you won't know how well the model will perform when it encounters unseen data. That is why it's important to set aside a portion of your dataset to evaluate your model.

In the next cell, the dataset is split into training and testing sets:

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X,
    y, random_state=1, stratify=y)
```

Recall that X is the input, and y is the output, or what we wish to predict. Scikit-learn's `train_test_split` module takes X and y as arguments and splits each into training and test sets. So in all, we end up with four sets. X is split into `X_train` and `X_test` sets, and y is split into `y_train` and `y_test` sets.

Which Scikit-learn module is used to split a dataset into training and testing sets?

- `make_blobs`
- `test_train_split`
- `train_test_split`
- `training_testing_split`

Check Answer

Finish ▸

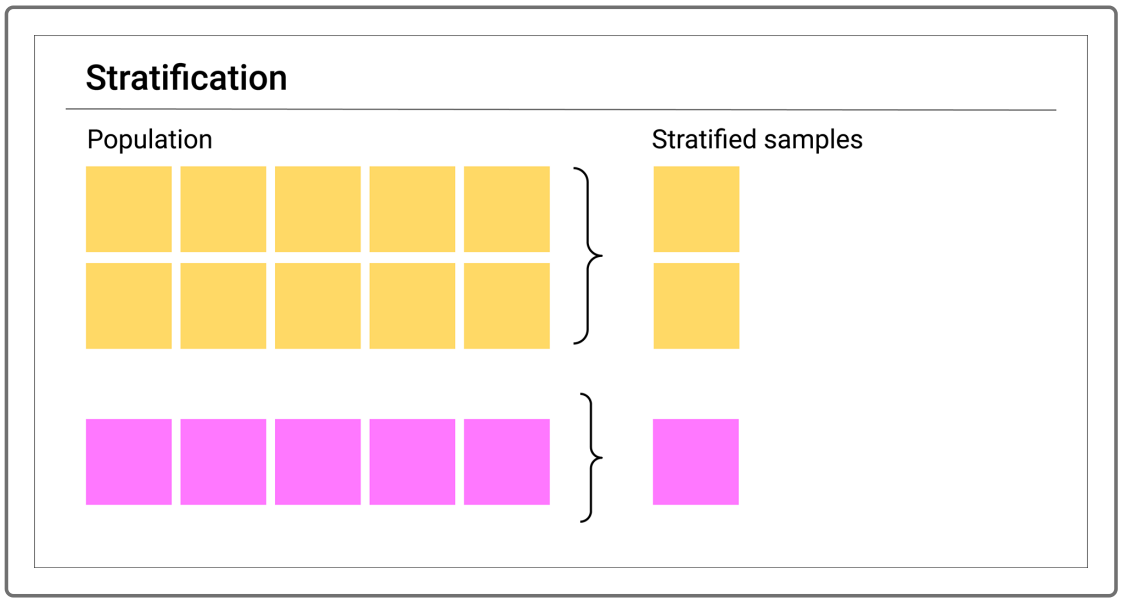
The `train_test_split()` function has four arguments here. The first two, as discussed, are X and y, which are split into train and test sets. We saw previously that `random_state` is used to make the data reproducible. Specifying a random state of 1 ensures that the same rows are assigned to train and test sets, respectively. A different random state number would

distribute different rows of data to the train and test sets. Note that the `random_state` argument is used here only for pedagogical purposes, so that you will obtain the same results shown here when you run the notebook. You will not need to use it when you create your own analyses.

NOTE

> Consult the **sklearn documentation (https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html)** for additional information about the `train_test_split()` function.

The last argument, `stratify`, also deserves discussion. Stratifying a dataset divides it proportionally. For example, say that 60% of a dataset belongs to the yellow class, and 40% of it belongs to the purple class. Stratifying it ensures that when the entire dataset is split into training and testing sets, 60% of both will belong to the yellow class, and 40% will belong to the purple class. Without specifying stratification, samples will be assigned randomly, so that it is possible to end up with a training set with a 68-32 split and a testing set with a 65-35 split, for example:



In our current dataset, both the purple and yellow clusters have roughly equal sizes, and a stratified split will not make a significant difference in

the outcome.

A dataset has 800 observations of Class A and 200 observations of Class B. Which of the following is a stratified split?

- Training set: 800. Testing set: 200
- Training set: 600. Testing set: 400
- Training set: 600 Class A, 200 Class B. Testing set: 150 Class A, 50 Class B.

Check Answer

Finish ▸

However, imagine a scenario in which one class is significantly larger than the other. For example, let's say that we're analyzing 10,000 credit card transactions, and only 40 of them are flagged as fraudulent. We might allocate 75% of the dataset to the training set, and 25% to the testing set, so we'd expect the training set to contain 75% of the fraudulent transactions (30), and the testing set to contain 25% of the fraudulent transactions (10). Without stratification, it's possible for the fraudulent transactions to be distributed disproportionately—for example, 20 to the training set and 20 to the testing set. And as the model trains on the unrepresentative data, it can reach wrong conclusions. It's therefore important to consider stratifying the data, especially when the classes are severely unbalanced, or when the dataset is small.

Choose the correct answer.

- Stratified sampling usually results in a more representative sample than random sampling.
- Stratification can be especially helpful when the dataset is small.
- Both A and B.

Check Answer

Finish ▸

IMPORTANT

> Stratification is especially important when there's a class imbalance.

Instantiate a Logistic Regression Model

In the next cell, we create a logistic regression model:

```
from sklearn.linear_model import LogisticRegression
classifier = LogisticRegression(solver='lbfgs', random_state=1)
classifier
```

```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
   intercept_scaling=1, 11_ratio=None, max_iter=100,
   multi_class='warn', n_jobs=None, penalty='12',
   random_state=1, solver='lbfgs' tol=0.0001, verbose=0,
   warm_start=False)
```

Let's break down this code:

- We first import `LogisticRegression` from the Scikit-learn library, and then instantiate the model.
- The `solver` argument is set to `'lbfgs'`, which is an algorithm for learning and optimization. The particular solver isn't very important in this example, but note that a number of optimizers exist.
- Once again, the `random_state` is specified so that you'll be able to reproduce the same results as you run this notebook.

Train the Logistic Regression Model

The next step is to train the model with the training set data. We use the `fit()` method to train the model:

```
classifier.fit(X_train, y_train)
```

Validate the Logistic Regression Model

The next step is to create predictions and assemble the results into a Pandas DataFrame:

```
predictions = classifier.predict(X_test)
pd.DataFrame({"Prediction": predictions, "Actual": y_test})
```

Let's break down this code:

- The first line of code uses the `predict()` method to create predictions based on `X_test`.
- The second line creates a DataFrame of predicted values and actual values.

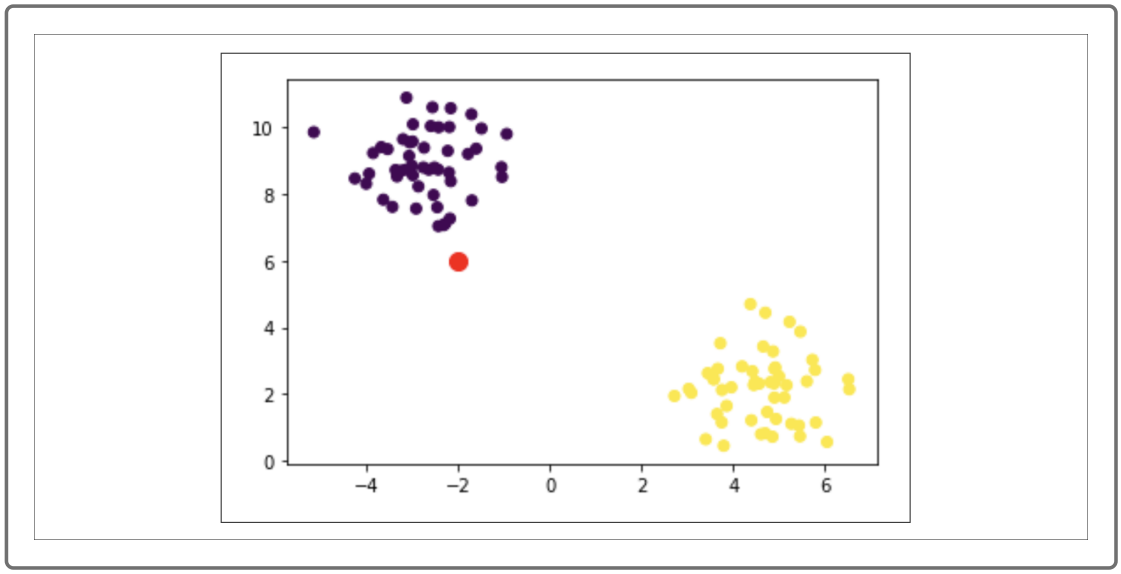
Next, we validate the model, or evaluate its performance. You can create a model and use it to make predictions, but you won't know how good the model is unless you assess its performance:

```
from sklearn.metrics import accuracy_score
accuracy_score(y_test, predictions)
```

The model achieved an accuracy score of 1.0. This means that every single observation in the testing set was predicted correctly by the model. All samples belonging to class 1 (yellow) were correctly predicted, and all samples belonging to class 0 (purple) were likewise correctly predicted by the model. Although perfect accuracy was achieved in this example, it is rare in actual practice. Moreover, an extremely high metric should raise your suspicion of overfitting. Overfitting refers to an instance in which the patterns picked up by a model are too specific to a specific dataset. We will discuss overfitting in greater detail later in the module.

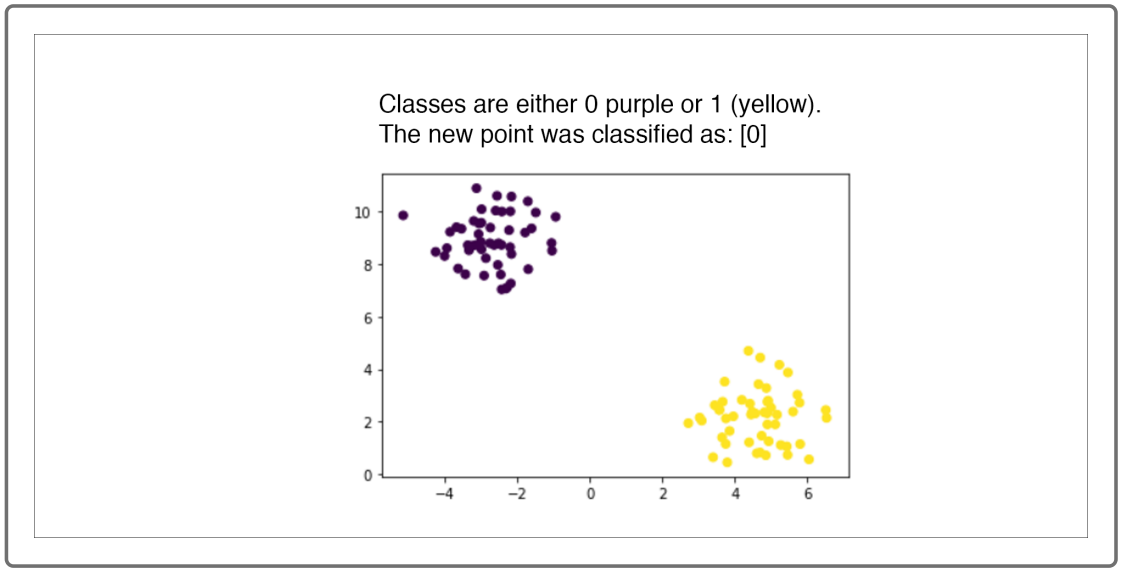
The sole purpose of the next cell is to create a new data point, which shows up as a red dot on the new plot. The logistic regression model will then classify this sample as belonging to class 0 (purple) or class 1 (yellow):

```
import numpy as np
new_data = np.array([[-2, 6]])
plt.scatter(X[:, 0], X[:, 1], c=y)
plt.scatter(new_data[0, 0], new_data[0, 1], c="r", marker="o", s=100)
plt.show()
```



Once again, we use `predict()` to predict which class the new data point belongs to. It turns out, as expected, that the red dot belongs to the purple cluster:

```
predictions = classifier.predict(new_data)
print("Classes are either 0 (purple) or 1 (yellow)")
print(f"The new point was classified as: {predictions}")
```



Let’s summarize the steps we took to use a logistic regression model:

1. Create a model with `LogisticRegression()`.
2. Train the model with `model.fit()`.
3. Make predictions with `model.predict()`.
4. Validate the model with `accuracy_score()`.

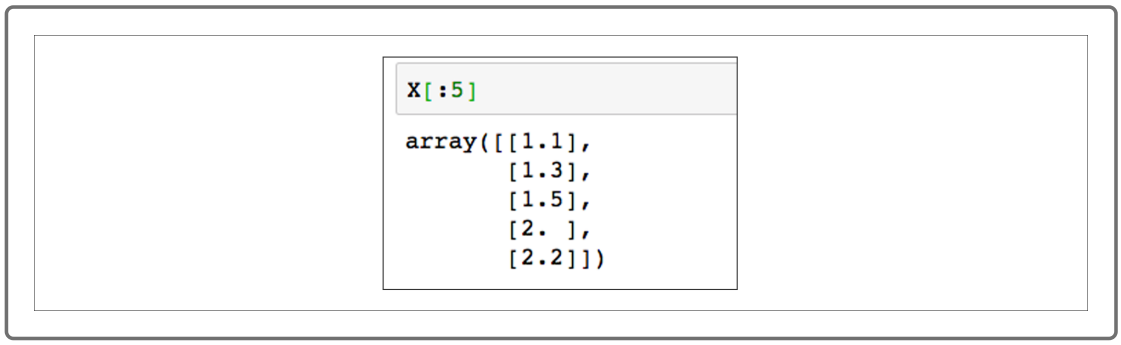
In the scatter plot, the `YearsExperience` was placed on the x-axis. Conventionally, the independent variable is placed on the x-axis, and the dependent variable is placed on the y-axis. The years of experience is the independent variable here because we assume that employee salary depends on the years of experience.

The values from the `YearsExperience` column are modified by the `reshape()` method. This method allows data needs to be formatted, or shaped, to follow Scikit-learn's specifications as follows:

- The first argument of `reshape()` specifies the number of rows. Here, the argument is -1, and means that the number of rows is unspecified. Accordingly, the NumPy library will automatically identify the number of rows in the dataset.
- The second argument of `reshape()` refers to the number of columns. Here, the argument is 1, meaning that there is only one column of independent variables.

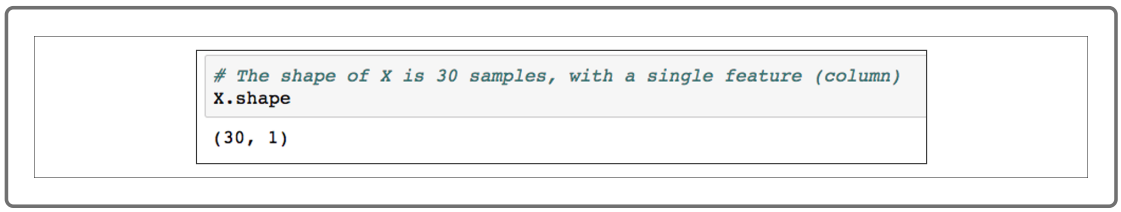
When we examine the first five entries in X, we see that the output is a two-dimensional NumPy array:

```
X[:5]
```



When we examine the shape of X, we see that there are 30 rows and 1 column:

```
X.shape
```



IMPORTANT

> Using a dataset with an incorrect shape, or formatting, can cause errors in Scikit-learn. Remember that you may have to use `reshape()` to format your data properly.

Next, we assign the target variable, or the `Salary` column, to y. Although it's possible to reshape this column, as we did with X, it's not required in this instance:

```
y = df.Salary
```

To summarize, the dataset has been separated into X and y components: X is the input data, and y is the output. You also may have noticed that X is capitalized, while y is not. Although you aren't required to stick to this convention, it's a common practice when using Scikit-learn.

The next step is to create an instance of the linear regression model. An object is instantiated, or created, from `sklearn.linear_model`'s `LinearRegression` class. Instantiation here means that the `LinearRegression` class is like a template that contains the algorithms required to perform linear regression. From this template, a specific object called `model` is created that will analyze the data and store information specific to this dataset:

```
model = LinearRegression()
```

After a model is instantiated, it will analyze the data and attempt to learn patterns in the data. This learning stage is alternatively called **fitting** or **training:**

```
model.fit(X, y)
```

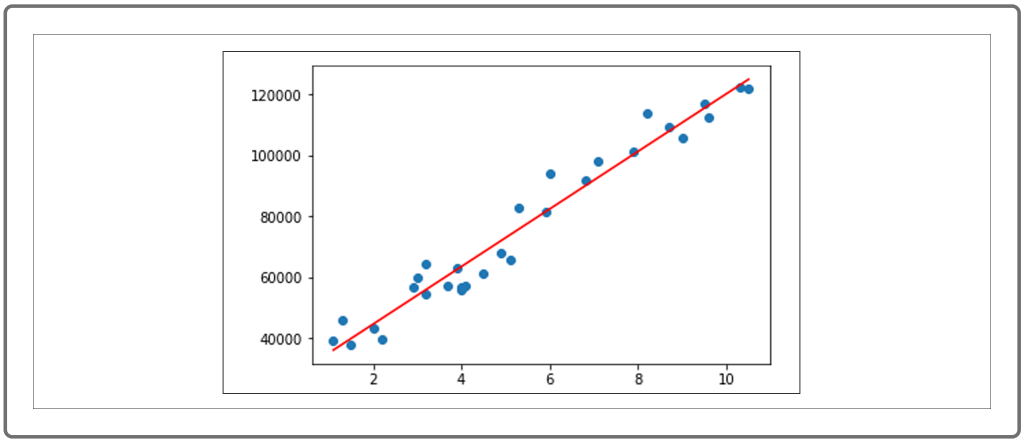
After the learning stage, the `predict()` method is used to generate predictions: given any number of a person's years of experience, the model will predict the salary:

```
y_pred = model.predict(X)
print(y_pred.shape)
```

Printing the shape of `y_pred` returns `(30,)`. Recall that a linear regression model generates a straight line that best fits the overall trend of the data. Based on the 30 available data points in X, `model.predict(X)` returned 30 predictions through which a straight line can be drawn.

Furthermore, the model can make predictions for years of experience beyond the range in the current data. Let's plot the predictions as a red line against the data points:

```
plt.scatter(X, y)
plt.plot(X, y_pred, color='red')
plt.show()
```



The best fit line is in red, drawn through the predictions. The maximum value of years of experience in the current dataset is 10.5, but the linear regression model can extrapolate beyond it. That is, if given 12 years of experience, it will be able to guess the associated salary level.

Finally, we can examine the specific parameters of our model: the slope and the y-intercept. The slope is represented by `model.coef_`, and `model.intercept_` is the y-intercept:

```
print(model.coef_)
print(model.intercept_)
```

We use the Scikit-learn [ ] method for the linear regression model to learn the patterns of the data.

Check Answer

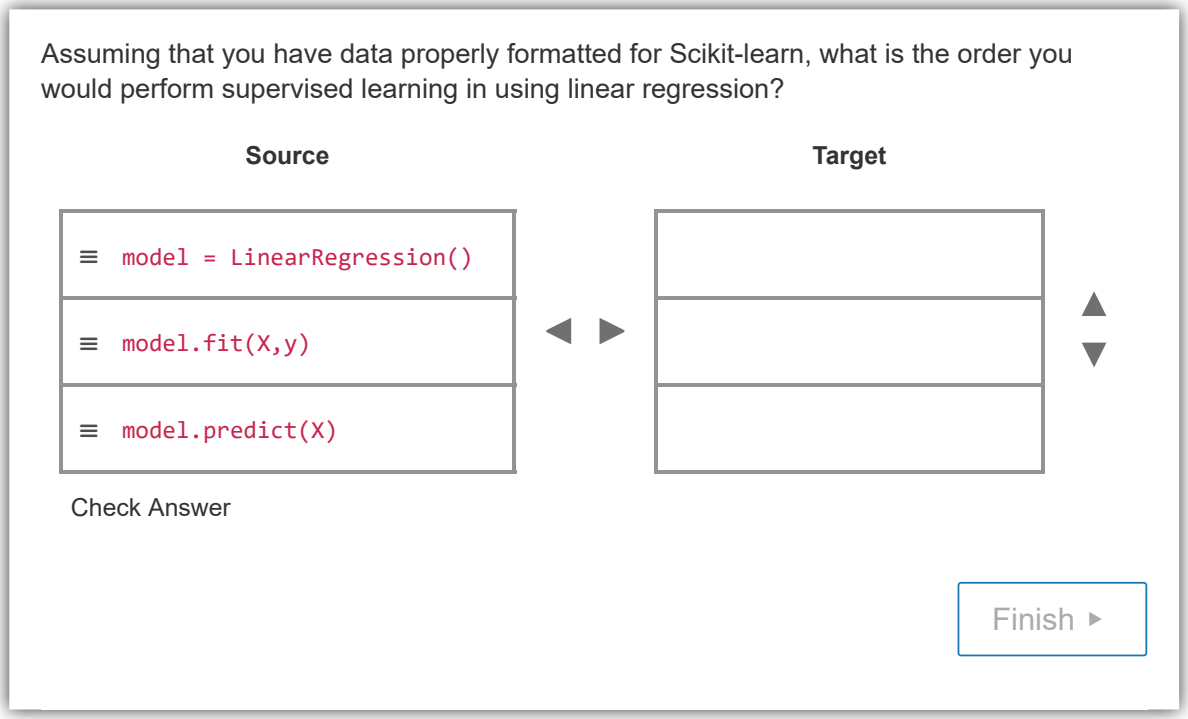
Finish ▸

Let's summarize the basic pattern for supervised learning we used in this linear regression example:

1. Split the data into input (X) and output (y).

2. Create an instance of the model with `model = LinearRegression()`.
3. Train the model with the dataset with `model.fit(X,y)`.
4. Create predictions with `y_pred = model.predict(X)`.

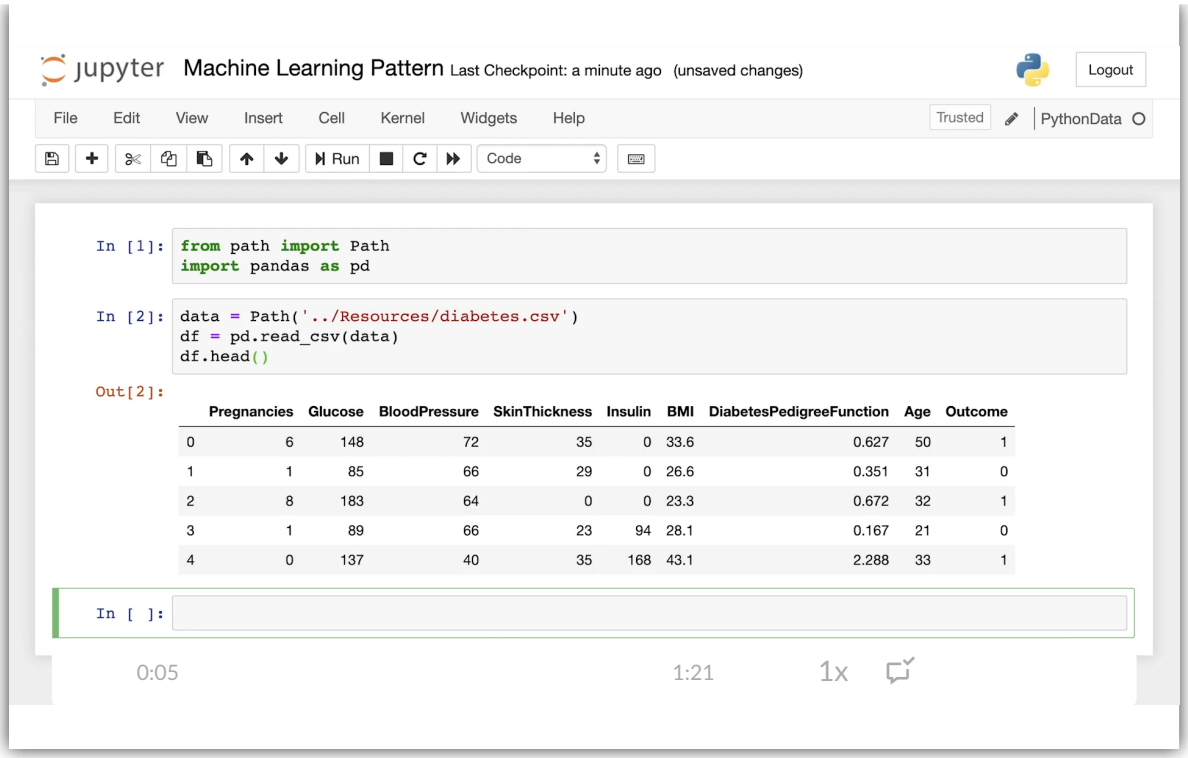
You'll encounter this pattern repeatedly in machine learning, and you'll expand on it to perform additional tasks, such as evaluating how well your model performs.



SKILL DRILL

To gain familiarity with the basics of supervised machine learning, try running the notebook again from the beginning. You may download the following Jupyter Notebook to check your work.

Download 17-2-3-linear_regression_salary.zip (https://courses.bootcampspot.com/courses/138/files/22546/download?wrap=1)



	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1

17.3.2 Logistic Regression to Predict Diabetes

Now that you've gotten your feet wet with logistic regression, Jill believes that it's time to implement a model with a real dataset. In the next step, you will follow the familiar pattern as you instantiate a model, train it, create predictions, then validate the model.

Let's solve another classification problem with logistic regression. This time, we'll use a dataset on diabetes among Pima Indian women.

First, download the files you'll need.

Download 17-3-2-logistic_regression_dm.zip (https://courses.bootcampspot.com/courses/138/files/22476/download?wrap=1)

Open `diabetes.ipynb`. We can see from the preview of the DataFrame that multiple variables (also called features), such as the number of previous pregnancies, blood glucose level, and age, can be used to predict the outcome: whether a person has diabetes (1) or does not have diabetes (0):

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1

A common task in machine learning is data preparation. In previous examples, we assigned the label X to input variables, and used them to predict y, or the output. With this diabetes dataset, we need to categorize features from the target. We can do so by separating the `Outcome` column from the other columns.

NOTE

> The terms features and variables are synonymous. Target and output are synonymous.

```
y = df["Outcome"]
X = df.drop(columns="Outcome")
```

- The `Outcome` column is defined as y, or the target.
- X, or features, is created by dropping the `Outcome` column from the DataFrame.

Later in this module, we'll be splitting the training and testing data, creating a logistic regression model, fitting the training data, and making a prediction. What steps can you do on your own in the following Skill Drills?

SKILL DRILL

Use the train_test_split module to split X and y into training and testing sets. You should end up with four

sets in total: `X_train`, `X_test`, `y_train`, `y_test`.

SKILL DRILL

Import the Scikit-learn module for logistic learning, and instantiate a model. Use the following arguments for the model:

- `solver='lbfgs'`
- `max_iter=200` (this sets an upper limit on the number of iterations used by the solver)
- `random_state=1`

SKILL DRILL

Train the model with the training dataset. Then use `X_test` to make predictions for y values.

Let's retrace the steps taken so far. We first split the dataset into training and testing sets:

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X,
   y, random_state=1, stratify=y)
```

Examining the shape of the training set with `X_train.shape` returned (576,8), meaning that there are 576 samples (rows) and eight features (columns).

The next step was to create a logistic regression model with the specified arguments for `solver`, `max_iter`, and `random_state`:

```
from sklearn.linear_model import LogisticRegression
classifier = LogisticRegression(solver='lbfgs',
   max_iter=200,
   random_state=1)
```

Next, we trained the model with the training data:

```
classifier.fit(X_train, y_train)
```

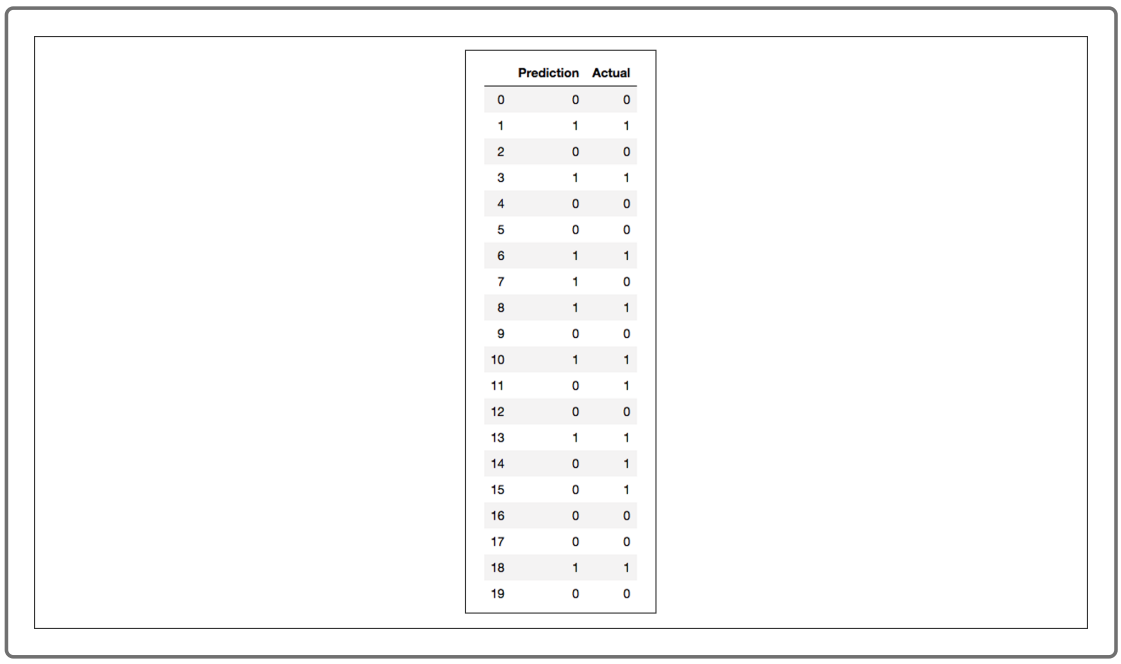
SKILL DRILL

Use the testing set to create predictions of the outcome.

To create predictions for y-values, we used the `X_test` set:

```
y_pred = classifier.predict(X_test)
```

When the first 20 rows of the predicted y-values (`y_pred`) are compared with the actual y-values (`y_test`), we see that most of the predictions are correct, but that there are also some missed predictions, such as rows 14 and 15:



	Prediction	Actual
0	0	0
1	1	1
2	0	0
3	1	1
4	0	0
5	0	0
6	1	1
7	1	0
8	1	1
9	0	0
10	1	1
11	0	1
12	0	0
13	1	1
14	0	1
15	0	1
16	0	0
17	0	0
18	1	1
19	0	0

SKILL DRILL

Use the `accuracy_score()` method module to assess the performance of the model. What will you use as the arguments for this method? What is the accuracy score? What does the accuracy score mean?

The final step is to answer an important question: how well does our logistic regression model predict? We do so with `sklearn.metrics.accuracy_score`:

```
from sklearn.metrics import accuracy_score
print(accuracy_score(y_test, y_pred))
```

This method compares the actual outcome (y) values from the test set against the model's predicted values. In other words, `y_test` are the outcomes (whether or not a woman has diabetes) from the original dataset that were set aside for testing. The model's predictions, `y_pred`, were compared with these actual values (`y_test`). The accuracy score is simply the percentage of predictions that are correct. In this case, the

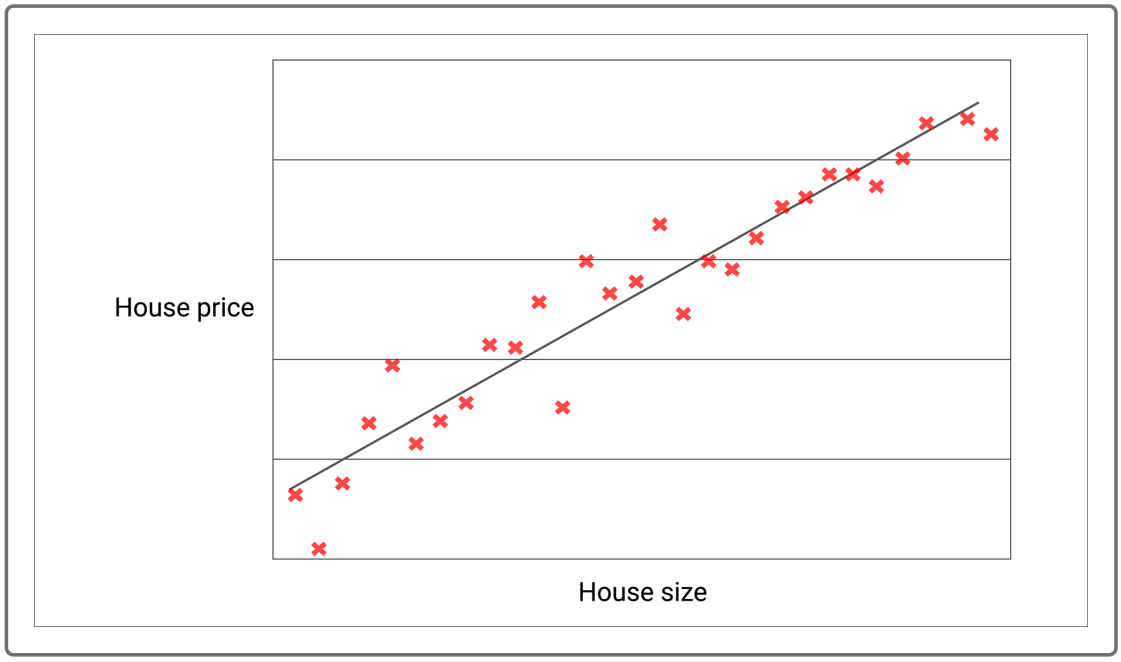
model’s accuracy score was 0.776, meaning that the model was correct 77.6% of the time.

17.3.3 How Logistic Regression Works

Jill informs you that a good data scientist not only understands the hows but the whys. She explains that understanding how a model works helps a data scientist assess a machine learning model's strengths, weaknesses, and how best to use it. She asks you to look into how a logistic regression model works.

When you protest that you haven't taken a math class in years, she reassures you that while math is indeed helpful to know, many basic underlying ideas in machine learning can be grasped without a graduate degree in math.

Before launching into a discussion of logistic regression, let's quickly review linear regression. The image below shows a scatter plot, through which a best fit line is drawn. In this case, the chart depicts the size and price of houses in a particular district:



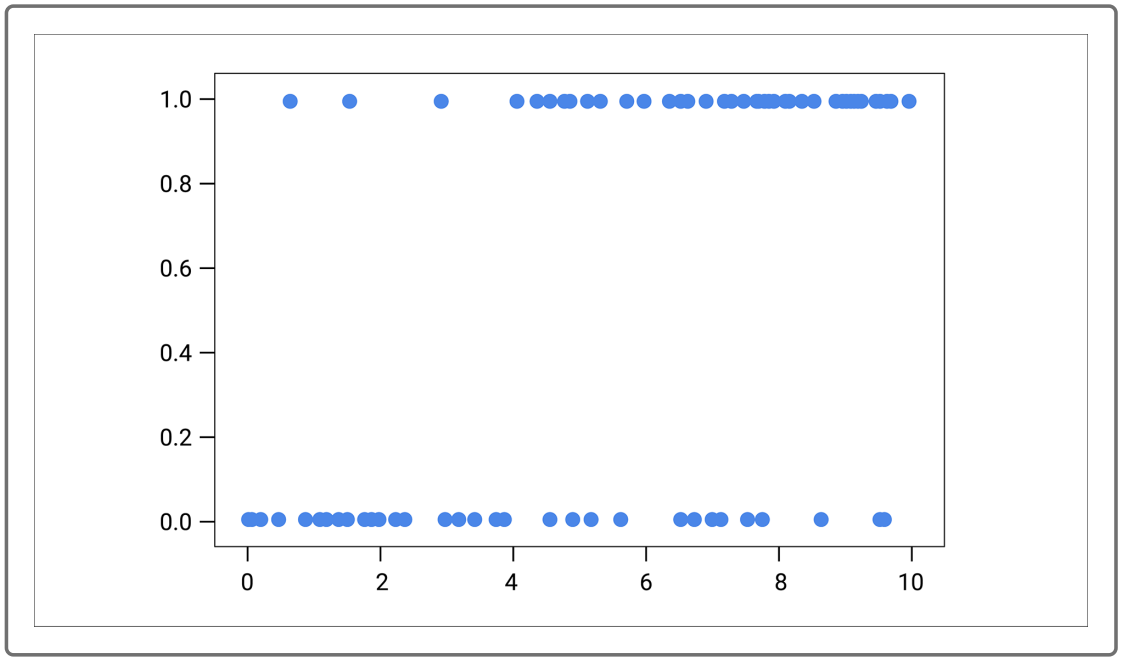
As the size of a house increases, it will generally fetch a higher price on the market. Each point represents a specific house in that district. How would you best describe the data values in the graph?

- The data values are continuous.
- The data values are discrete.

Check Answer

Finish ▸

But what happens when the outcome variable is binary, meaning that only two outcomes are possible? For example, let's say that the x-axis on the scatter plot below represents a college applicant's score on an entrance exam, and that the y-axis represents acceptance to a particular college. There is a wide range of test scores, but there are only two possible outcomes: acceptance (1) or rejection (0):

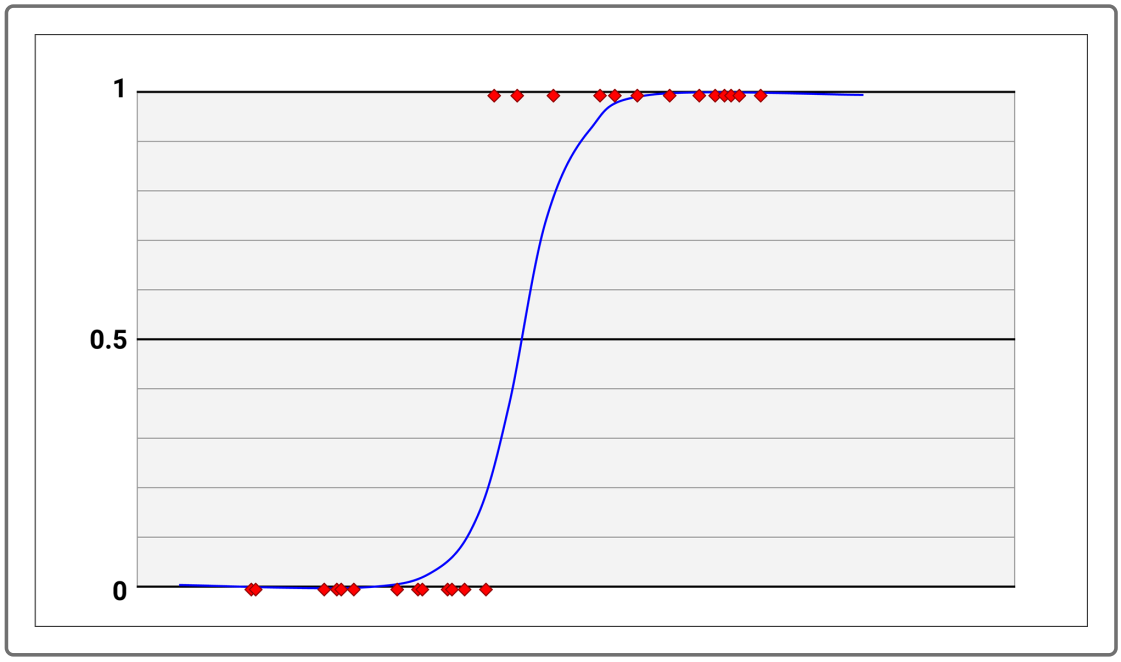


Linear regression clearly would not work in this case. Try drawing a best fit line here! A best fit line drawn through this scatter plot would be neither descriptive nor meaningful.

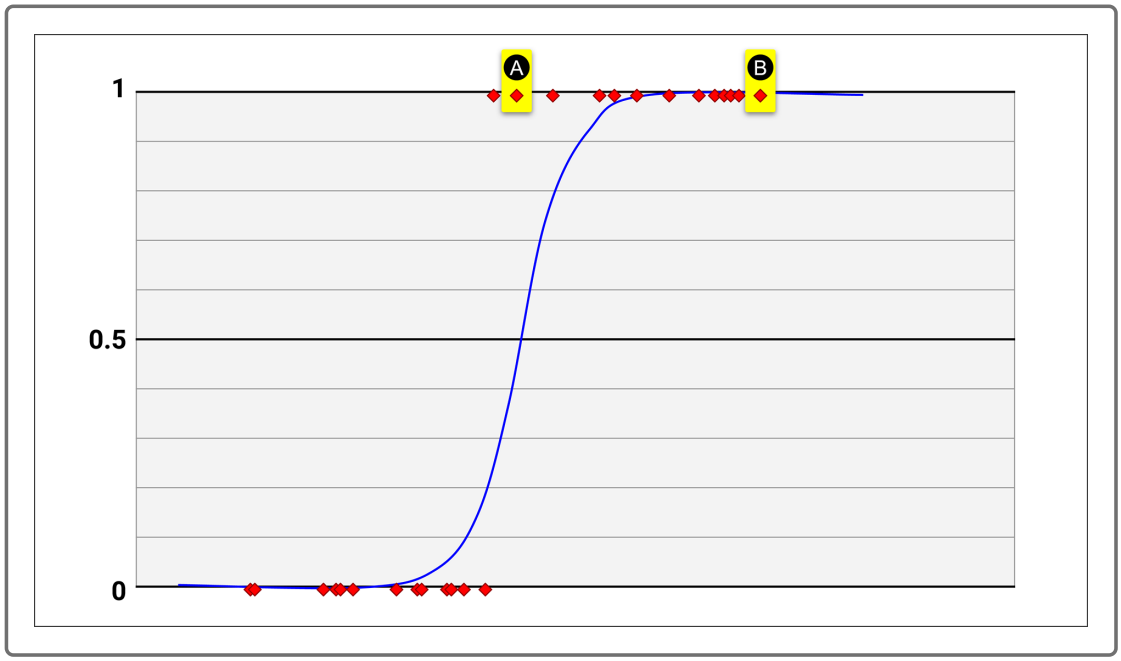
Instead, the probability of an outcome is represented with the following equation:

```
log(probability of admission/(1 - probability of admission))
```

It's important to note that the results of this equation ultimately generate an S-shaped curve that represents the probability of being admitted at a given test score:

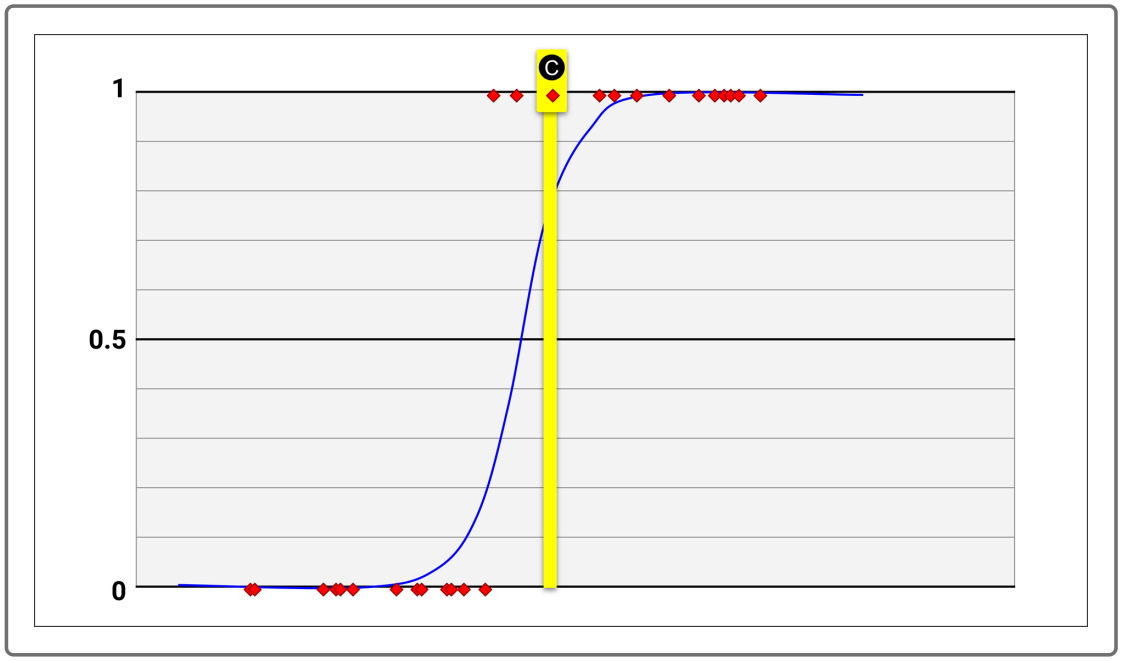


Each test score along the x-axis is associated with a probability of acceptance. In the following example, for a student with a score of A, the probability of acceptance is somewhat higher than 50%, whereas a student with a score of B has a nearly 100% chance of being admitted:



This **S-shaped curve,** also called a sigmoid curve, can then be used to predict acceptance for new applicants. The score at which the vertical line is drawn has approximately 80% probability of acceptance. Because this

value exceeds the cutoff point, which in this case is defined as 50%, it's predicted that candidates with this score will be accepted:



The company you are working for wants to filter spam emails based on certain criteria. What type of regression analysis is this?

- Linear
- Logistic

Check Answer

What type of variable would describe whether an email is spam or not?

- Continuous
- Binary

Check Answer

Finish ▸

Finally, the results are made linear with a little more mathematical manipulation. The final product is a linear equation, like the one seen in

linear regression. It is for this reason that both linear regression and logistic regression are considered linear models.

The Logit Function

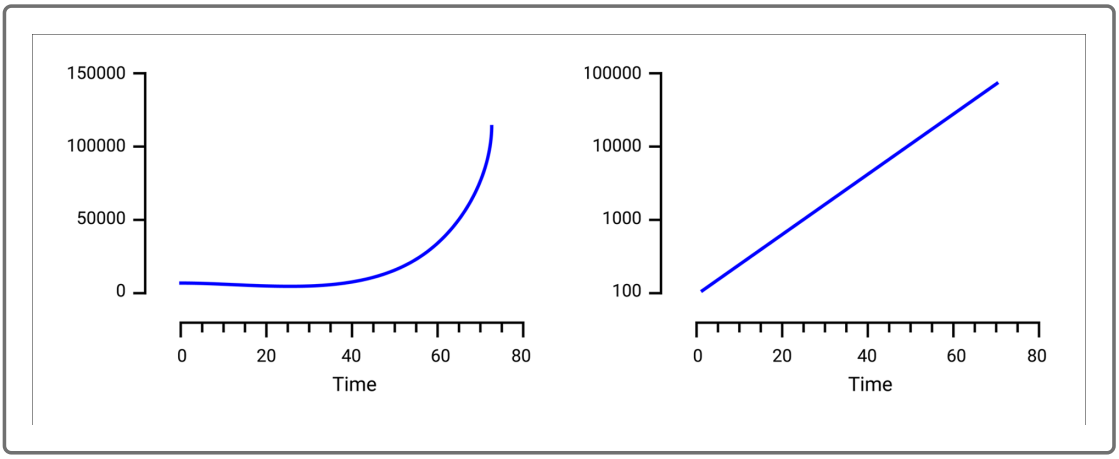
This brief mathematical discussion of logistic regression is optional. Feel free to skip this section, but keep reading if you'd like a bit more explanation behind the math.

We saw earlier that the sigmoid probability curve is generated based on the following equation, which is also called the logit function:

```
log(probability of admission/(1 - probability of admission))
```

The fraction seen here is the ratio of the probability of an occurrence and nonoccurrence. For example, let's say that applicants with a given score have a 90% probability of acceptance. Therefore, they will have a 10% probability of rejection. The logarithm of the ratio of the two probabilities is expressed as log(0.9/0.1). Based on this equation (which undergoes some rearrangement), the S-shaped curve can be created from the existing data points.

So why is a logarithm of the ratio obtained? Logarithms are useful when dealing with ratios. If you are unfamiliar or rusty with logarithms, a short description of them is that they are the opposite of exponents, just as subtractions undo additions, and divisions undo multiplications. In the following illustration, to the left is an exponential curve. In this case, as the value on the x-axis increases, its y-axis value increases rapidly. The illustration on the right shows that the curve is been straightened into a line after plotting the logarithms of the values, since logarithms undo exponents:



To understand why logarithms might be useful, consider two ratios: an even ratio and an extremely lopsided ratio. A score with an even chance (50%) of acceptance also has an even chance (50%) of being rejected. The ratio of the two probabilities is 0.5/0.5, or 1. On the other hand, an application that has 99.999% chance of being accepted has 0.001% chance of being rejected. The ratio of the two probabilities is 0.99999/0.00001 or 99999. The discrepancy between the two ratios is almost 100,000-fold! The use of logarithms smoothens out this asymmetry by scaling the numbers.

NOTE

> To learn more, consult online resources such as Khan Academy's **introduction to logarithms (https://www.khanacademy.org/math/algebra2/x2ec2f6f830c9fb89:logs/x2ec2f6f830c9fb89:log-intro/v/logarithms)** .

Logistic Regression

0:01 2:30 1x

17.4.1 Assess Accuracy, Precision, and Sensitivity

It's not enough to use a machine learning model to create predictions. The model must answer an important question: how well does it perform? You have seen that accuracy score is one way of assessing a classification model's performance. That is, what percentage of predictions does it get right?

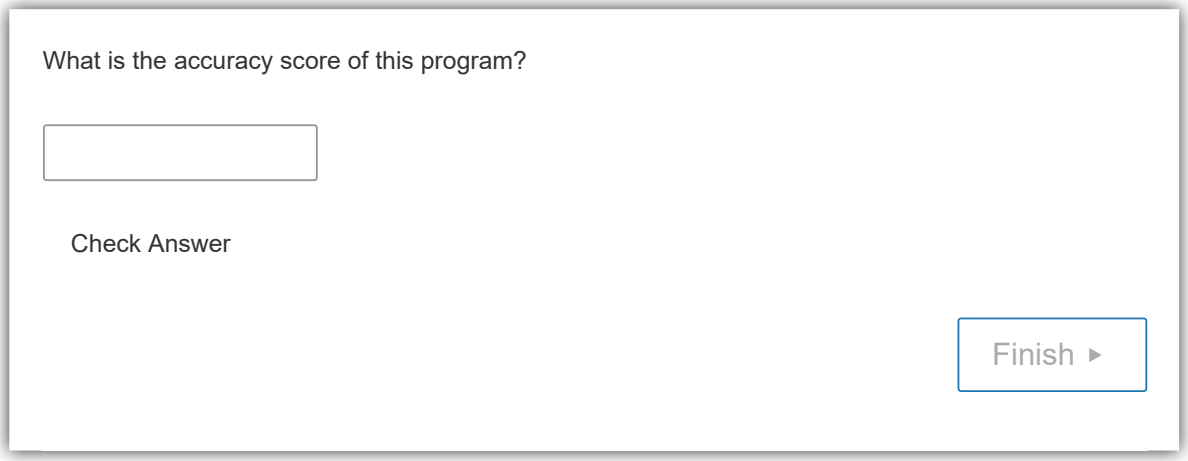
Jill explains that there are other ways to validate a classification model, and asks you to look into them. This is where the statistical rubber meets the road!

Accuracy

In an earlier module, you learned that the performance of a linear regression model is measured based on the difference between its predicted values and actual values.

However, this method cannot be used when the target values are not continuous. Different methods must be used to assess a model with discrete outcomes. We have already seen one way of validating such a model's performance: its **accuracy score.** An accuracy score is not always an appropriate or a meaningful performance metric, however.

Imagine the following scenario, in which a credit card company wishes to detect fraudulent transactions in real time. Historically, out of 100,000 transactions, 10 have been fraudulent. An analyst writes a program to detect fraudulent transactions, but due to an uncaught bug, it flags every transaction as not fraudulent. Out of 100,000 transactions, it correctly classifies the 99,990 transactions that are not fraudulent, and it erroneously classifies all 10 transactions that are fraudulent.



The program's accuracy score appears to be impressive at 99.99%. However, it fails spectacularly at its job, detecting 0 out of 10 fraudulent transactions, a success rate of 0%.

Predictions in such a scenario with binary outcomes can be categorized according to the table below:

	Predicted True	**Predicted False**
Actually True	TRUE POSITIVE	FALSE NEGATIVE
Actually False	FALSE POSITIVE	TRUE NEGATIVE

Any given prediction falls under one of two categories: true or false. In the context of fraud detection, a true prediction would mean that the model categorizes the transaction as fraudulent. A false prediction means that the model categorizes the transaction as not fraudulent.

If a transaction is predicted to be fraudulent and is really a fraudulent transaction, it is a true positive (TP).

If a transaction is predicted to be fraudulent but is not fraudulent, it is a false positive (FP). It falsely categorized an innocent transaction as fraudulent.

Similarly, if a transaction is predicted to be non-fraudulent but is actually fraudulent, it is a false negative (FN).

And when a transaction is predicted to be non-fraudulent and is in reality non-fraudulent, it is a true negative (TN).

A patient has a streptococcal infection, and a clinical test for the infection came back negative. What is the classification?

- True positive
- True negative
- False negative
- False positive

Check Answer

A 41-year-old woman takes a mammogram, which comes back positive for breast cancer. Subsequent examination of her breast tissue by a pathologist reveals that her tissue is noncancerous. What is the classification?

- True positive
- True negative
- False negative
- False positive

Check Answer

Finish ▸

Precision

Imagine that a man is experiencing weight loss and chills. He consults an online test that uses machine learning algorithms to see whether he might have cancer, which informs him that he indeed has cancer. However, the online test is not perfect. When the test was previously evaluated in a study, the results were collated into the following table, called a **confusion matrix**.

	Predicted True	**Predicted False**
Actually True	30	10
Actually False	20	40

How many people, in total, were assessed in this study?

Check Answer

How many people actually had cancer?

Check Answer

How many people were diagnosed with cancer?

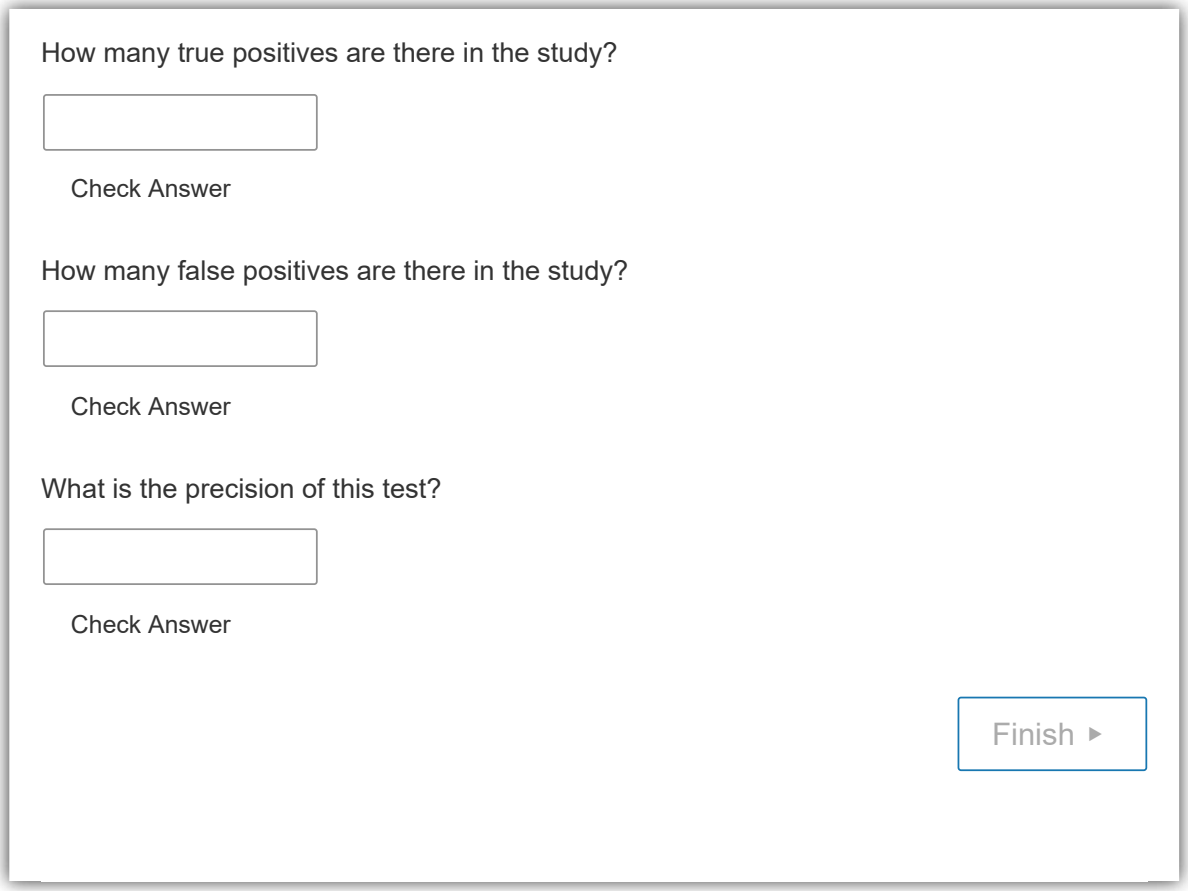
Check Answer

Finish ▸

The man in this scenario has a positive diagnosis for cancer. He wants to know how likely it is that he actually has cancer. **Precision,** also known as

positive predictive value (PPV), is a measure of this. Precision is obtained by dividing the number of true positives (TP) by the number of all positives (i.e., the sum of true positives and false positives, or TP + FP).

Precision = TP/(TP + FP)



In this study, a total of 50 people were predicted to have cancer. Of the 50, 30 people actually had cancer. The precision is therefore 30/50, or 0.6.

NOTE

> The terms precision and positive predictive value (PPV) are interchangeable.

To summarize, in machine learning, precision is a measure of how reliable a positive classification is. The following formulation may help you in

remembering precision: “I know that the test for cancer came back positive. How likely is it that I have cancer?”

Sensitivity

Another way to assess a model’s performance is with sensitivity, also called recall. While the term **recall** is more commonly used in machine learning, the two terms are synonymous and will be used interchangeably from this point.

The following formulation may help you understand sensitivity: “I know that I have cancer. How likely is it that the test will diagnose it?” Here is the formula for sensitivity:

Sensitivity = TP/(TP + FN)

In this context, all who have cancer means true positives (those who have cancer and were correctly diagnosed) and false negatives (those who have cancer and were incorrectly diagnosed as not having cancer). **Sensitivity** is a measure of how many people who actually have cancer were correctly diagnosed.

NOTE

> The terms sensitivity and recall are used interchangeably.

What is the sensitivity of this test for those who actually had cancer?

Check Answer

Which is more important in a screening test to detect cancer: precision or sensitivity?

- Precision
- Sensitivity

Check Answer

Finish ▸

Tradeoff Between Precision and Sensitivity

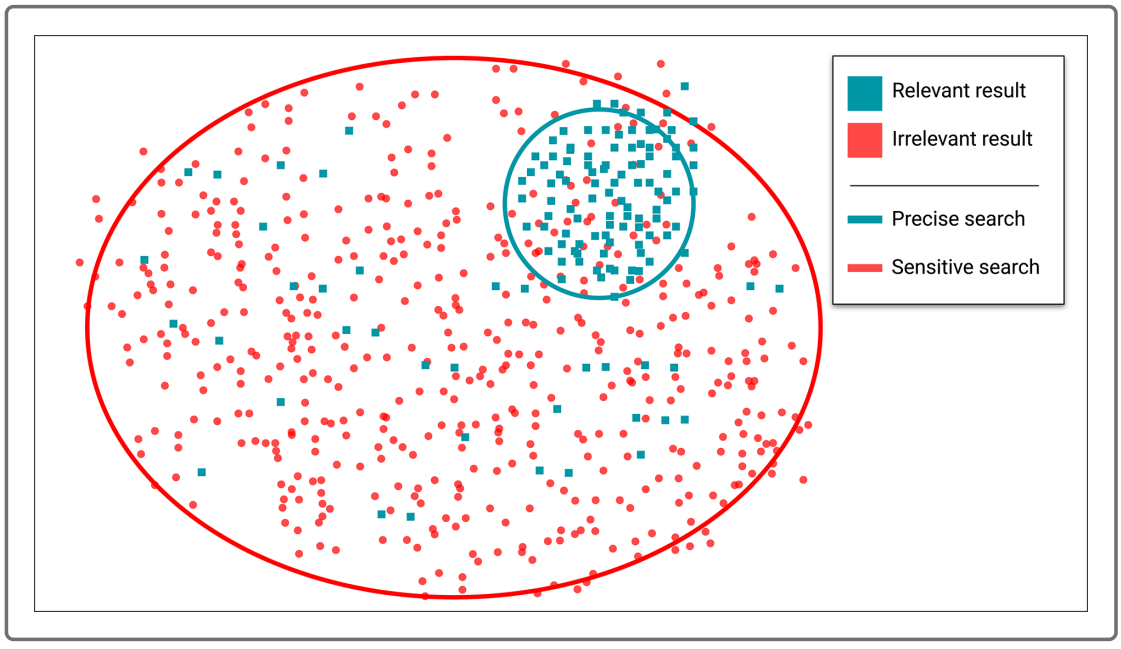
There are situations in which high sensitivity is more important. With cancer screening, for example, a high sensitivity is more important than high precision. Remember that high sensitivity means that among people who actually have cancer, most of them will be diagnosed correctly. High precision, on the other hand, means that if the test comes back positive, there's a high likelihood that the patient has cancer.

It may appear at first that the two terms refer to the same thing, but they do not. As an extreme example, let's say that among 100 people, 50 have cancer and 50 do not. A screening algorithm is extremely aggressive and labels everyone to have cancer. Since everyone who actually has cancer is detected, the sensitivity is 1.0, or 100%. However, the precision is low: being diagnosed with cancer in this case only means a 50% likelihood of actually having cancer. In other words, there are many false positives.

To return to a previous point: Why is high sensitivity more important than precision for a cancer screening test? It's better to detect everyone who might have cancer, even if it means a certain number of false positives, than to miss people who do have cancer. After all, those with a positive

result for cancer can undergo further testing to confirm or rule out cancer. The false positives in a highly sensitive test are accepted as a cost of doing business.

In contrast, there are situations in which precision is more important than sensitivity. Imagine that the criminal justice system depended on a machine learning algorithm to judge the innocence or guilt of a person on trial. Is high sensitivity or high precision preferable? Perfect sensitivity would mean that everyone who committed a crime is declared guilty. However, a potential consequence of such an aggressive algorithm is that people who didn't commit a crime are also declared guilty. Perfect precision would mean that someone who's been declared guilty actually is guilty. But it also means that there may be people who committed a crime who aren't found guilty. If the justice system values sparing innocent people false imprisonments more than punishing all guilty criminals, precision trumps sensitivity.



When using a machine learning algorithm to detect fraudulent credit card transactions, which is more important between sensitivity and precision?

- Precision
- Sensitivity

Check Answer

A political campaign has a database of potential donors. The role of the phone bank is to call potential donors for contributions. Due to limited time and staffing, however, not everyone on the database can be telephoned, and a machine learning algorithm is used to sort the list into likely and unlikely donors. Is sensitivity or precision more important in this case?

- Precision
- Sensitivity

Check Answer

In an algorithm for spam email detection, which is more important: precision or sensitivity?

- Precision
- Sensitivity

Check Answer

Finish ▸

In summary, there's a fundamental tension between precision and sensitivity. Highly sensitive tests and algorithms tend to be aggressive, as they do a good job of detecting the intended targets, but also risk resulting in a number of false positives. High precision, on the other hand, is usually the result of a conservative process, so that predicted positives are likely true positives; but a number of other true positives may not be predicted. In practice, there is a trade-off between sensitivity and precision that requires a balancing act between the two.

F1 Score

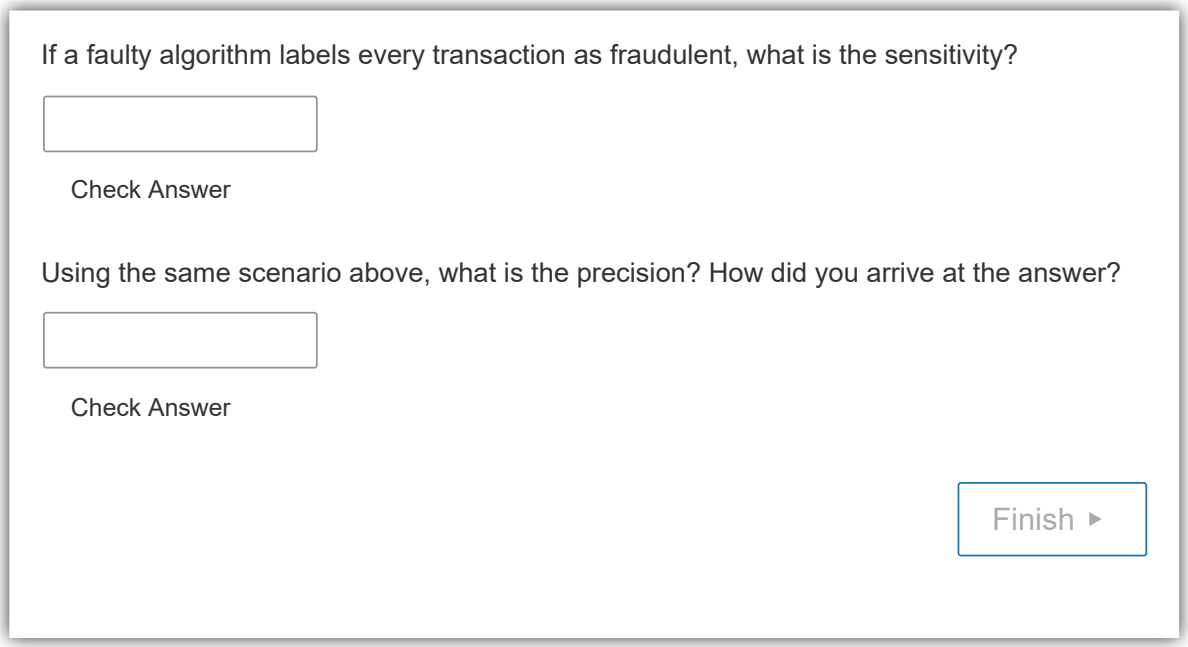
The F1 score, also called the harmonic mean, can be characterized as a single summary statistic of precision and sensitivity. The formula for the F1 score is the following:

2(Precision * Sensitivity)/(Precision + Sensitivity)

NOTE

> The terms F1 score and harmonic mean are interchangeable.

To illustrate the F1 score, let's return to the scenario of a faulty algorithm for detecting fraudulent credit card transactions. Say that 100 transactions out of 100,000 are fraudulent.



In such a scenario, the sensitivity is very high, while the precision is very low. Clearly, this is not a useful algorithm. Nor does averaging the sensitivity and precision yield a useful figure. Let's try calculating the F1 score.

Using the same scenario above, what is the F1 score in this case? How did you arrive at the answer?

Check Answer

Finish ▸

The F1 score is 0.002. We noted previously that there's usually a trade-off between sensitivity and precision, and that a balance must be struck between the two. A useful way to think about the F1 score is that a pronounced imbalance between sensitivity and precision will yield a low F1 score.

17.4.2 Confusion Matrix in Practice

Great job so far! Jill tells you that metrics such as sensitivity and precision can be a bit confusing at first. She assures you, however, that with enough practice, they will become second nature. She suggests that you return to a real-world dataset to deepen your understanding.

Let's look at an example of generating a confusion matrix in Python and interpreting it. To get started, download the following file:

Download 17-4-2-precision__recall_f1.zip (https://courses.bootcampspot.com/courses/138/files/22483/download?wrap=1)

You now understand how precision, recall (sensitivity), and the F1 score can be used to assess a model's performance. Let's return to the Pima Indian diabetes dataset to go through an example in Python. Run all the cells in the notebook. All the data preparation steps have been performed, and a logistic regression model was trained and created predictions.

In the next cell, import the relevant modules for validation and print the `confusion_matrix`, which is the table of true positives, false positives, true

negatives, and false negatives.

```
from sklearn.metrics import confusion_matrix, classification_report
matrix = confusion_matrix(y_test, y_pred)
print(matrix)
```

The table printed in the notebook is unlabeled, but can be interpreted as the following:

	Predicted True	**Predicted False**
Actually True	113	12
Actually False	31	36

How many true positives and false positives are there?

Check Answer

What is the sensitivity/recall of this model?

Check Answer

What is the precision of this model?

Check Answer

Finish ▸

Although we can manually calculate the metrics of the model, Scikit-learn's `classification_report` module performs the task for us:

```
report = classification_report(y_test, y_pred)
print(report)
```

```
                   precision    recall  f1-score   support

               0       0.78      0.90      0.84       125
               1       0.75      0.54      0.63        67

        accuracy                           0.78       192
       macro avg       0.77      0.72      0.73       192
    weighted avg       0.77      0.78      0.77       192
```

The precision for prediction of the nondiabetics and diabetics are in line with each other. However, the recall (sensitivity) for predicting diabetes is much lower than it is for predicting an absence of diabetes. The lower recall for diabetics is reflected in the dropped F1 score as well.

17.5.1 Overview of Support Vector Machines

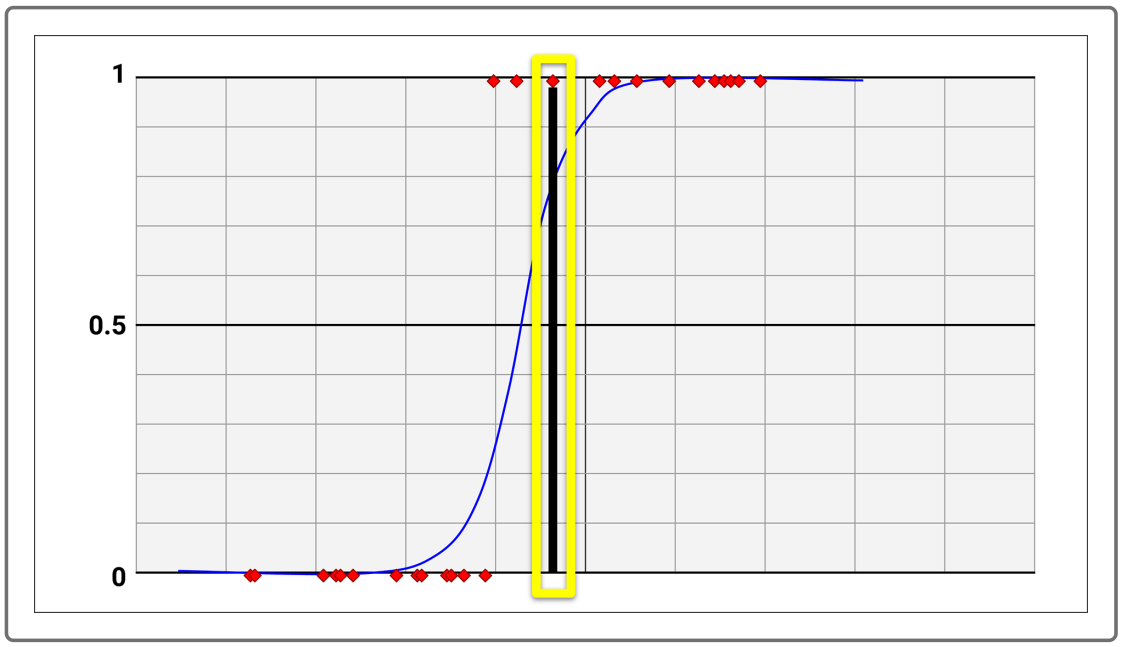
You've been working hard to understand supervised learning from all angles: theory, practice, and statistics. You're beginning to reap the rewards of your hard work as you hit your stride.

Now that you're becoming comfortable with using logistic regression and evaluating its results, Jill suggests that you learn about another powerful classification model: support vector machines. Although the name is possibly a little intimidating, you'll be able to bring much of your previous knowledge into using a support vector machine in practice.

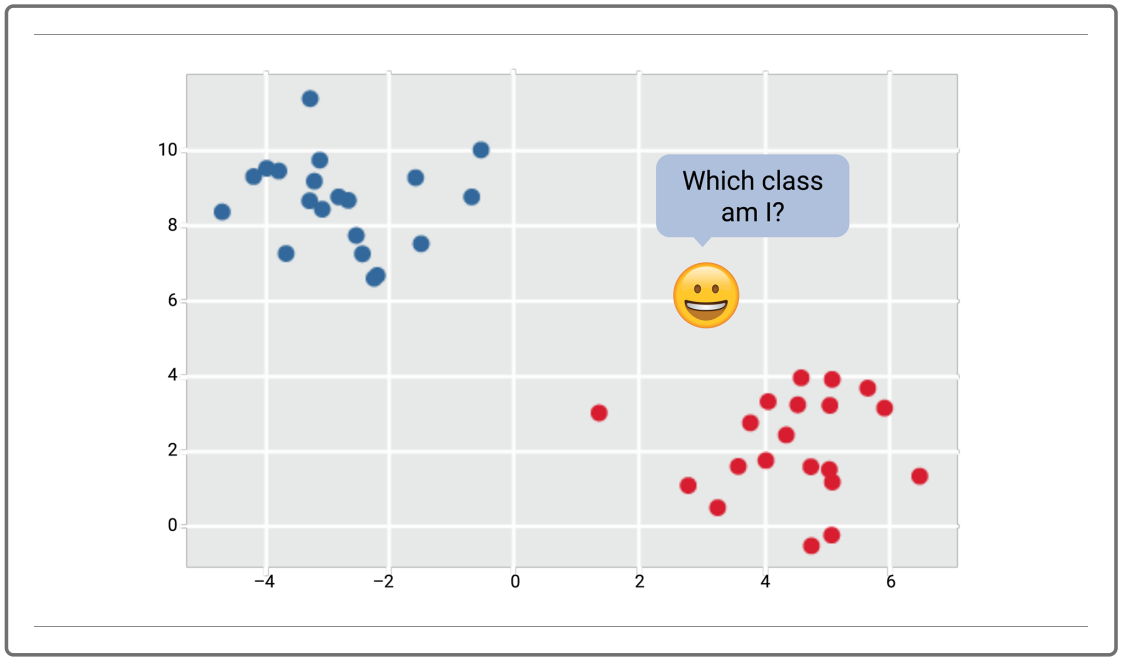
Support vector machine (SVM), like logistic regression, is a binary classifier: It can categorize samples into one of two categories (for example, yes or no).

To understand support vector machines, let's revisit logistic regression first. A logistic regression model evaluates the probability of an occurrence. For example, the model would take features into account (for example, an applicant's income and credit score) and decide whether to approve the application.

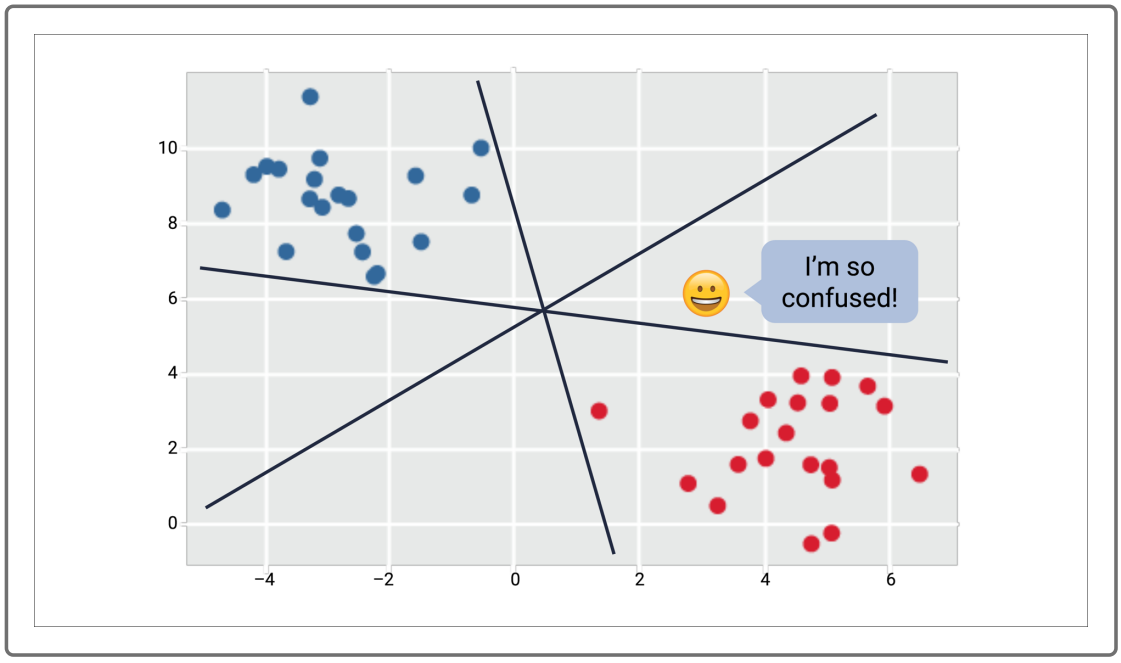
The outcome is binary because the only possible options are to approve or to deny the loan application: If the probability is higher than 0.5, the application is classified as approved, or if the probability is less than that, the application is classified as denied. There is a strict cutoff line that divides one classification from the other:



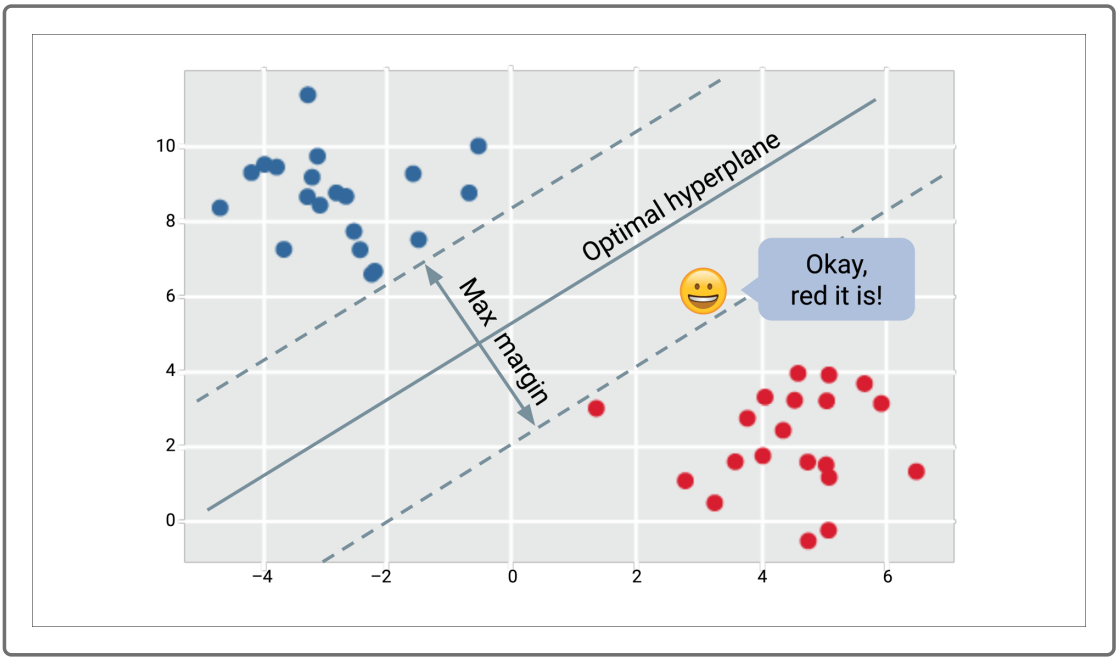
SVM also categorizes the target variable into one of two classes (for example, approved or denied). However, it differs from logistic regression in several ways. As a linear classifier, the goal of SVM is to find a line that separates the data into two classes:



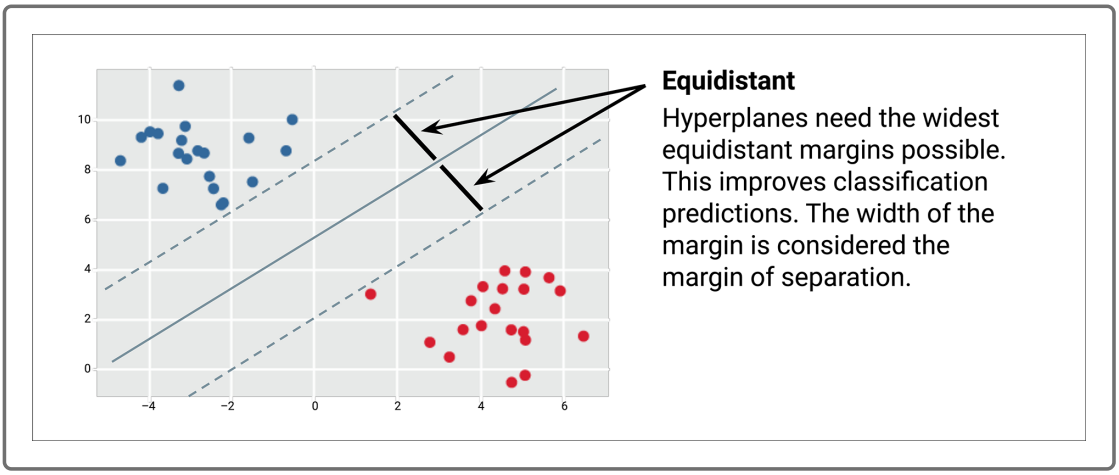
However, there may be many different ways to draw the boundary line, as shown in the diagram below. Which boundary to choose isn't always clear from visual inspection, and choosing the wrong boundary can affect the performance of the model:



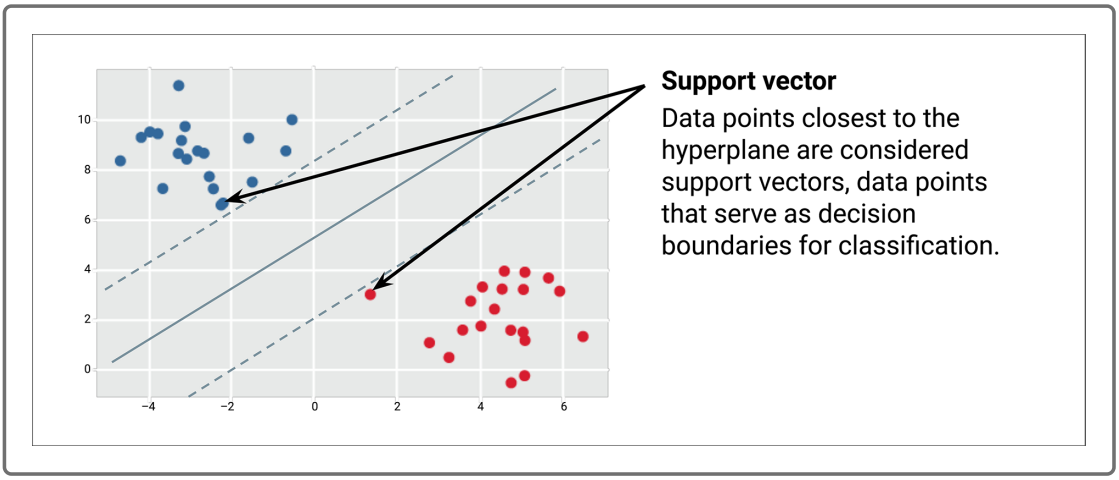
In a two-dimensional grid, as shown below, SVM draws a line at the edge of each class, and attempts to maximize the distance between them. It does so by separating the data points with the largest possible margins:



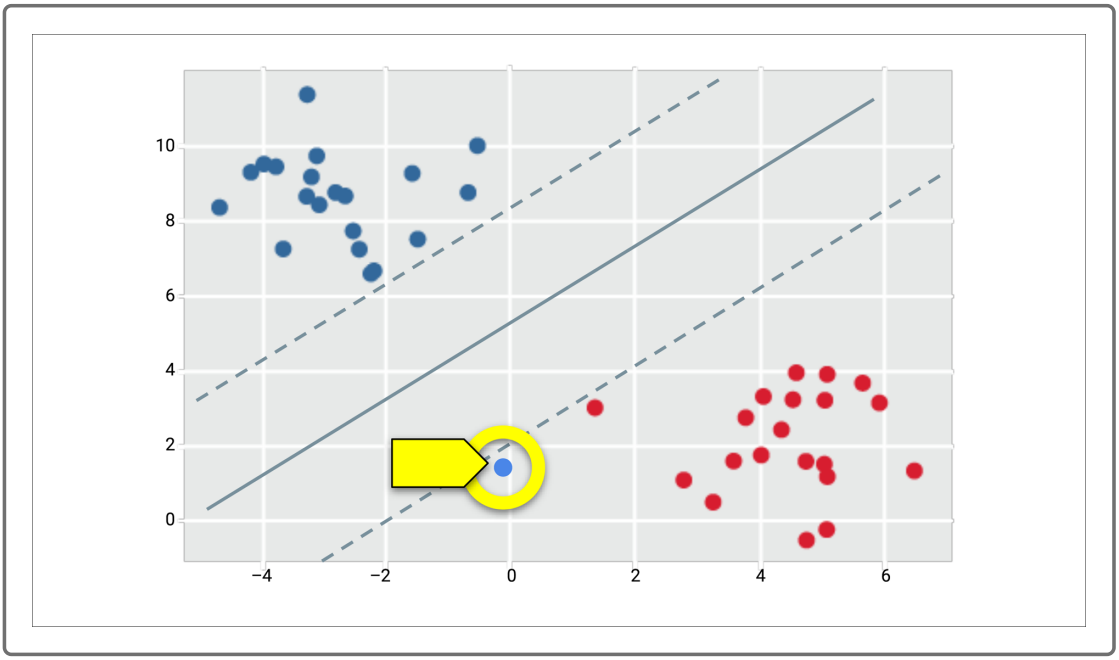
A hyperplane is the line exactly between the two margins (i.e., equidistant from both margins). Again, the SVM's goal is to find the hyperplane with the widest possible margins (i.e., the largest margin of separation between the two classes):



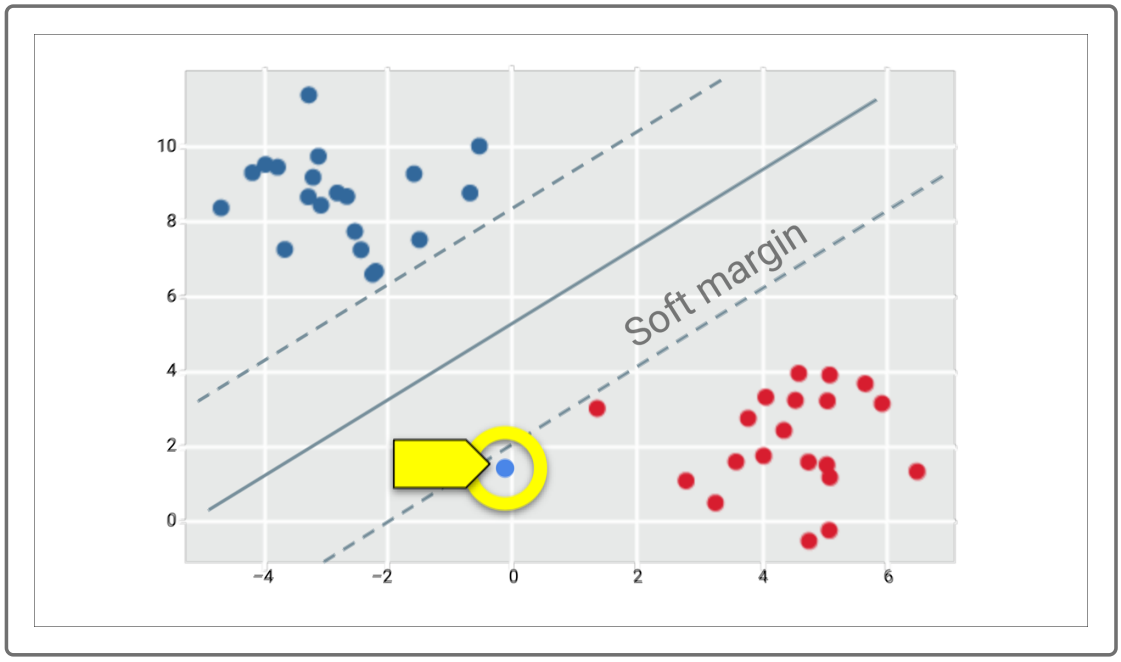
Support vectors are defined as the data points closest to the hyperplane:



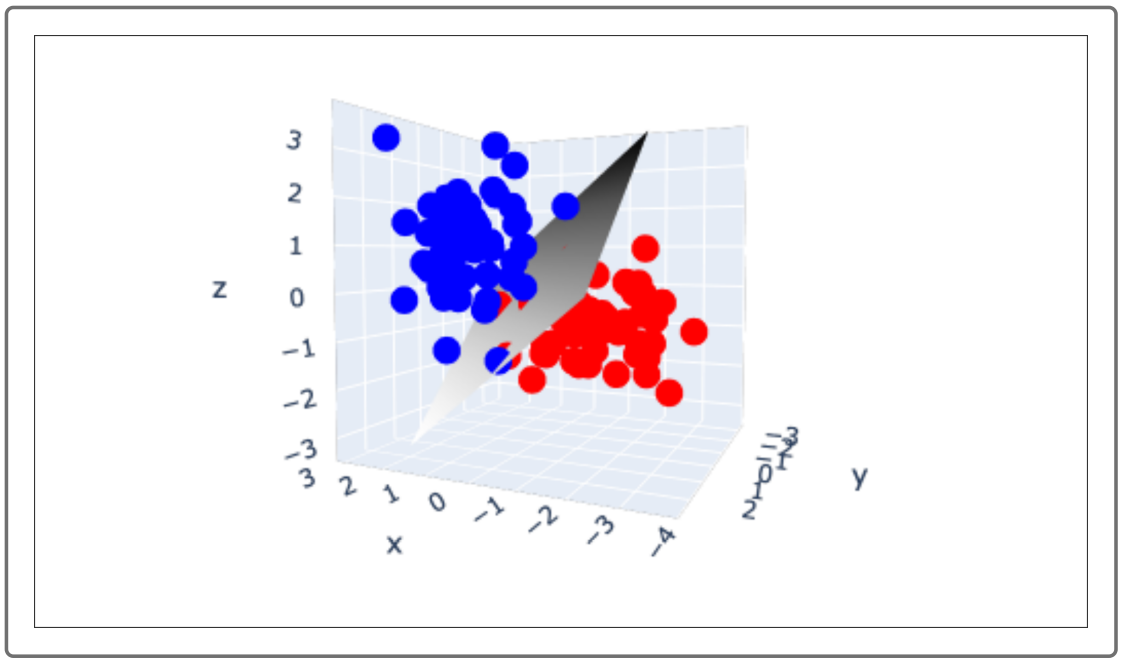
Real-life data, however, can be messy and will often not yield such a clean line of separation. Imagine that a data point belonging to the blue class were found closer to the cluster of data points that belong to the red class. In this case, would the hyperplane have to be relocated? Would the support vectors have to be redefined?



SVMs can accommodate such outliers by using soft margins. A soft margin allows SVM to make allowances for outliers that cross the hyperplane while maintaining support vectors and hyperplane to maximize the overall separation of the two classes:



Up to this point, we have visualized using SVM in datasets with two features. A dataset with three features (e.g., age, education, income) and a target with two classes (e.g., approval or denial of a loan application) would be visualized as a 3D space, with a hyperplane separating the two classes:



What are support vectors?

- A line exactly between the two margins that is placed at an equal distance from both margins.
- Data points closest to the margin of separation.
- A line that separates the data into two classes

Check Answer

Finish ▸

To summarize, SVM works by separating the two classes in a dataset with the widest possible margins. The margins, however, are soft and can make exceptions for outliers. This stands in contrast to the logistic regression model. In logistic regression, any data point whose probability of belonging to one class exceeds the cutoff point belongs to that class; all other data points belong to the other class.

17.5.2 SVM in Practice

Although the ideas behind support vector machines are different from those behind logistic regression, actually implementing a SVM model is very similar to what you have done. As before, you will split your dataset, create and train a model, create predictions, then validate the model.

Now that we have looked at how an SVM model works, let's look at using SVM in practice. To get started, download the following files.

Download 17-5-2-svm.zip (https://courses.bootcampspot.com/courses/138/files/22491/download?wrap=1)

Open the notebook and load the dataset:

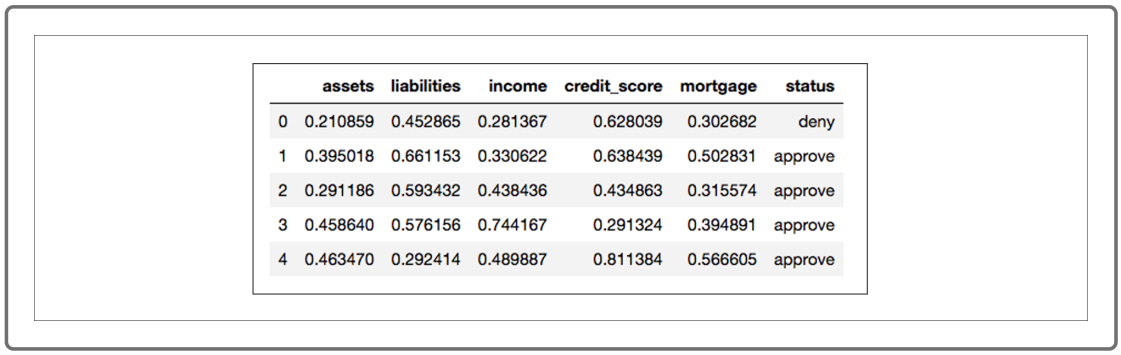
```
from path import Path
import numpy as np
import pandas as pd

data = Path('../Resources/loans.csv')
```

```
df = pd.read_csv(data)
df.head()
```

Each row in the dataset represents an application for a loan, and information is available on the applicant's assets, liabilities, income, credit score, and mortgage size. We also have information on whether the application was approved or denied. Here, the target variable is `status`, and all other columns are features used to predict the loan application status.

It's worth noting that the data in this dataset have been normalized. In this case, the data in the numerical features, such as assets and liabilities, have been scaled to be between 0 and 1.

We will discuss scaling in greater detail later, but note for now that some models require scaling the data, and that in this dataset, the scaling has been done for you:



	assets	liabilities	income	credit_score	mortgage	status
0	0.210859	0.452865	0.281367	0.628039	0.302682	deny
1	0.395018	0.661153	0.330622	0.638439	0.502831	approve
2	0.291186	0.593432	0.438436	0.434863	0.315574	approve
3	0.458640	0.576156	0.744167	0.291324	0.394891	approve
4	0.463470	0.292414	0.489887	0.811384	0.566605	approve

The next two steps should be familiar. We separate the dataset into features (X) and target (y):

```
y = df["status"]
X = df.drop(columns="status")
```

We then further split the dataset into training and testing sets. Note that the shape of the training is (75, 5), meaning 75 rows and five columns. It is

generally good practice to stratify the data when splitting into training and testing sets, especially when the dataset is small, as is the case here:

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X,
   y,  random_state=1, stratify=y)
X_train.shape
```

Next, we import the SVC module from Scikit-learn, then instantiate it. The kernel specifies the mathematical functions used to separate the classes. The kernel, in this example, identifies the orientation of the hyperplane as linear. However, a number of kernels exist that define nonlinear boundaries:

```
from sklearn.svm import SVC
model = SVC(kernel='linear')
```

We then train the model with `fit()`:

```
model.fit(X_train, y_train)
```

Next, we create predictions with the model:

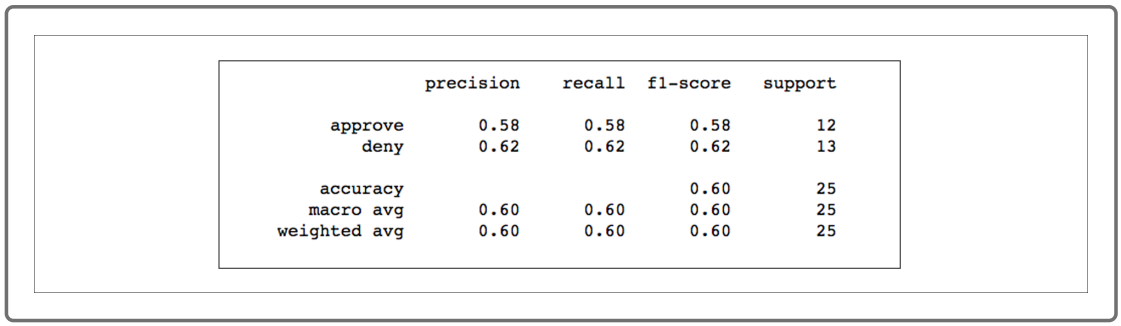
```
y_pred = model.predict(X_test)
results = pd.DataFrame({
   "Prediction": y_pred,
   "Actual": y_test
}).reset_index(drop=True)
results.head()
```

We assess the `accuracy_score` of the model, which is 0.6:

```
from sklearn.metrics import accuracy_score
accuracy_score(y_test, y_pred)
```

We then generate a `confusion_matrix` and print the classification report:

```
from sklearn.metrics import confusion_matrix
confusion_matrix(y_test, y_pred)

from sklearn.metrics import classification_report
print(classification_report(y_test, y_pred))
```



	precision	recall	f1-score	support
approve	0.58	0.58	0.58	12
deny	0.62	0.62	0.62	13
accuracy			0.60	25
macro avg	0.60	0.60	0.60	25
weighted avg	0.60	0.60	0.60	25

In summary, much of using a SVM model in practice follows the pattern we saw with logistic regression: split the dataset, create a model, train the model, create predictions, then validate the model.

After looking over the summary of the results in the classification report, answer the following questions.

A. What is the recall rate of the approve category?

B. Does the model perform adequately to pose an acceptable risk for a lender?

Check Answer

Finish ▸

SKILL DRILL

Assess the performance of a logistic regression model, namely the precision, recall, and F1 scores for the `approve` category. Compare it with the performance of the SVM model. Which model performs better?

17.6.1 Encode Labels With Pandas

It's often said that much of a data scientist's time is spent cleaning and preparing data. In machine learning, too, the data rarely comes ready for analysis.

One of the tasks involved in data preparation for machine learning is to convert textual data into numerical data.

While many datasets contain categorical features (e.g., M or F), machine learning algorithms typically only work with numerical data. Categorical and text data must therefore be converted to numerical data for use in machine learning—which is what we'll do in this section.

First, download the files you'll need for this task.

Download 17-6-1-label_encode.zip (https://courses.bootcampspot.com/courses/138/files/22494/download?wrap=1)

Download the files and open the Jupyter Notebook. We'll first import the modules we'll use and open the dataset in a Pandas DataFrame with the following code:

```
import pandas as pd
from path import Path

file_path = Path("../Resources/loans_data.csv")
loans_df = pd.read_csv(file_path)
loans_df.head()
```

A preview of the DataFrame reveals seven columns: six features and a target. The dataset contains simulated loan data. There are 500 records, and each row represents a loan application:

	amount	term	month	age	education	gender	bad
0	1000	30	June	45	High School or Below	male	0
1	1000	30	July	50	Bachelor	female	0
2	1000	30	August	33	Bachelor	female	0
3	1000	15	September	27	college	male	0
4	1000	30	October	28	college	female	0

The dataset includes the following columns:

- **Amount:** The loan amount in U.S. dollars.
- **Term:** The loan term in months.
- **Month:** The month of the year when the loan was requested.
- **Age:** Age of the loan applicant.
- **Education:** Educational level of the loan applicant.
- **Gender:** The sex of the loan applicant.
- **Bad:** Status of the application (1: bad, or denial; 0: good, or approval).

Based on the preview of this loans dataset, which column is likely to be the target?

- Amount
- Term
- Bad

Check Answer

Finish ▸

IMPORTANT

> Scikit-learn's algorithms only understand numeric data.

To use Scikit-learn's machine learning algorithms, the text features (month, education, and gender) will have to be converted into numbers. This process is called encoding. Furthermore, the steps taken to prepare the data to make them usable for building machine learning models are called preprocessing. Encoding text labels into numerical values is one preprocessing step. Later we'll discuss scaling, another preprocessing step.

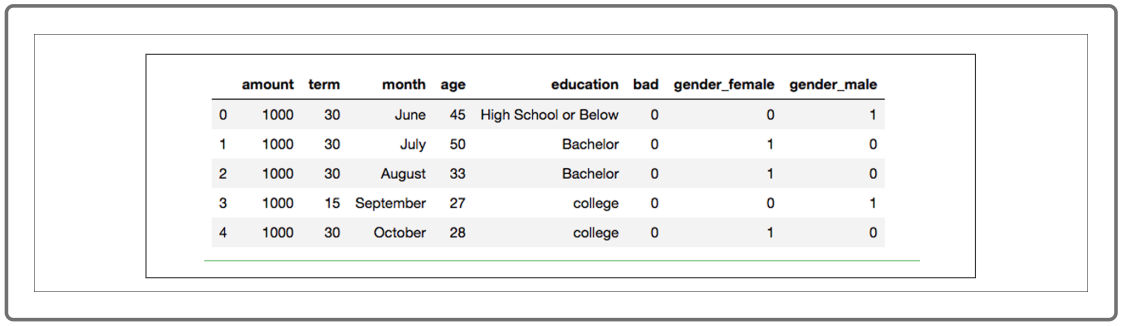
The first and the simplest encoding we'll perform in this dataset is with the `gender` column, which contains only two values: male and female. We'll convert these values into numerical ones with the `pd.get_dummies()` method:

```
loans_binary_encoded = pd.get_dummies(loans_df, columns=["gender"])
loans_binary_encoded.head()
```

The method takes two arguments:

- The first argument for `pd.get_dummies()` here is the DataFrame.

- The second argument specifies the column to be encoded.

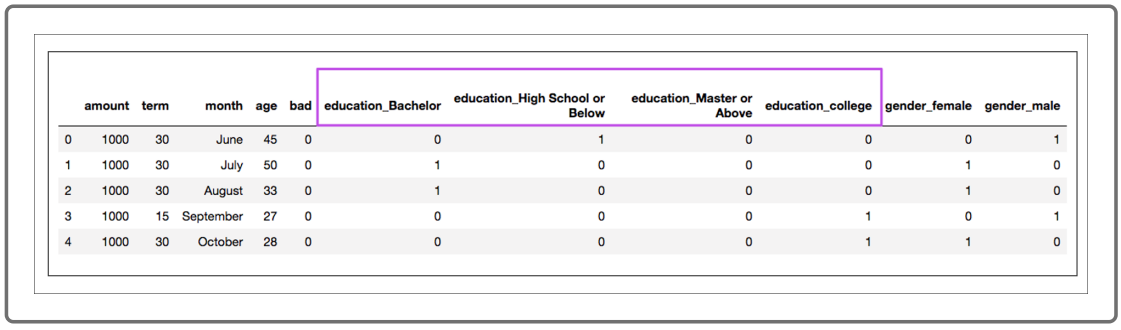


	amount	term	month	age	education	bad	gender_female	gender_male
0	1000	30	June	45	High School or Below	0	0	1
1	1000	30	July	50	Bachelor	0	1	0
2	1000	30	August	33	Bachelor	0	1	0
3	1000	15	September	27	college	0	0	1
4	1000	30	October	28	college	0	1	0

The gender column has split into two columns, `gender_female` and `gender_male`, with each column now containing 0 (false) or 1 (true). Since the first row represents a male loan applicant, the `gender_female` column reads 0 and the `gender_male` column reads 1.

It's also possible to encode multiple columns at the same time.

```
loans_binary_encoded = pd.get_dummies(loans_df, columns=["education", "gende
loans_binary_encoded.head()
```



	amount	term	month	age	bad	education_Bachelor	education_High School or Below	education_Master or Above	education_college	gender_female	gender_male
0	1000	30	June	45	0	0	1	0	0	0	1
1	1000	30	July	50	0	1	0	0	0	1	0
2	1000	30	August	33	0	1	0	0	0	1	0
3	1000	15	September	27	0	0	0	0	1	0	1
4	1000	30	October	28	0	0	0	0	1	1	0

As before, the `gender` column has split into two columns. The education column has split into four columns (`Bachelor, High School or Below`, `Master or Above`, and `college`), with an associated 0 or 1. If a loan applicant has a bachelor's degree, that column will read 1, and the others (`High School or Below`, `Master or Above`, and `college`) will read 0. For an applicant who did not graduate from high school, the `education_Bachelor`, `education_Master or`

`Above`, and `education_college` columns will be 0, and the `education_High School or Below` will show 1.

17.6.2 Encode Labels With Scikit-learn

Pandas, as you have seen, offers tools to encode your data. Scikit-learn offers another way to encode your labels.

Scikit-learn's `LabelEncoder` module can also transform text into numerical data. Let's look at an example. Continue down the notebook from the preceding section:

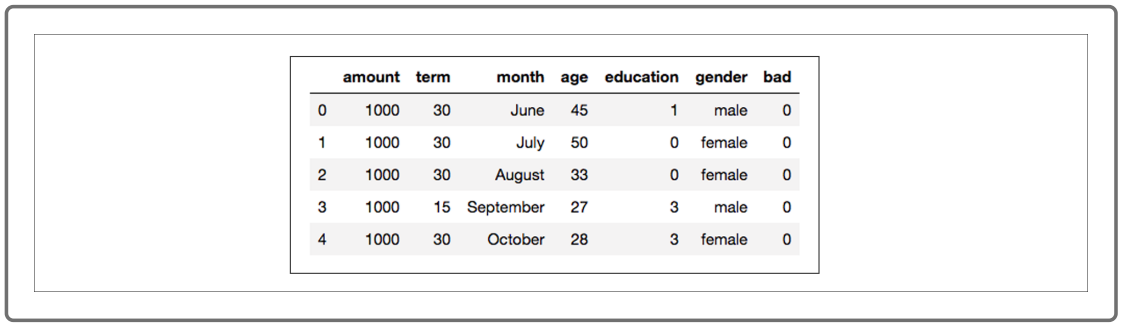
```
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
df2 = loans_df.copy()
df2['education'] = le.fit_transform(df2['education'])
```

The code includes the following elements:

- After importing the module, an instance of the label encoder object is created and assigned the variable `le`.
- A copy of the original `loans_df` is created for this example, but this step is not necessary for using label encoder.

- The label encoder's `fit_transform()` method is used to first train the label encoder, then convert the text data into numerical data.

The result is a numerical encoding of the `education` column. In contrast to `pd.get_dummies()`, the label encoder assigns a number between 0 and 3 for each of the education categories. The applicant in the first row, for example, has the value 1, which represents high school or below:



	amount	term	month	age	education	gender	bad
0	1000	30	June	45	1	male	0
1	1000	30	July	50	0	female	0
2	1000	30	August	33	0	female	0
3	1000	15	September	27	3	male	0
4	1000	30	October	28	3	female	0

SKILL DRILL

Use Scikit-learn's `LabelEncoder` to encode the `gender` column.

17.6.3 Create Custom Encoding

Jill explains that not every machine learning task can be performed by out-of-the-box solutions, meaning libraries written by other programmers. Sometimes you have to roll up your sleeves and write your own custom code!

It's also possible to create custom encoding functions. To understand why this might be useful, let's first look at using the `LabelEncoder` module. With it, you'll transform the `month` column into numbers. The goal is to transform each month into its corresponding order: for example, January should be transformed to 1, since it's the first month of the year. Similarly, July should be transformed to 7, since it's the seventh month of the year:

```
label_encoder = LabelEncoder()
loans_df["month_le"] = label_encoder.fit_transform(loans_df["month"])
loans_df.head()
```

	amount	term	month	age	education	gender	bad	month_le
0	1000	30	June	45	High School or Below	male	0	6
1	1000	30	July	50	Bachelor	female	0	5
2	1000	30	August	33	Bachelor	female	0	1
3	1000	15	September	27	college	male	0	11
4	1000	30	October	28	college	female	0	10

Note that a new instance of `LabelEncoder` was created here as `label_encoder`. The month of August, for example, is converted to 1 instead of 8. July is converted to 5 instead of 7.

Instead, we can create a dictionary of the months of the year and apply a custom function to convert the month names to their corresponding integers:

```
months_num = {
   "January": 1,
   "February": 2,
   "March": 3,
   "April": 4,
   "May": 5,
   "June": 6,
   "July": 7,
   "August": 8,
   "September": 9,
   "October": 10,
   "November": 11,
   "December": 12,
}
```

In the next cell, a `lambda` function is applied to the `month` column to perform the actual conversion:

```
loans_df["month_num"] = loans_df["month"].apply(lambda x: months_num[x])
```

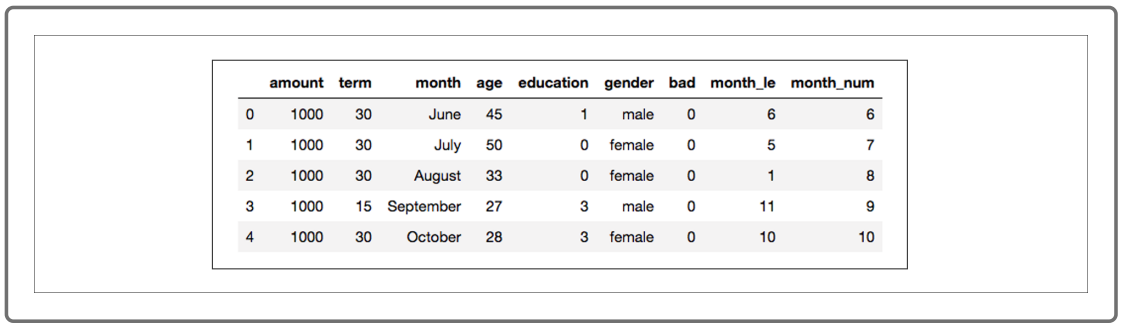
The following actions are taking place:

- A transformation is made to the values of the `month` column, and the transformed values are placed in the `month_num` column.
- The `apply()` method runs the function inside its parentheses on each element of the `month` column.
- The `lambda` function takes an argument (`x`), and returns `months_num[x]`. For example, if the value in the `month` column is "June," the function returns `months_num["June"]`, which is 6.

REWIND

`Lambda` functions are anonymous Python functions.

The DataFrame's `month_num` column now displays each month as a number:



	amount	term	month	age	education	gender	bad	month_le	month_num
0	1000	30	June	45	1	male	0	6	6
1	1000	30	July	50	0	female	0	5	7
2	1000	30	August	33	0	female	0	1	8
3	1000	15	September	27	3	male	0	11	9
4	1000	30	October	28	3	female	0	10	10

The code in the next cell is merely cleanup—it drops the unnecessary columns related to the month:

```
loans_df = loans_df.drop(["month", "month_le"], axis=1)
loans_df.head()
```

	amount	term	age	education	gender	bad	month_num
0	1000	30	45	1	male	0	6
1	1000	30	50	0	female	0	7
2	1000	30	33	0	female	0	8
3	1000	15	27	3	male	0	9
4	1000	30	28	3	female	0	10

SKILL DRILL

Create a new Jupyter Notebook and open `loans_data.csv` as a Pandas DataFrame. Encode the following labels of the dataset: `month`, `education`, and `gender`. Then save your DataFrame as `loans_data_encoded.csv`.

Your DataFrame should look like this:

	amount	term	age	bad	month_num	education_Bachelor	education_High School or Below	education_Master or Above	education_college	gender_female	gender_male
0	1000	30	45	0	6	0	1	0	0	0	1
1	1000	30	50	0	7	1	0	0	0	1	0
2	1000	30	33	0	8	1	0	0	0	1	0
3	1000	15	27	0	9	0	0	0	1	0	1
4	1000	30	28	0	10	0	0	0	1	1	0

17.6.4 Scale and Normalize Data

Data scaling and normalization are steps that are sometimes necessary when preparing data for machine learning. Jill explains that when features have different scales, features using larger numbers can have a disproportionate impact on the model. It is therefore important to understand when to scale and normalize data.

Earlier, we worked with a dataset that had already been scaled for us: the values in each column were rescaled to be between 0 and 1. Such scaling is often necessary with models that are sensitive to large numerical values. This is normally the case with models that measure distances between data points. SVM is one model that usually benefits from scaling.

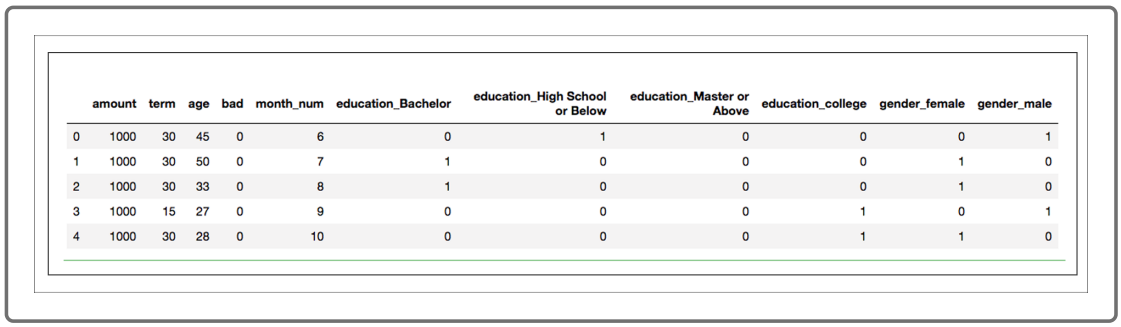
In this section, we'll use Scikit-learn's `StandardScaler` module to scale data. The model -> fit -> predict/transform workflow is also used when scaling data. The standard scaler standardizes the data. This means that each feature will be rescaled so that its mean is 0 and its standard deviation is 1.

Get started by downloading the files you'll need.

[Download 17-6-4-scale.zip (https://courses.bootcampspot.com/courses/138/files/22500/download?wrap=1)](https://courses.bootcampspot.com/courses/138/files/22500/download?wrap=1)

Open Jupyter Notebook and read in your saved `loans_data_encoded.csv` file that you created in the previous Skill Drill. If you had trouble creating the encoded data, the dataset is included in the files you downloaded:

```
import pandas as pd
from path import Path

file_path = Path("../Resources/loans_data_encoded.csv")
encoded_df = pd.read_csv(file_path)
encoded_df.head()
```



	amount	term	age	bad	month_num	education_Bachelor	education_High School or Below	education_Master or Above	education_college	gender_female	gender_male
0	1000	30	45	0	6	0	1	0	0	0	1
1	1000	30	50	0	7	1	0	0	0	1	0
2	1000	30	33	0	8	1	0	0	0	1	0
3	1000	15	27	0	9	0	0	0	1	0	1
4	1000	30	28	0	10	0	0	0	1	1	0

To scale the data in this DataFrame, we'll first import the `StandardScaler` module and create an instance of it as `data_scaler`:

```
from sklearn.preprocessing import StandardScaler
data_scaler = StandardScaler()
```

The next step is to train the scaler and transform the data. Notice that the fit and transform steps are combined into a single `fit_transform()` method. However, they can also be used sequentially with `data_scaler.fit()`, then `data_scaler.transform()`:

```
loans_data_scaled = data_scaler.fit_transform(encoded_df)
```

After transforming the data, we can preview the scaled data:

```
loans_data_scaled[:5]
```

```
array([[ 0.49337687,  0.89789115,  2.28404253, -0.81649658, -0.16890147,
        -0.39336295,  1.17997648, -0.08980265, -0.88640526, -0.42665337,
         0.42665337],
       [ 0.49337687,  0.89789115,  3.10658738, -0.81649658,  0.12951102,
         2.54218146, -0.84747452, -0.08980265, -0.88640526,  2.34382305,
        -2.34382305],
       [ 0.49337687,  0.89789115,  0.3099349 , -0.81649658,  0.42792352,
         2.54218146, -0.84747452, -0.08980265, -0.88640526,  2.34382305,
        -2.34382305],
       [ 0.49337687, -0.97897162, -0.67711892, -0.81649658,  0.72633602,
        -0.39336295, -0.84747452, -0.08980265,  1.12815215, -0.42665337,
         0.42665337],
       [ 0.49337687,  0.89789115, -0.51260995, -0.81649658,  1.02474851,
        -0.39336295, -0.84747452, -0.08980265,  1.12815215,  2.34382305,
        -2.34382305]])
```

After standardization, what are the mean and the standard deviation of the first column?

- Mean: 0, standard deviation: 0
- Mean: 0, standard deviation: 1
- Mean: 1, standard deviation: 0
- Mean: 1, standard deviation: 1

Check Answer

Finish ▸

After transformation, the data in each column should be standardized. Let's verify that the mean of each column is 0 and its standard deviation is 1:

```
import numpy as np
print(np.mean(loans_data_scaled[:,0]))
print(np.std(loans_data_scaled[:,0]))
```

The following actions are taking place:

- The `np.mean()` and `np.std()` methods return the mean and standard deviation of an array.
- `loans_data_scaled[:,0]` returns all rows and the first column of the dataset.
- The mean of the first column is evaluated as -3.552713678800501e-16, an infinitesimally small number that approximates 0.
- The standard deviation is evaluated as 0.9999999999999999, which is very close to 1.

So the standardization was indeed successful, and all numerical columns should now have a mean of 0 and a standard deviation of 1, reducing the likelihood that large values will unduly influence the model.

SKILL DRILL

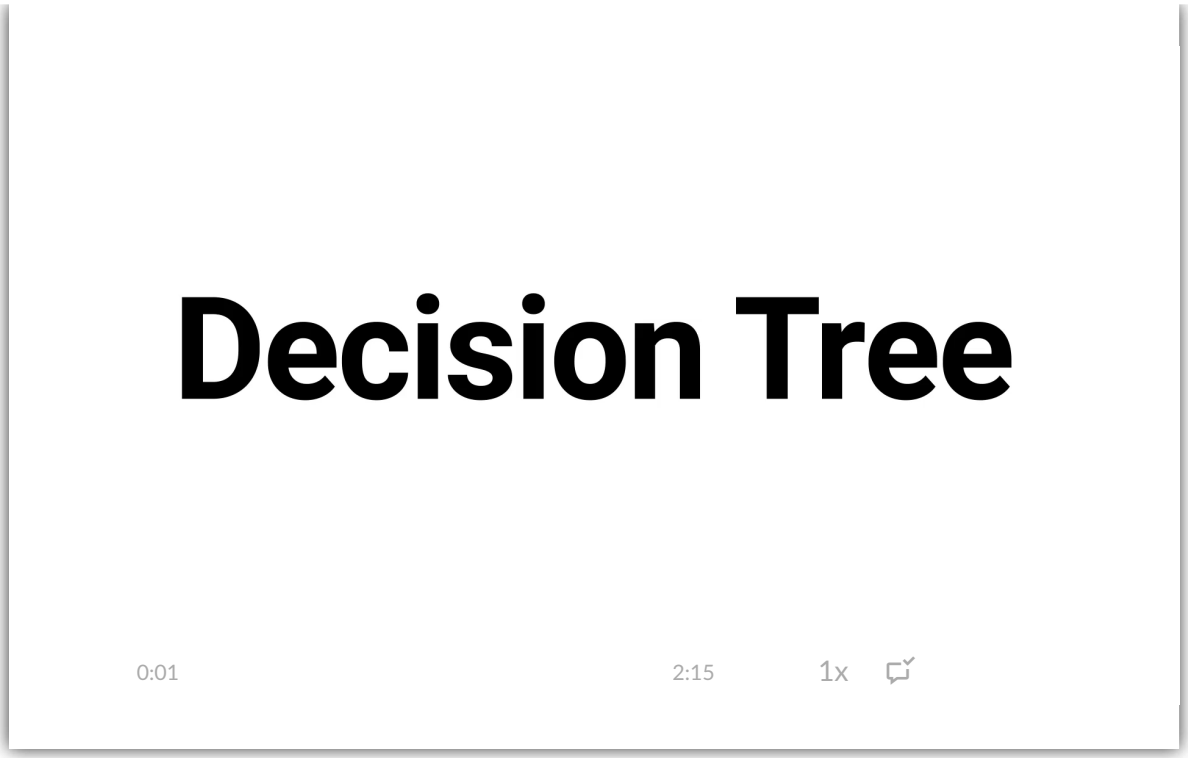
`loans_data_scaled.shape` returns (500, 11), indicating that there are 500 rows and 11 columns. Create a `for` loop to verify that all columns of the dataset have been standardized.

17.7.1 Overview of Decision Trees

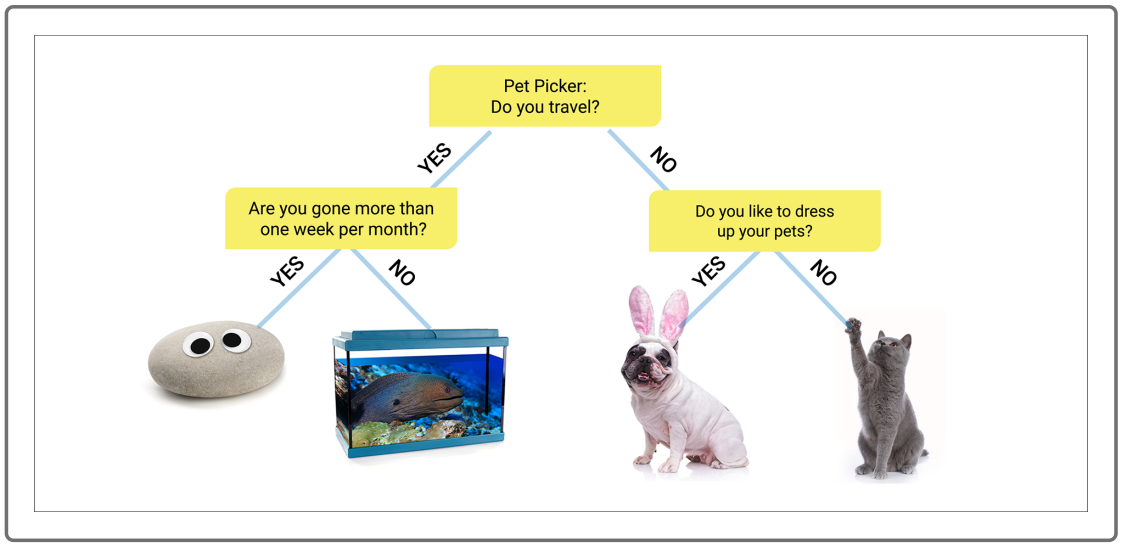
You're becoming more and more comfortable with supervised learning, and Jill suggests that you look into decision trees. The basic idea behind decision trees is simple. Furthermore, decision trees can be combined into even more powerful classifiers. In fact, decision trees are the basis of many models that win machine learning competitions.

As you may have guessed, decision trees are used in decision analysis, like determining if a coin flip will be heads or tails, or whether a loan application is approved. In short, decision trees encode a series of true/false questions that are represented by a series of if/else statements. Decision trees are one of the most interpretable models, as they provide a clear representation of how the model works.

Decision trees are natural ways in which you can classify or label objects by asking a series of questions designed to zero in on the true answer. However, decision trees can become very complex and very deep, depending on how many questions have to be answered. Deep and complex trees tend to overfit to the data and do not generalize well.



The following example, a pet picker decision tree, covers some key concepts:

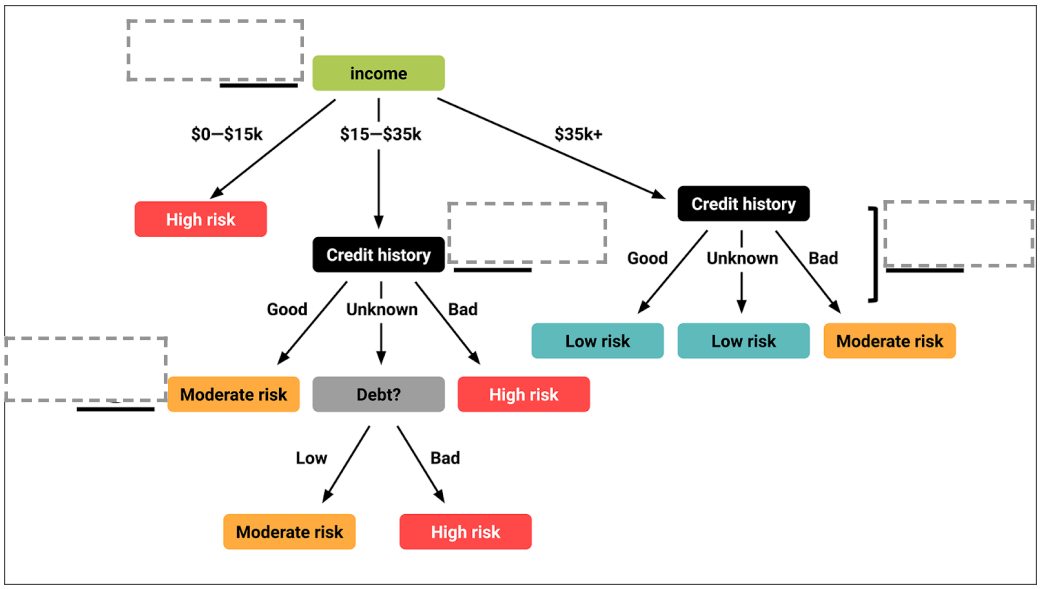


At the top of this decision tree is the **root node** or the **parent node**: “Pet Picker: Do you travel?” The root node represents the entire population. This node gets divided into two or more homogeneous sets in our decision tree to answer the question, “Do you travel?” Each answer, “Yes” or “No,” is a **branch** or subsection of the tree. When we divide a node into two or more subnodes, it’s called **splitting**.

When we split the root node into two subnodes, they are called **child nodes.** Our child nodes include two questions: “Are you gone more than one week per month?” and “Do you like to dress up your pets?” These child nodes are split again into subnodes—the images in the decision tree—making each child node a **decision node.** The four images do not split further and are referred to as a **leaf** or **terminal node**.

Our pet picker decision **tree depth** is not that deep because, at two, it has a low number of decision nodes one encounters before making a decision: “Are you gone more than one week per month?” and “Do you like to dress up your pets?”

Label the parts of the following credit approval decision tree using the following terms: child node, leaf node, root node, and splitting.



child node | leaf node | splitting | root node

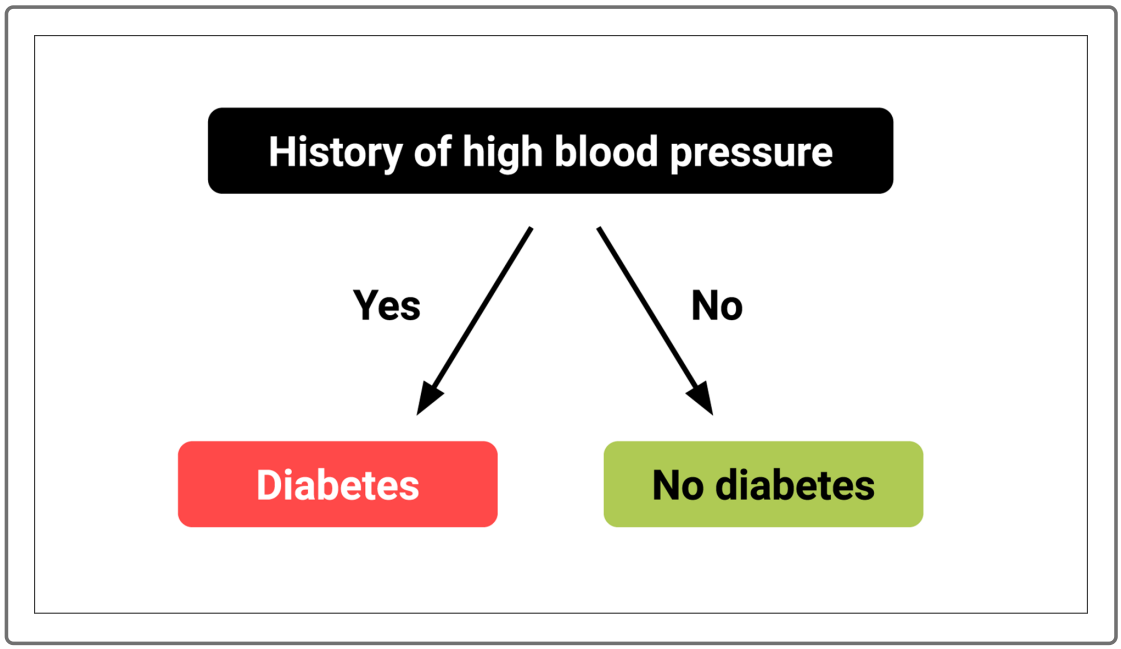
Check Answer

Finish ▸

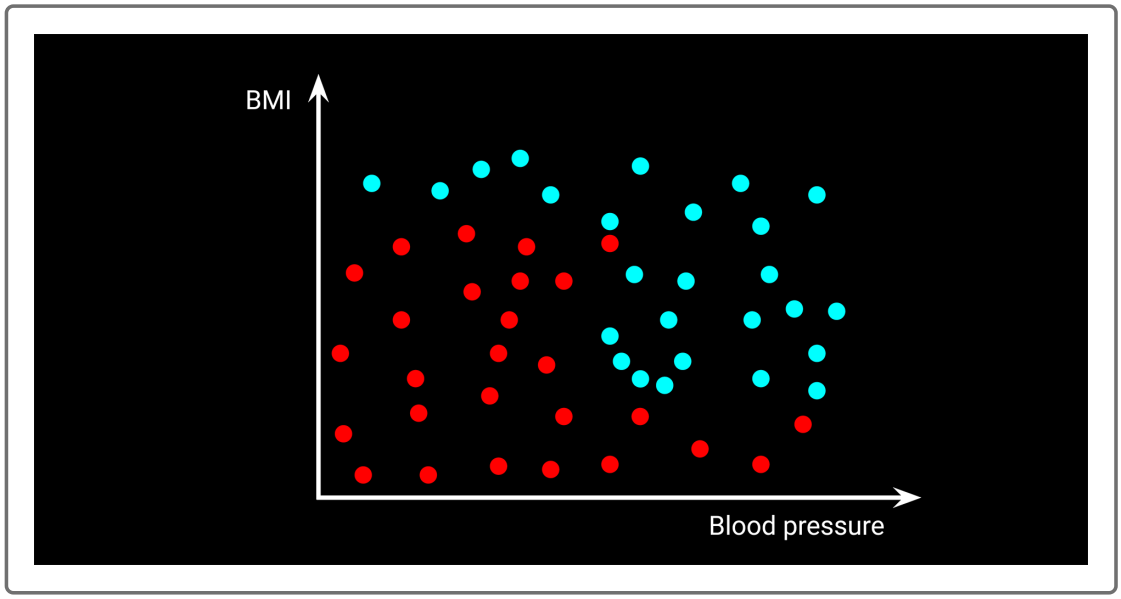
Decision trees, as you have seen, are like a series of if-else statements. But how does a decision tree determine which if-else statement to use first? In other words, how does it decide on the root node?

To understand this process, let's imagine a scenario in which we use multiple factors, such as body mass index (BMI) over 30, and a history of high blood pressure, to predict whether a patient has diabetes. There are two candidates for the root node in this example: BMI and blood pressure. The classification that creates the best split becomes the root node.

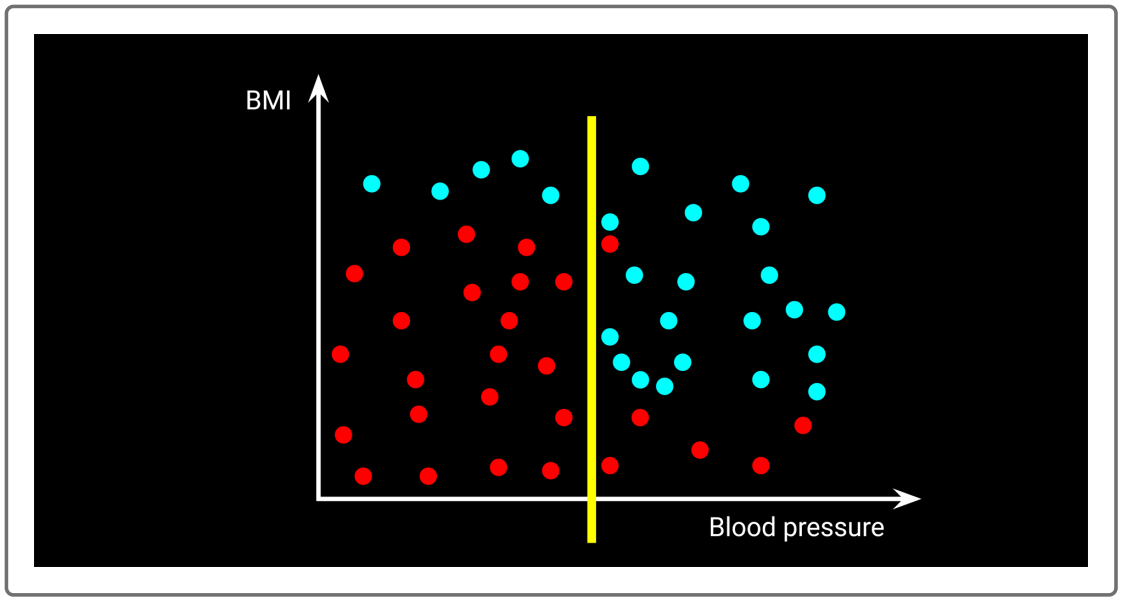
For example, let's say that a history of hypertension best predicts whether a person will be diabetic. In that case, it becomes the root node:



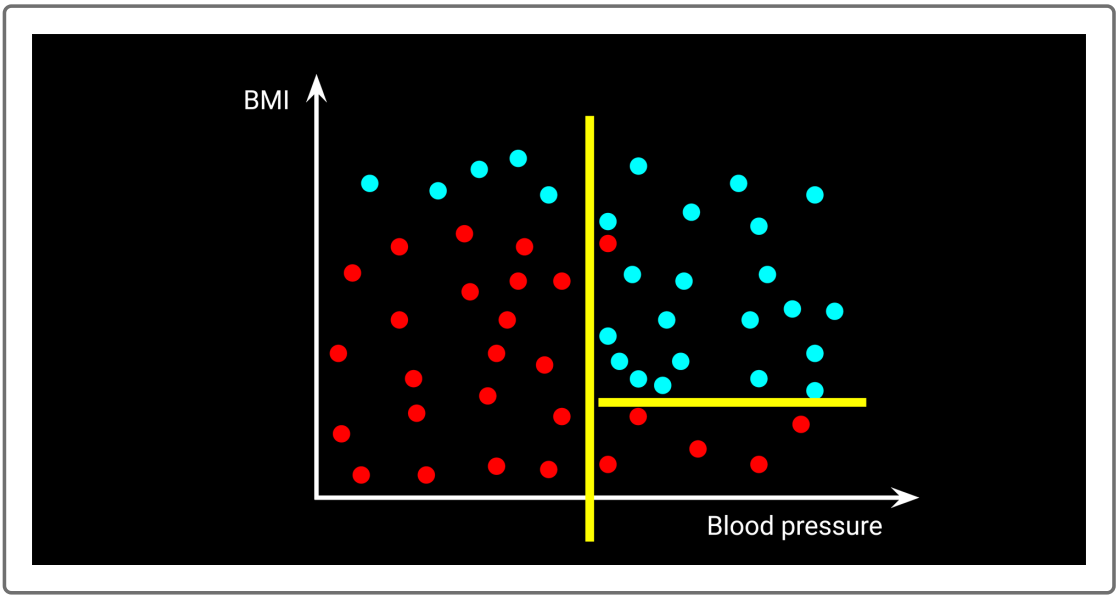
The same idea holds when the variables are continuous rather than discrete. Let's say that we're dealing with the same two variables, BMI and blood pressure, to predict diabetes. The decision tree looks for the best value along each axis to split the diabetics from non-diabetics:



Looking at this image, we can see that a vertical line along the x-axis (blood pressure) would best split the dataset into diabetics and non-diabetics, resulting in the fewest errors. In other words, a specific blood pressure value can be used to split the dataset into diabetic and non diabetic groups:



The same process is repeated to create subsequent child nodes. In this example, a horizontal line that represents a particular BMI will become the next node:



17.7.2 Predict Loan Application Approval

Now that you have learned how decision trees work, it's time to look at using a decision tree model in practice. You will first perform the data preprocessing steps.

Let's try to predict loan application approvals using a decision tree on the `loans_data_encoded.csv` data we previously used to encode the dataset. Feel free to begin a new notebook, or to follow along the provided notebook.

Download 17-7-2-decision_tree.zip (https://courses.bootcampspot.com/courses/138/files/22506/download?wrap=1)

In Jupyter Notebook, import the following dependencies:

```
# Initial imports
import pandas as pd
from path import Path
from sklearn import tree
from sklearn.preprocessing import StandardScaler
```

```
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, accuracy_score, classification
```

Next, read in your saved `loans_data_encoded.csv` file.

```
# Loading data
file_path = Path("../Resources/loans_data_encoded.csv")
df_loans = pd.read_csv(file_path)
df_loans.head()
```

	amount	term	age	bad	month_num	education_Bachelor	education_High School or Below	education_Master or Above	education_college	gender_female	gender_male
0	1000	30	45	0	6	0	1	0	0	0	1
1	1000	30	50	0	7	1	0	0	0	1	0
2	1000	30	33	0	8	1	0	0	0	1	0
3	1000	15	27	0	9	0	0	0	1	0	1
4	1000	30	28	0	10	0	0	0	1	1	0

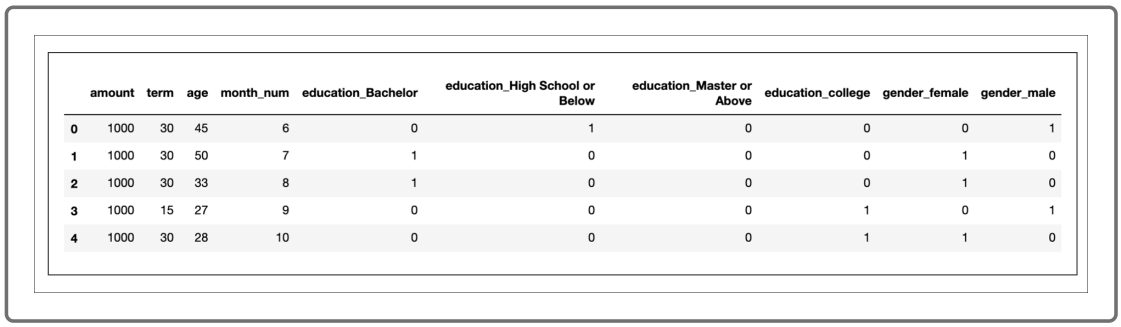
Our goal is to predict if a loan application is worthy of approval based on information we have in our `df_loans` DataFrame. To do this, we'll have to split our dataset into features (or inputs) and target (or outputs). The features set, `X`, will be a copy of the `df_loans` DataFrame without the `bad` column. These features are all the variables that help determine whether a loan application should be denied.

REWIND

Recall that X is the input data and y is the output data.

```
# Define the features set.
X = df_loans.copy()
X = X.drop("bad", axis=1)
X.head()
```

The output from the following code block will give us the following features set.



	amount	term	age	month_num	education_Bachelor	education_High School or Below	education_Master or Above	education_college	gender_female	gender_male
0	1000	30	45	6	0	1	0	0	0	1
1	1000	30	50	7	1	0	0	0	1	0
2	1000	30	33	8	1	0	0	0	1	0
3	1000	15	27	9	0	0	0	1	0	1
4	1000	30	28	10	0	0	0	1	1	0

The target set is the `bad` column, indicating whether or not a loan application is good (0) or bad (1). Run the following code to generate the target set data.

```
# Define the target set.
y = df_loans["bad"].values
y[:5]
```

A preview of the target set indicates five good (loan worthy) applications.

```
array([0, 0, 0, 0, 0])
```

Split the Data Into Training and Testing Sets

To train and validate our model, we'll need to split the features and target sets into training and testing sets. This will help determine the relationships between each feature in the features training set and the target training set, which we'll use to determine the validity of our model using the features and target testing sets.

In Jupyter Notebook, add the following code that will split our data into training and testing sets.

```
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=78)
```

When the `train_test_split()` function is executed, our data is split into a specific proportion of the original data sets. By default, our training and testing data sets are 75% and 25%, respectively, of the original data. Using the following code, we can see the data's 75-25 split.

```
# Determine the shape of our training and testing sets.
print(X_train.shape)
print(X_test.shape)
print(y_train.shape)
print(y_test.shape)
```

The output from running the code above shows that the `X_train` and `y_train` is 75% of 500 and that the `X_test` and `y_test` are 25%.

```
(375, 10)
(125, 10)
(375, 1)
(125, 1)
```

We can manually specify the desired split with the `train_size` parameter.

```
# Splitting into Train and Test sets into an 80/20 split.
X_train2, X_test2, y_train2, y_test2 = train_test_split(X, y, random_state=7
```

To see that the shape of our training and testing sets is a 80-20 split, we run the following code.

```
# Determine the shape of our training and testing sets.
print(X_train2.shape)
print(X_test2.shape)
print(y_train2.shape)
print(y_test2.shape)
```

The output from this code will give the following results.

```
(400, 10)
(100, 10)
(400, 1)
(100, 1)
```

NOTE

> Consult the sklearn documentation for additional information about the `train_test_split()` (https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html) function and the parameters it takes.

Scale the Training and Testing Data

Now that we have split our data into training and testing sets, we can scale the data using Scikit-learn's `StandardScaler`.

REWIND

The standard scaler standardizes the data. Which means that each feature will be rescaled so that its mean is 0 and its standard deviation is 1.

NOTE

> Typically, models that compute distances between data points, such as SVM, require scaled data. Although decision trees don't require scaling the data, it can be helpful when comparing the performances of different models.

To scale our data, we'll use the `StandardScaler` as before and fit the instance, `scaler`, with the training data and then scale the features with the `transform()` method:

```
# Creating a StandardScaler instance.
scaler = StandardScaler()
# Fitting the Standard Scaler with the training data.
X_scaler = scaler.fit(X_train)

# Scaling the data.
X_train_scaled = X_scaler.transform(X_train)
X_test_scaled = X_scaler.transform(X_test)
```

After transformation, complete the following code block to determine the actual mean and the standard deviation of the first column of the scaled data.

```
import [          ] as np
np.mean(X_train_scaled[          ])
np.mean([          ])
np.[          ](X_train_scaled[          ])
np.[          ]([          ])
```

Check Answer

Finish ▸

17.7.3 Make Predictions and Evaluate Results

Now that you have preprocessed your dataset, you can now turn your attention to using the decision tree model to make predictions and evaluate the results.

Fit the Decision Tree Model

After scaling the features data, the decision tree model can be created and trained. First, we create the decision tree classifier instance and then we train or fit the "model" with the scaled training data.

Add and run the following code block to create the decision tree instance and fit the model:

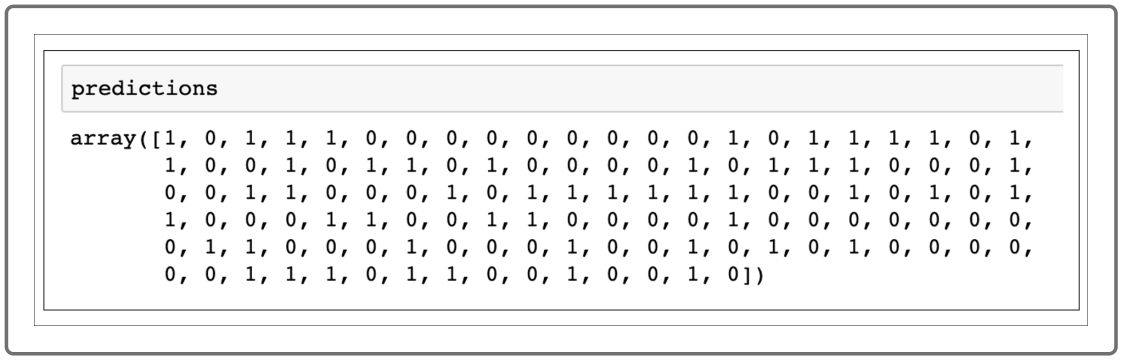
```
# Creating the decision tree classifier instance.
model = tree.DecisionTreeClassifier()
# Fitting the model.
model = model.fit(X_train_scaled, y_train)
```

Make Predictions Using the Testing Data

After fitting the model, we can run the following code to make predictions using the scaled testing data:

```
# Making predictions using the testing data.
predictions = model.predict(X_test_scaled)
```

The output from this code will be an array of 125 predictions with either a 1 for a bad loan application or a 0 for a good, or approved, loan application.



```
predictions
```

```
array([1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1, 1, 0, 1,
       1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 0, 0, 1,
       0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 1, 0, 1,
       1, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0,
       0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0,
       0, 0, 1, 1, 1, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0])
```

NOTE

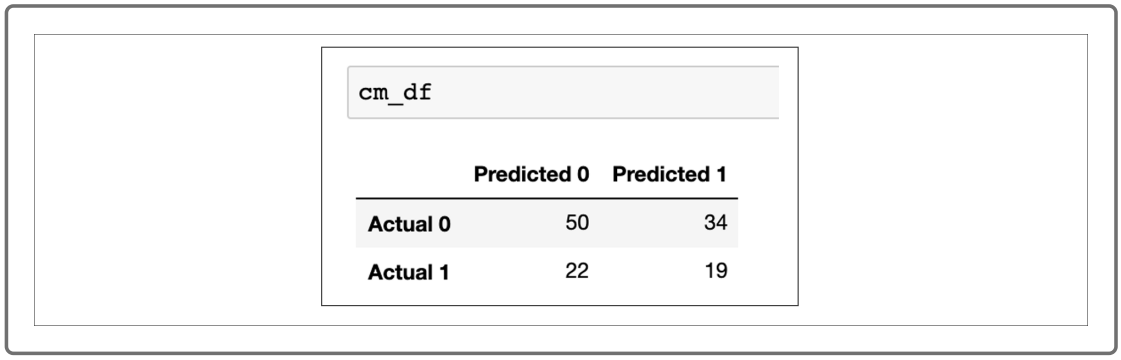
> Your predictions array may look different if you don't use the same seeding in the `random_state` parameter.

Evaluate the Model

Finally, we'll determine how well our model classifies loan applications. First, we need to use a confusion matrix.

The following code block creates the `confusion_matrix` using the `y_test` and the `predictions` that we just calculated and adds the `confusion_matrix` array to a DataFrame:

```
# Calculating the confusion matrix
cm = confusion_matrix(y_test, predictions)

# Create a DataFrame from the confusion matrix.
cm_df = pd.DataFrame(
    cm, index=["Actual 0", "Actual 1"], columns=["Predicted 0", "Predicted 1

cm_df
```



The results show that:

- Out of 84 good loan applications (Actual 0), 50 were predicted to be good (Predicted 0), which we call true positives.
- Out of 84 good loan applications (Actual 0), 34 were predicted to be bad (Predicted 1), which are considered false negatives.
- Out of 41 bad loan applications (Actual 1), 22 were predicted to be good (Predicted 0) and are considered false positives.
- Out of 41 bad loan applications (Actual 1), 19 were predicted to be bad (Predicted 1) and are considered true negatives.

What is the recall (sensitivity) for bad loans (Actual 1)?

- 0.13
- 0.25
- 0.46
- 0.78

Check Answer

Finish ▸

We can add these terms to the confusion matrix and add the row and column totals to get the following table:

n=125	**Predicted Good**	**Predicted Bad**	
Actual Good	TP = 50	FN = 34	**84**
Actual Bad	FP = 22	TN = 19	**41**
	72	**53**	

Next, we can determine the accuracy, or how often the classifier is correct with the model, by running the following code:

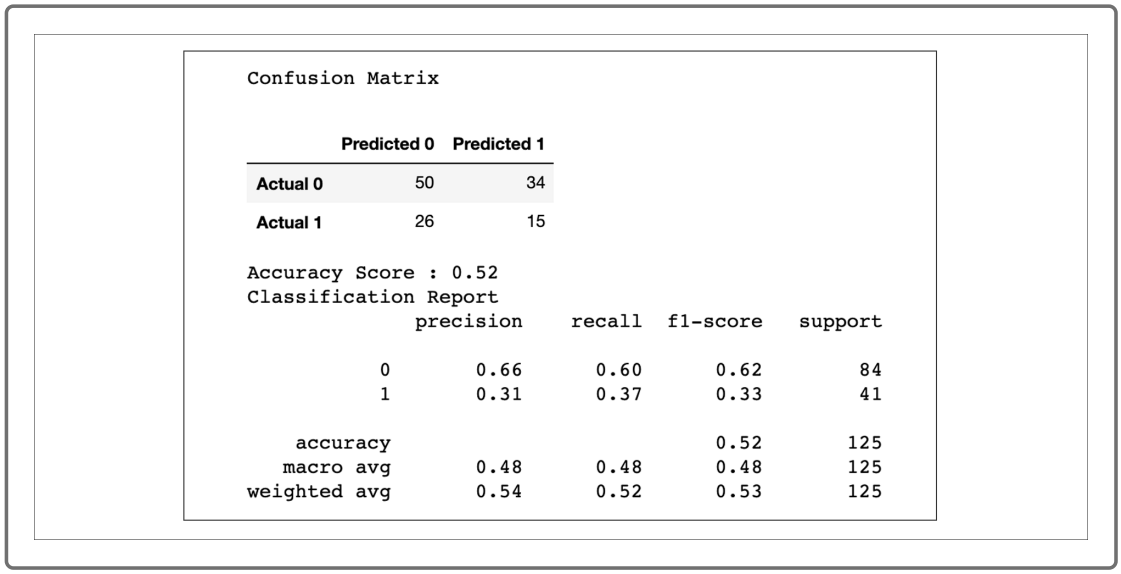
```
# Calculating the accuracy score.
acc_score = accuracy_score(y_test, predictions)
```

The accuracy of our model is 0.552, which can also be calculated as follows:

```
(True Positives (TP) + True Negatives (TN)) / Total = (50 + 19)/125 = 0.552
```

Lastly, we can print out the above results along with the classification report, which will give us the precision, recall, F1 score, and support for the two classes.

```
# Displaying results
print("Confusion Matrix")
display(cm_df)
print(f"Accuracy Score : {acc_score}")
print("Classification Report")
print(classification_report(y_test, predictions))
```



Confusion Matrix

	Predicted 0	Predicted 1
Actual 0	50	34
Actual 1	26	15

Accuracy Score : 0.52
Classification Report

	precision	recall	f1-score	support
0	0.66	0.60	0.62	84
1	0.31	0.37	0.33	41
accuracy			0.52	125
macro avg	0.48	0.48	0.48	125
weighted avg	0.54	0.52	0.53	125

Let's go over the results in the classification report:

- **Precision:** Precision is the measure of how reliable a positive classification is. From our results, the precision for the good loan applications can be determined by the ratio TP/(TP + FP), which is 50/(50 + 22) = 0.69. The precision for the bad loan applications can be determined as follows: 19/(19 + 34) = 0.358. A low precision is indicative of a large number of false positives—of the 53 loan

applications we predicted to be bad applications, 34 were actually good loan applications.

- **Recall:** Recall is the ability of the classifier to find all the positive samples. It can be determined by the ratio: TP/(TP + FN), or 50/(50 + 34) = 0.595 for the good loans and 19/(19 + 22) = 0.463 for the bad loans. A low recall is indicative of a large number of false negatives.
- **F1 score:** F1 score is a weighted average of the true positive rate (recall) and precision, where the best score is 1.0 and the worst is 0.0.
- **Support:** Support is the number of actual occurrences of the class in the specified dataset. For our results, there are 84 actual occurrences for the good loans and 41 actual occurrences for bad loans.

In summary, this model may not be the best one for preventing fraudulent loan applications because the model's accuracy, 0.552, is low, and the precision and recall are not good enough to state that the model will be good at classifying fraudulent loan applications. Modeling is an iterative process: you may need more data, more cleaning, another model parameter, or a different model. It's also important to have a goal that's been agreed upon, so that you know when the model is good enough.

NOTE

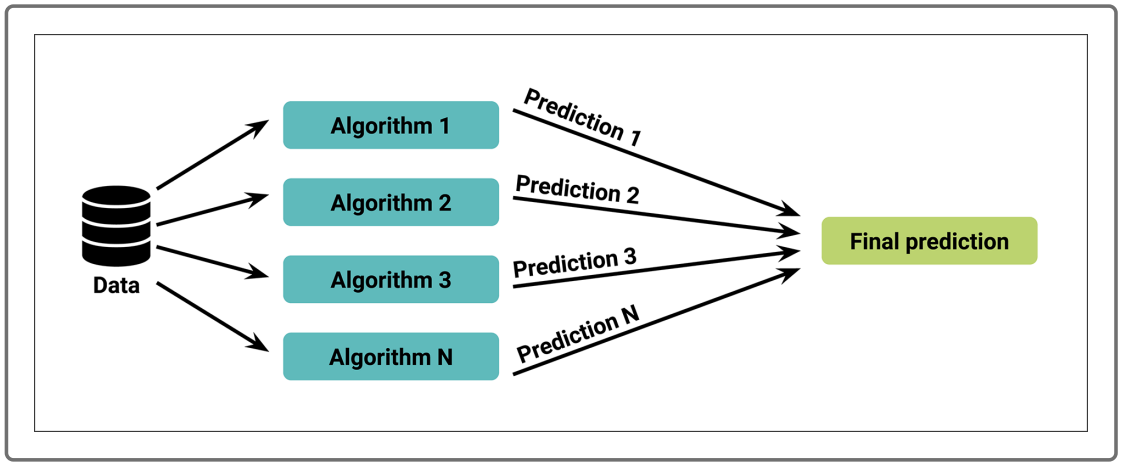
> Consult the **sklearn.metrics.precision_recall_fscore_support documentation (https://scikit-learn.org/stable/modules/generated/sklearn.metrics.precision_recall_fscore_support.html)** for additional information about the classification scores.

17.8.1 Overview of Ensemble Learning

As you and Jill discuss ways to improve a model's performance, she brings up ensemble learning. Ensemble learning builds on the idea that two is better than one. A single tree may be prone to errors, but many of them can be combined to form a stronger model. A random forest model, for example, combines many decision trees into a forest of trees.

The concept of ensemble learning is the process of combining multiple models, like decision tree algorithms, to help improve the accuracy and robustness, as well as decrease variance of the model, and therefore increase the overall performance of the model.

In the example below, there are multiple algorithms that are being used to learn and make their predictions based on the data. The final prediction is based on the accumulated predictions from each algorithm:



Weak Learners

Some algorithms are weak learners. Weak learners tend to have very few branches on the decision tree. A single weak learner will make inaccurate and imprecise predictions because they are poor at learning adequately as result of limited data, like too few features, or using data points that can't be classified.

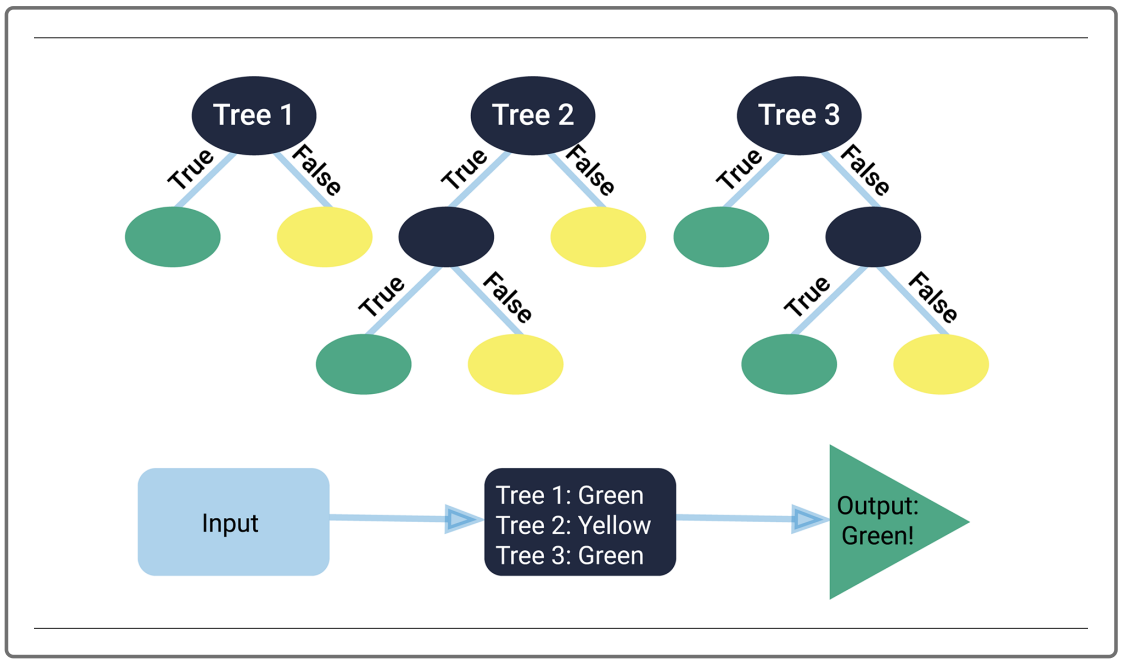
Weak learners shouldn't be considered unworthy. Weak learners are valuable because there are models that can combine many weak learners to create a more accurate and robust prediction engine. When we combine weak learners, together they can perform just as well as any strong learner.

Our loan application prediction algorithm can be considered a moderate to weak learner because it was not good at classifying bad loan applications. However, if we combine this decision tree with more decision trees—all using different training and testing sets—then the prediction may be more accurate.

We can combine weak learners using a specific algorithm, like Random Forests, GradientBoostedTree, and XGBoost. We will learn about Random Forests next, and cover gradient boosting later in this module.

Random Forests

Instead of having a single, complex tree like the ones created by decision trees, a random forest algorithm will sample the data and build several smaller, simpler decision trees. Each tree is simpler because it is built from a random subset of features:



These simple trees are weak learners because they are created by randomly sampling the data and creating a decision tree for only that small portion of data. And since they are trained on a small piece of the original data, they are only slightly better than a random guess. However, many *slightly better than average* small decision trees can be combined to create a strong learner, which has much better decision making power.

Random forest algorithms are beneficial because they:

- Are robust against overfitting as all of those weak learners are trained on different pieces of the data.
- Can be used to rank the importance of input variables in a natural way.
- Can handle thousands of input variables without variable deletion.

- Are robust to outliers and nonlinear data.
- Run efficiently on large datasets.

Since we determined that the decision tree in the previous example was not good at classifying bad loan applications, we're going to use the same loan applications' encoded dataset to predict bad loan applications using a random forest.

17.8.2 Predict Loan Applications

Jill now asks you to run a random forest model to make classifications. As you have done before, the first step is to prepare the data for the random forest classifier model.

When we imported our dependencies to create the decision tree in the previous example, we use the "tree" module from the sklearn library, `from sklearn import tree`.

For the random forest model, we'll use the "ensemble" module from the sklearn library. All the remaining dependencies will be the same. In the dependencies, replace `from sklearn import tree` with `from sklearn.ensemble import RandomForestClassifier` so that our dependencies look like the following.

```
# Initial imports.
import pandas as pd
from path import Path
from sklearn.ensemble import RandomForestClassifier
from sklearn.preprocessing import StandardScaler
```

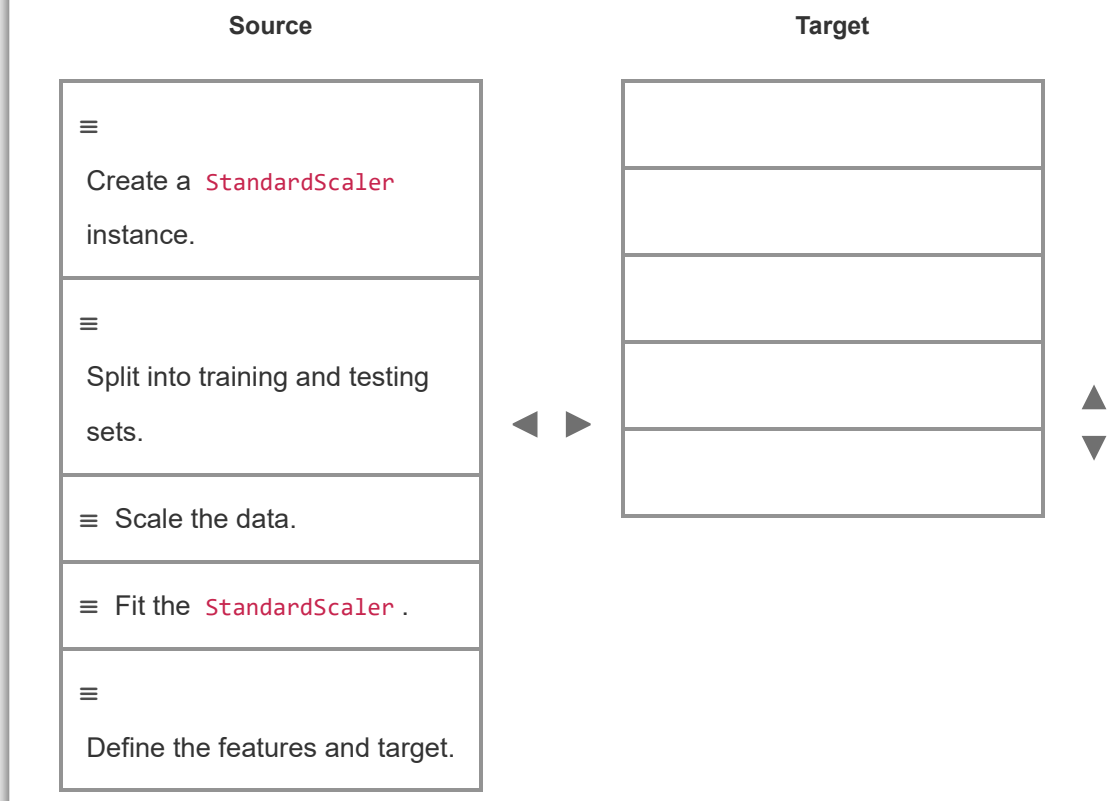
```
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, accuracy_score, classification
```

Next, read in your `loans_data_encoded.csv` file from previous exercises.

```
# Loading data
file_path = Path("../Resources/loans_data_encoded.csv")
df_loans = pd.read_csv(file_path)
df_loans.head()
```

After the data has been loaded, we're going to preprocess data just like we did for the decision tree model.

Prior to fitting with the random forest model, we have to preprocess the data. Place the following steps in the correct order for preprocessing data.



Check Answer

Finish ▸

Preprocess the Data

Now, we're going to walk through the preprocessing steps for the loan applications' encoded data so that we can fit our training and testing sets with the random forest model.

If you do not quite remember the steps for preprocessing, add the blocks of code in your Jupyter Notebook as follows.

1. First, we define the features set.

```
# Define the features set.
X = df_loans.copy()
X = X.drop("bad", axis=1)
X.head()
```

2. Next, we define the target set. Here, we're using the `ravel()` method, which performs the same procedure on our target set data as the `values` attribute.

```
# Define the target set.
y = df_loans["bad"].ravel()
y[:5]
```

3. Now, we split into the training and testing sets.

```
# Splitting into Train and Test sets.
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=7
```

4. Lastly, we can create the `StandardScaler` instance, fit the scaler with the training set, and scale the data.

```
# Creating a StandardScaler instance.
scaler = StandardScaler()
# Fitting the Standard Scaler with the training data.
X_scaler = scaler.fit(X_train)

# Scaling the data.
X_train_scaled = X_scaler.transform(X_train)
X_test_scaled = X_scaler.transform(X_test)
```

If you were able to do these steps without having to follow along, congratulations!

17.8.3 Fit the Model, Make Predictions, and Evaluate Results

Now that you have prepared the data, you will put the random forest classifier model to practice, then evaluate the results.

Now that we have preprocessed the data into training and testing data for both features and target sets, we can fit the random forest model, make predictions, and evaluate the model.

Fit the Random Forest Model

Before we fit the random forest model to our `X_train_scaled` and `y_train` training data, we'll create a random forest instance using the random forest classifier, `RandomForestClassifier()`.

```
# Create a random forest classifier.
rf_model = RandomForestClassifier(n_estimators=128, random_state=78)
```

The `RandomForestClassifier` takes a variety of parameters, but for our purposes we only need the `n_estimators` and the `random_state`.

NOTE

> Consult the sklearn documentation for additional information about the `RandomForestClassifier (https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html)` and the parameters it takes.

The `n_estimators` will allow us to set the number of trees that will be created by the algorithm. Generally, the higher number makes the predictions stronger and more stable, but can slow down the output because of the higher training time allocated. The best practice is to use between 64 and 128 random forests, though higher numbers are quite common despite the higher training time. For our purposes, we'll create 128 random forests.

After we create the random forest instance, we need to fit the model with our training sets.

```
# Fitting the model
rf_model = rf_model.fit(X_train_scaled, y_train)
```

Make Predictions Using the Testing Data

After fitting the model, we can run the following code to make predictions using the scaled testing data:

```
# Making predictions using the testing data.
predictions = rf_model.predict(X_test_scaled)
```

The output will be similar as when the predictions were determined for the decision tree.

```
predictions
```

```
array([1, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       1, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 1, 1, 1, 1, 0,
       1, 1, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 1,
       0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1,
       0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0,
       0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0])
```

Evaluate the Model

After making predictions on the scaled testing data, we analyze how well our random forest model classifies loan applications by using the `confusion_matrix`.

```
# Calculating the confusion matrix.
cm = confusion_matrix(y_test, predictions)

# Create a DataFrame from the confusion matrix.
cm_df = pd.DataFrame(
    cm, index=["Actual 0", "Actual 1"], columns=["Predicted 0", "Predicted 1

cm_df
```

	Predicted 0	Predicted 1
Actual 0	51	33
Actual 1	23	18

Based on the results in the confusion matrix DataFrame, which of the following is incorrect?. For the purposes of this question, 0 is considered positive, and 1 is considered negative.

- There are 51 true positives.
- There are 23 false positives.
- There are 23 true negatives.
- There are 33 false negatives.

Check Answer

Finish ▸

These results are relatively the same as the decision tree model. To improve our predictions, let's increase the `n_estimators` to 500. After running all the code again, after changing the `n_estimators` to 500, our confusion matrix DataFrame is about the same as before.

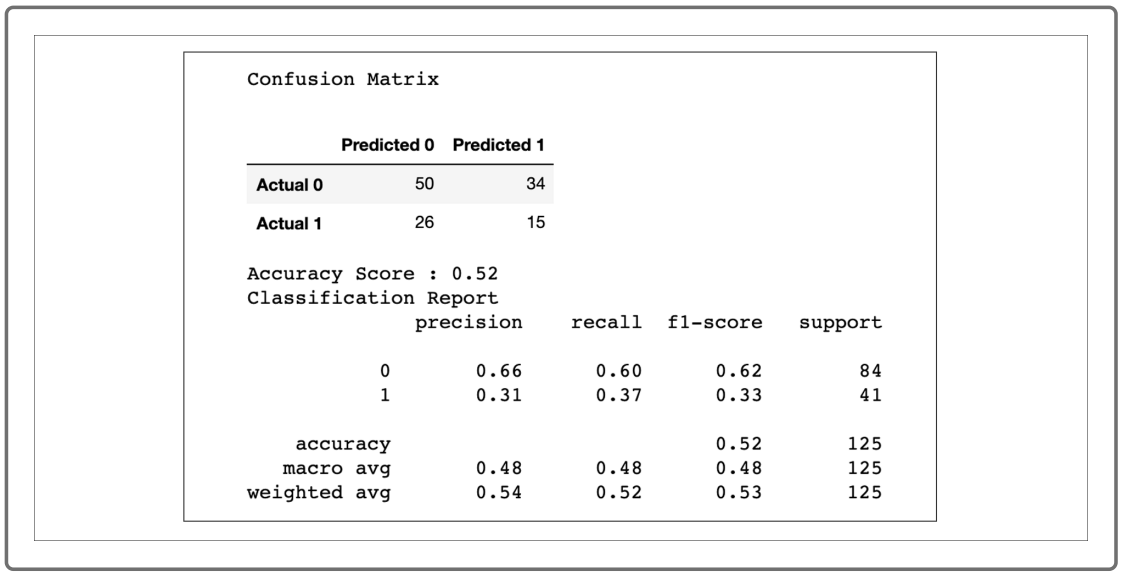
	Predicted 0	Predicted 1
Actual 0	50	34
Actual 1	26	15

Using the equation `(TP + TN) / Total`, we can determine our accuracy (determine how often the classifier predicts correctly) by running the following code. For this model, our accuracy score is 0.520:

```
# Calculating the accuracy score.
acc_score = accuracy_score(y_test, predictions)
```

Lastly, we can print out the above results along with the classification report for the two classes:

```
# Displaying results
print("Confusion Matrix")
display(cm_df)
print(f"Accuracy Score : {acc_score}")
print("Classification Report")
print(classification_report(y_test, predictions))
```



Confusion Matrix

	Predicted 0	Predicted 1
Actual 0	50	34
Actual 1	26	15

Accuracy Score : 0.52
Classification Report

	precision	recall	f1-score	support
0	0.66	0.60	0.62	84
1	0.31	0.37	0.33	41
accuracy			0.52	125
macro avg	0.48	0.48	0.48	125
weighted avg	0.54	0.52	0.53	125

From the confusion matrix results, the precision for the the bad loan applications is low, indicating a large number of false positives, which indicates an unreliable positive classification. The recall is also low for the bad loan applications, which is indicative of a large number of false negatives. The F1 score is also low (33).

In summary, this random forest model is not good at classifying fraudulent loan applications because the model's accuracy, 0.520, and F1 score are low.

Rank the Importance of Features

One nice byproduct of the random forest algorithm is to rank the features by their importance, which allows us to see which features have the most impact on the decision.

To calculate the feature importance, we can use the `feature_importances_` attribute with the following code:

```
# Calculate feature importance in the Random Forest model.
importances = rf_model.feature_importances_
importances
```

The output from this code returns an array of scores for the features in the `X_test` set, whose sum equals 1.0:

```
importances
```

```
array([0.05454782, 0.07997292, 0.43280448, 0.32973986, 0.01887172,
       0.02110219, 0.00271658, 0.02151063, 0.01887818, 0.01985562])
```

To sort the features by their importance with the column in the `X_test` set, we can modify our code above as follows:

```
# We can sort the features by their importance.
sorted(zip(rf_model.feature_importances_, X.columns), reverse=True)
```

In the code, the sorted function will sort the zipped list of features with their column name (`X.columns`) in reverse order—more important features first—with `reverse=True`.

Running this code will return the following output:

```
[(0.43280447750315343, 'age'),
 (0.32973986443922343, 'month_num'),
 (0.07997292251445517, 'term'),
 (0.05454782107242418, 'amount'),
 (0.021510631303272416, 'education_college'),
 (0.021102188881175144, 'education_High School or Below'),
 (0.01985561654170213, 'gender_male'),
 (0.018878176828577283, 'gender_female'),
 (0.018871722006693077, 'education_Bachelor'),
 (0.002716578909323729, 'education_Master or Above')]
```

Now we can clearly see which features, or columns, of the loan application are more relevant. The `age` and `month_num` of the loan application are the more relevant features.

To improve this model, we can drop some of the lower ranked features.

Drop some of the lower ranked features, like education and/or gender. Does dropping these features improve our random forest model?

- Yes
- No

Check Answer

Finish ▸

17.9.1 Bootstrap Aggregation

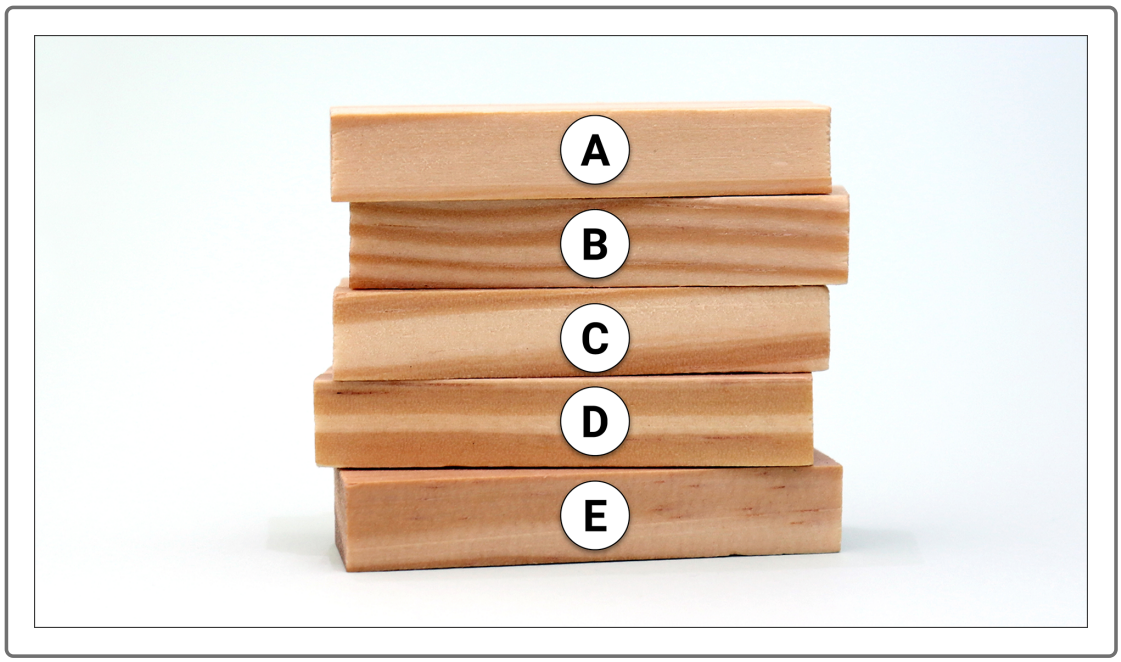
Bootstrap aggregation, or "bagging," is an ensemble learning technique that combines weak learners into a strong learner. In fact, you have already seen a model that uses bootstrap aggregation as part of its algorithm: the random forest model.

Jill reminds you that decision trees are prone to overfitting, meaning that the algorithm's predictions are excessively tailored to the specific dataset. When there's overfitting, a model's performance will suffer when it encounters a new dataset. One way to try to overcome this problem is with bootstrap aggregation. Let's look at how it works in more detail.

Bootstrap aggregation, also called bagging, is a machine learning technique used to combine weak learners into a strong learner. Bagging is composed of two parts: bootstrapping and aggregation.

Bootstrapping

Bootstrapping is a sampling technique in which samples are randomly selected, then returned to the general pool and replaced, or put back into the general pool. As a concrete example, picture your dataset as a bag containing five wooden blocks, each labeled with the letter A, B, C, D, or E.



Imagine that you draw samples of the dataset from this bag three times. Since each sampling is the same size as the original dataset, you must draw five blocks for each sample. To do so, you grab a wooden block randomly from the bag, and after noting which block you drew, you replace it, meaning that you put it back into the bag. Because you return the block to the bag, it's possible to draw the same block again in the next draw. You repeat the process until you have a sample whose size is the same as the original dataset. The result might appear as follows:

Sample 1: A, A, A, B, D

Sample 2: A, B, B, C, E

Sample 3: B, C, D, D, E

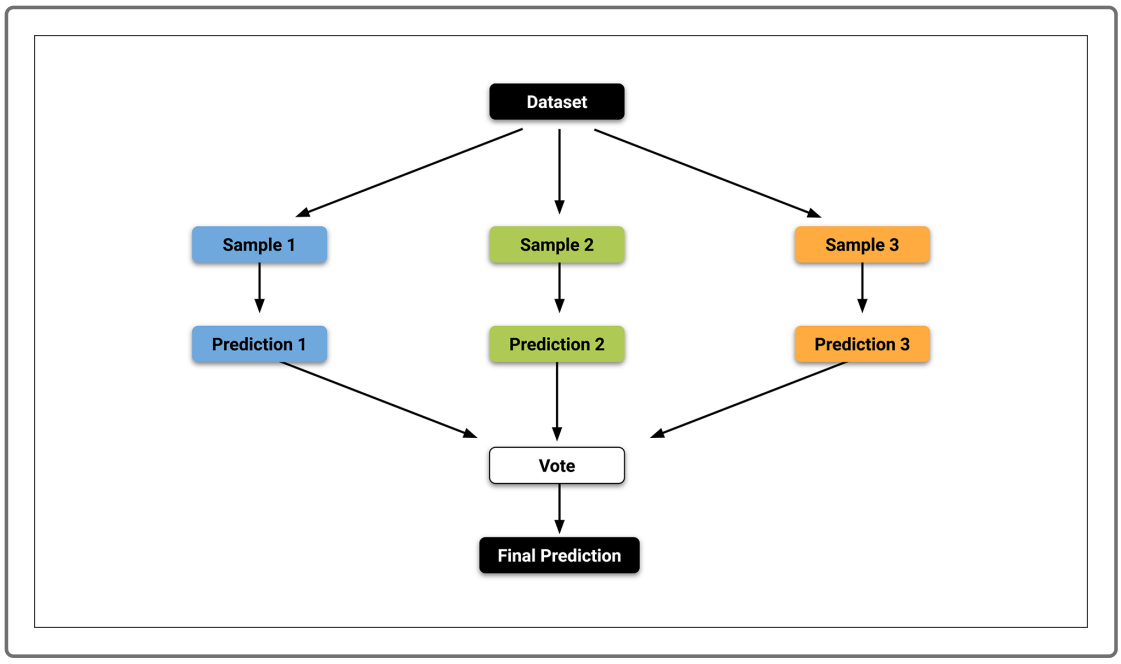
In our example above, each sample contains multiple occurrences of the same block. Sample 1 drew the letter A three times, Sample 2 drew the

letter B twice, and Sample 3 drew the letter D twice. In other words, each observation (letter) may occur repeatedly in any given sample. In real life, this means that if your dataset were a Pandas DataFrame, a given row may occur multiple times in a sample.

In summary, bootstrapping is simply a sampling technique with which a number of samples are made, and in which an observation can occur multiple times.

Aggregation

In the aggregation step, different classifiers are run, using the samples drawn in the bootstrapping stage. Each classifier is run independently of the others, and all the results are aggregated via a voting process. Each classifier will vote for a label (a prediction). The final prediction is the one with the most votes.



A dataset consists of the following items: 1, 2, 3, 4, and 5. Which of the following could be a bootstrap sample?

- 1, 2, 3, 4, 5
- 1, 1, 2, 4, 4
- Both A and B.

Check Answer

Finish ▸

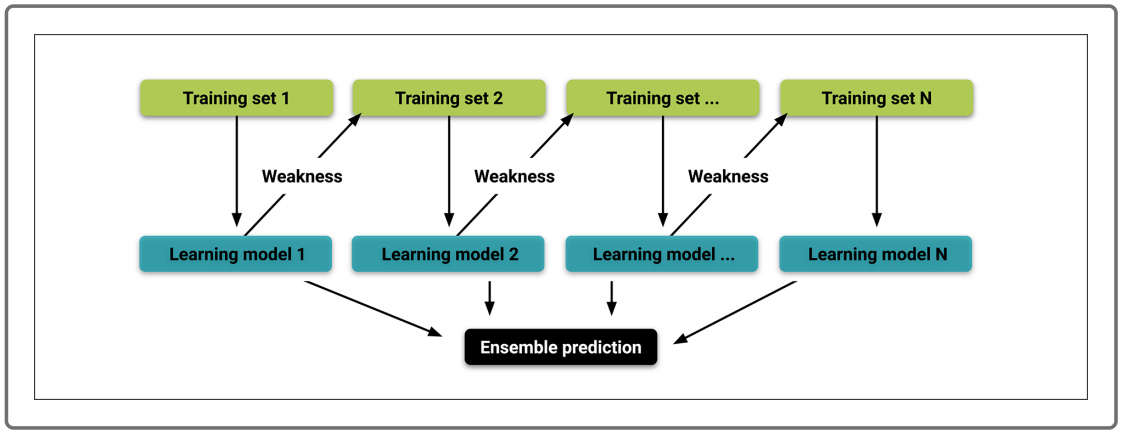
17.9.2 Boosting

Boosting is another technique to combine weak learners into a strong learner. However, there is a major difference between bagging and boosting. In bagging, as you have seen, multiple weak learners are combined at the same time to arrive at a combined result.

In boosting, however, the weak learners are not combined at the same time. Instead, they are used sequentially, as one model learns from the mistakes of the previous model.

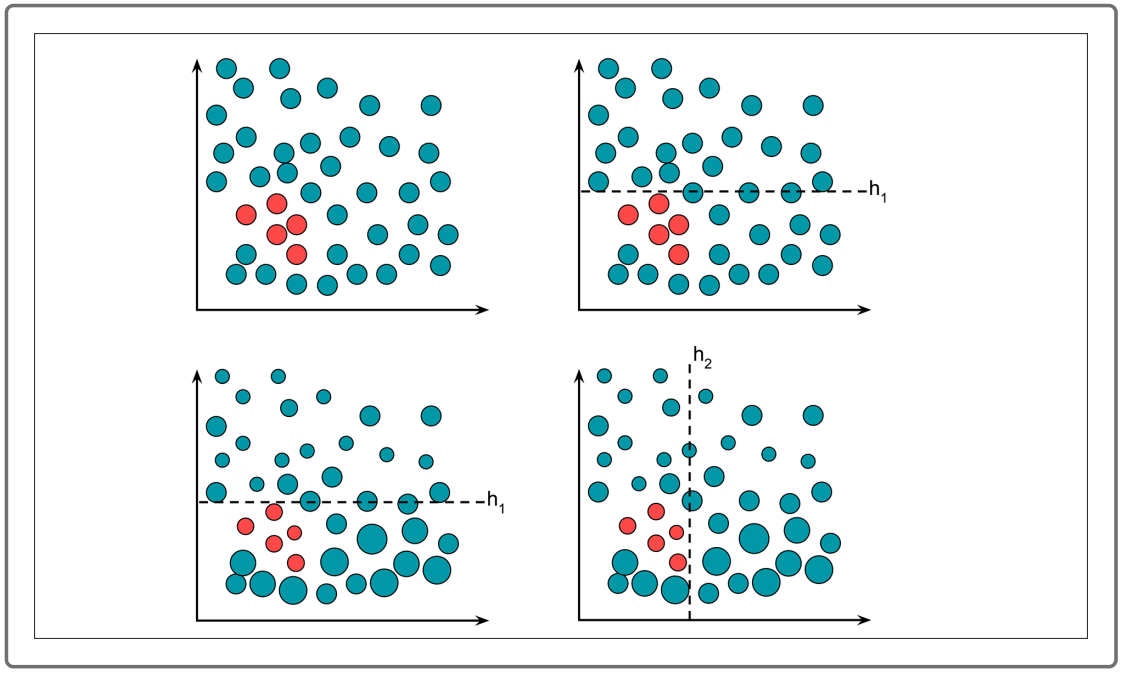
Jill assures you that learning this ensemble learning technique will be worth your time. After all, many machine learning competitions have been won with this powerful technique.

Like bagging, boosting is also a technique to combine a set of weak learners into a strong learner. We saw in bagging that the different models work independently of one another. In contrast, boosting trains a sequence of weak models. As shown below, each model learns from the errors of the previous model, and the models form an ensemble:

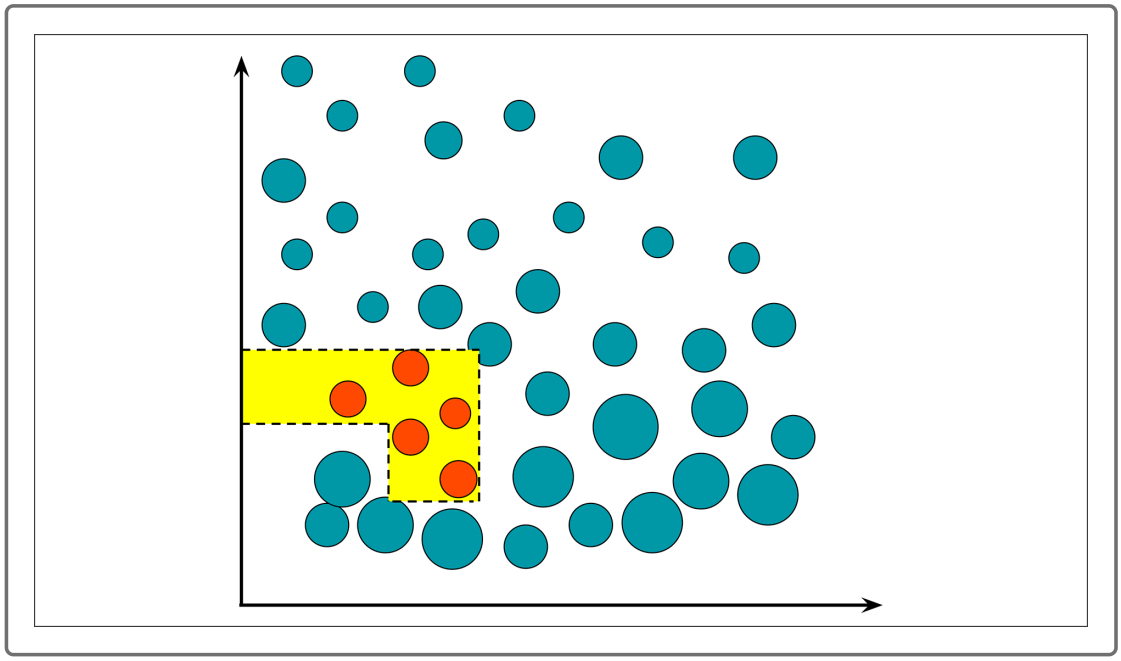


Adaptive Boosting

The idea behind Adaptive Boosting, called AdaBoost, is easy to understand. In AdaBoost, a model is trained then evaluated. After evaluating the errors of the first model, another model is trained. This time, however, the model gives extra weight to the errors from the previous model. The purpose of this weighting is to minimize similar errors in subsequent models. Then, the errors from the second model are given extra weight for the third model. This process is repeated until the error rate is minimized:



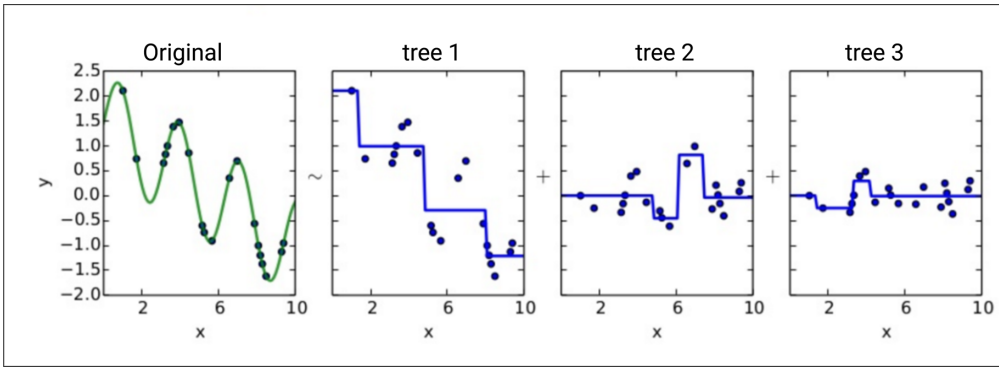
The final classifier might appear like the following:



Gradient Boosting

Gradient boosting, like AdaBoost, is an ensemble method that works sequentially. In contrast to AdaBoost, gradient boosting does not seek to minimize errors by adjusting the weight of the errors. Instead, it follows this process:.

1. A small tree (called a stump) is added to the model, and the errors are evaluated.
2. A second stump is added to the first and attempts to minimize the errors from the first stump. These errors are called pseudo-residuals.
3. A third stump is added to the first two and attempts to minimize the pseudo-residuals from the previous two.
4. The process is repeated until the errors are minimized as much as possible, or until a specified number of repetitions has been reached:



In gradient boosting, the learning rate refers to how aggressively pseudo-residuals are corrected during each iteration. In general, it is preferable to begin with a lower learning rate and, if necessary, adjust the rate upward.

Gradient boosting is a powerful technique that is often used in machine learning competitions.

17.9.3 Boosting in Practice

Armed with an understanding of boosting, you're excited to put this technique into practice. Jill encourages you to dive into gradient boosting. Your next step is to apply your knowledge to Python code.

Let's look at an example of using a gradient boosted tree model to enhance the performance of weak learners by combining them into an ensemble.

Start by downloading the files you'll need for this section.

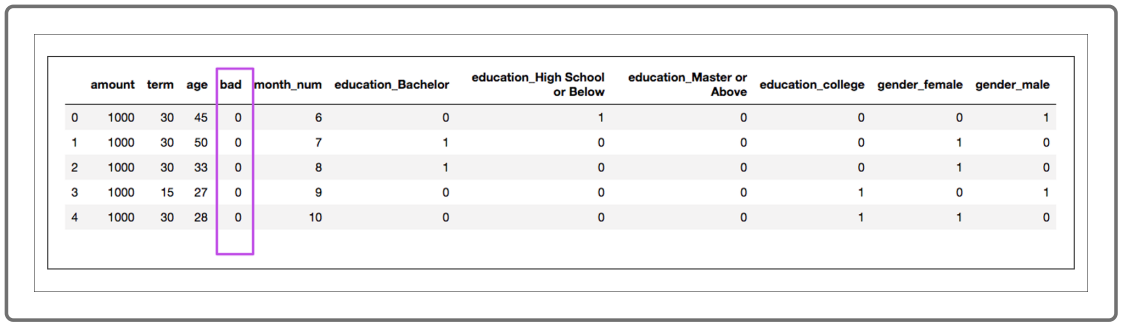
Download 17-9-3-gradient_boosted_tree.zip (https://courses.bootcampspot.com/courses/138/files/22512/download?wrap=1)

Open `gradient_boosted_tree.ipynb`, or if you like, create a new notebook in the same location, and place the following code.

```
import pandas as pd
from path import Path
```

```
file_path = Path("../Resources/loans_data_encoded.csv")
loans_df = pd.read_csv(file_path)
loans_df.head()
```

A preview of the DataFrame reveals that the dataset again contains information on loan applications. The bad column is the target column, with 0 indicating a good loan application and 1 indicating a bad loan application.



	amount	term	age	bad	month_num	education_Bachelor	education_High School or Below	education_Master or Above	education_college	gender_female	gender_male
0	1000	30	45	0	6	0	1	0	0	0	1
1	1000	30	50	0	7	1	0	0	0	1	0
2	1000	30	33	0	8	1	0	0	0	1	0
3	1000	15	27	0	9	0	0	0	1	0	1
4	1000	30	28	0	10	0	0	0	1	1	0

Then, separate the feature columns from the target column.

```
X = loans_df.copy()
X = X.drop("bad", axis=1)
y = loans_df["bad"].values
```

Next, split the dataset into training and testing sets. Again, the `random_state` argument is optional.

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
X_train, X_test, y_train, y_test = train_test_split(X,
    y, random_state=1)
```

The data is scaled in the next step. Scaling is typically necessary when using models that calculate distances between data points, such as SVM.

While not strictly required for tree-based models, it can be a good idea to scale the data, especially when comparing the performances of different models.

```
scaler = StandardScaler()
X_scaler = scaler.fit(X_train)
X_train_scaled = X_scaler.transform(X_train)
X_test_scaled = X_scaler.transform(X_test)
```

In the next step, a `for` loop is used to identify the learning rate that yields the best performance.

```
from sklearn.ensemble import GradientBoostingClassifier
learning_rates = [0.05, 0.1, 0.25, 0.5, 0.75, 1]
for learning_rate in learning_rates:
   classifier = GradientBoostingClassifier(n_estimators=20,
   learning_rate=learning_rate,
   max_features=5,
   max_depth=3,
   random_state=0)
   classifier.fit(X_train_scaled, y_train.ravel)
```

The `GradientBoostingClassifier` includes the following:

- An array called `learning_rates` is manually created and contains a range of values.
- For each learning rate value, a `GradientBoostingClassifier` model is instantiated.
- The `max_depth` argument refers to the size of the decision tree stumps used in gradient boosting.
- The `n_estimators` argument refers to the number of trees used.
- The `n_estimators`, `max_features`, and `max_depth` parameters are fixed at the defined values. These, like the learning rate, can be optimized,

but we'll stick to the default values used in the example above.

During each iteration of the `for` loop, the accuracy scores of the training and testing sets are also printed for each learning rate.

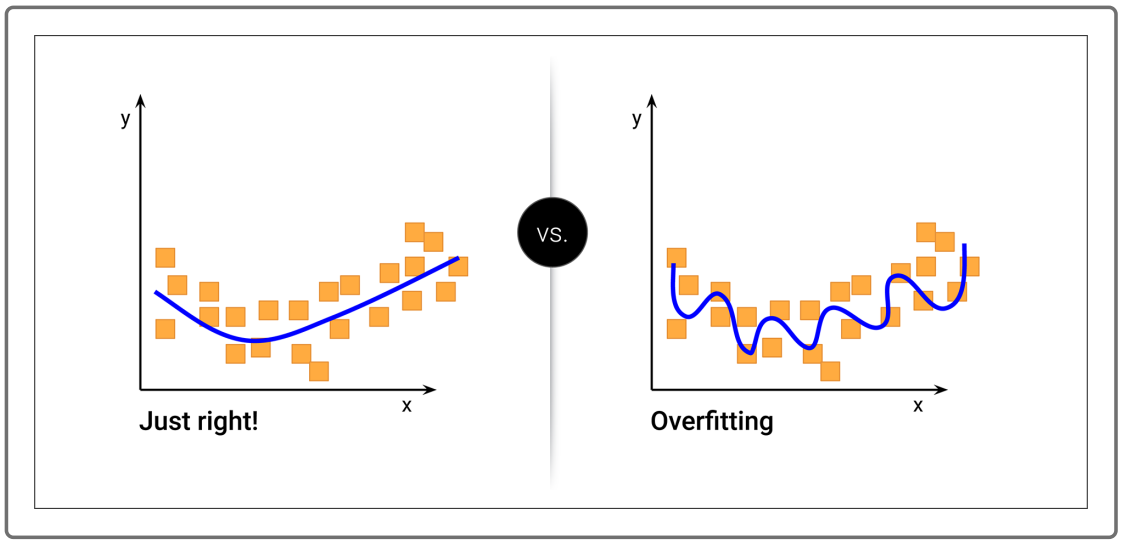
```
    print("Learning rate: ", learning_rate)
    print("Accuracy score (training): {0:.3f}".format(
        classifier.score(
            X_train_scaled,
            y_train)))
    print("Accuracy score (validation): {0:.3f}".format(
        classifier.score(
            X_test_scaled,
            y_test)))
```

Previously, we used Scikit-learn's `accuracy_score` module to validate a model. The method used here is `classifier.score()`, which yields the same result.

Of the learning rates used, 0.5 yields the best accuracy score for the testing set and a high accuracy score for the training set. This is the value we'll implement in the final model. Also, note that the testing accuracy is more important here than the training accuracy.

```
Learning rate:  0.05
Accuracy score (training): 0.627
Accuracy score (validation): 0.520

Learning rate:  0.1
Accuracy score (training): 0.667
Accuracy score (validation): 0.528

Learning rate:  0.25
Accuracy score (training): 0.723
Accuracy score (validation): 0.536

Learning rate:  0.5
Accuracy score (training): 0.755
Accuracy score (validation): 0.560

Learning rate:  0.75
Accuracy score (training): 0.781
Accuracy score (validation): 0.520

Learning rate:  1
Accuracy score (training): 0.792
Accuracy score (validation): 0.480
```

A model that performs well on the training set but poorly on the testing set is said to be “overfit.” Overfitting is akin to memorizing the answers to an exam: It will help on that particular exam, but not on any others. In other words, overfitting occurs when a model gives undue importance to patterns within a particular dataset that are not found in other, similar datasets. Instead of learning a general pattern that can be applied to other similar datasets, it learns the patterns specific to one dataset.



Which of the following is an example of overfitting?

- Training score: 0.9; testing score: 0.9
- Training score: 0.9; testing score: 0.6
- Training score: 0.8; testing score: 0.85

Check Answer

Finish ▸

Using the `learning_rate` value obtained from the `for` loop, we instantiate a model, train it, then create predictions.

```
classifier = GradientBoostingClassifier(n_estimators=20,
   learning_rate=0.5, max_features=5, max_depth=3, random_state=0)

classifier.fit(X_train_scaled, y_train)
predictions = classifier.predict(X_test_scaled)
```

Having created predictions with the gradient boosted tree model, we can assess the model's performance. This time, the `accuracy_score()` method is used.

```
from sklearn.metrics import confusion_matrix
from sklearn.metrics import accuracy_score
from sklearn.metrics import classification_report
acc_score = accuracy_score(y_test, predictions)
print(f"Accuracy Score : {acc_score}")
```

Predictably, the `accuracy_score()` method returns the same score as that of the `classifier.score()` method.

```
Accuracy Score: 0.56
```

Next, we generate a `confusion_matrix` of the results.

```
cm = confusion_matrix(y_test, predictions)
cm_df = pd.DataFrame(
   cm, index=["Actual 0", "Actual 1"],
   columns=["Predicted 0", "Predicted 1"]
)
display(cm_df)
```

	Predicted 0	Predicted 1
Actual 0	49	16
Actual 1	39	21

Finally, we can generate a classification report to evaluate the precision, recall, and F1 scores.

```
print("Classification Report")
print(classification_report(y_test, predictions))
```

```
Classification Report
              precision    recall  f1-score   support

           0       0.56      0.75      0.64        65
           1       0.57      0.35      0.43        60

    accuracy                           0.56       125
   macro avg       0.56      0.55      0.54       125
weighted avg       0.56      0.56      0.54       125
```

SKILL DRILL

How would you interpret the results of this classification report?

17.10.1 Oversampling

Jill congratulates you on the great work you've done so far. Well done! Before setting you free to tackle your machine learning assignment, however, she would like you to become familiar with a family of resampling techniques designed to deal with class imbalance.

Class imbalance is a common problem in classification. It occurs when one class is much larger than the other class. For example, if you work for a credit card company and want to detect fraudulent transactions, you will deal with many more non-fraudulent transactions than fraudulent ones. In this case, the non-fraudulent class is much larger than the fraudulent class.

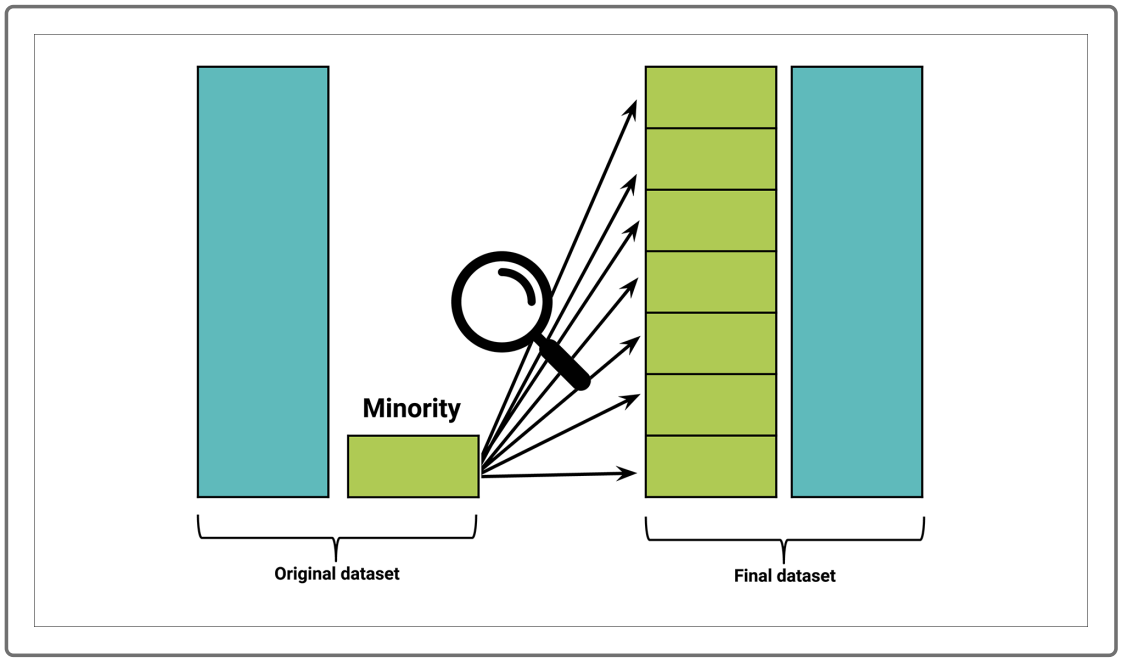
Under Jill's tutelage, you will learn about three techniques to address class imbalance: oversampling, undersampling, and a combination approach of oversampling and undersampling. But for now, you'll start with oversampling.

Class imbalance refers to a situation in which the existing classes in a dataset aren't equally represented. Earlier we discussed a fraud detection scenario in which a large number of credit card transactions are legitimate, and only a small number are fraudulent. For example, let's say

that out of 100,000 transactions, 50 are fraudulent and the rest are legitimate. The pronounced imbalance between the two classes (fraudulent and non-fraudulent) can cause machine learning models to be biased toward the majority class. In such a case, the model will be much better at predicting non-fraudulent transactions than fraudulent ones. This is a problem if the goal is to detect fraudulent transactions!

In such a case, even a model that blindly classifies every transaction as non-fraudulent will achieve a very high degree of accuracy. As we saw previously, one strategy to deal with class imbalance is to use appropriate metrics to evaluate a model's performance, such as precision and recall.

Another strategy is to use **oversampling**. The idea is simple and intuitive: If one class has too few instances in the training set, we choose more instances from that class for training until it's larger.



We'll discuss two oversampling techniques: random oversampling and synthetic minority oversampling technique.

Random Oversampling

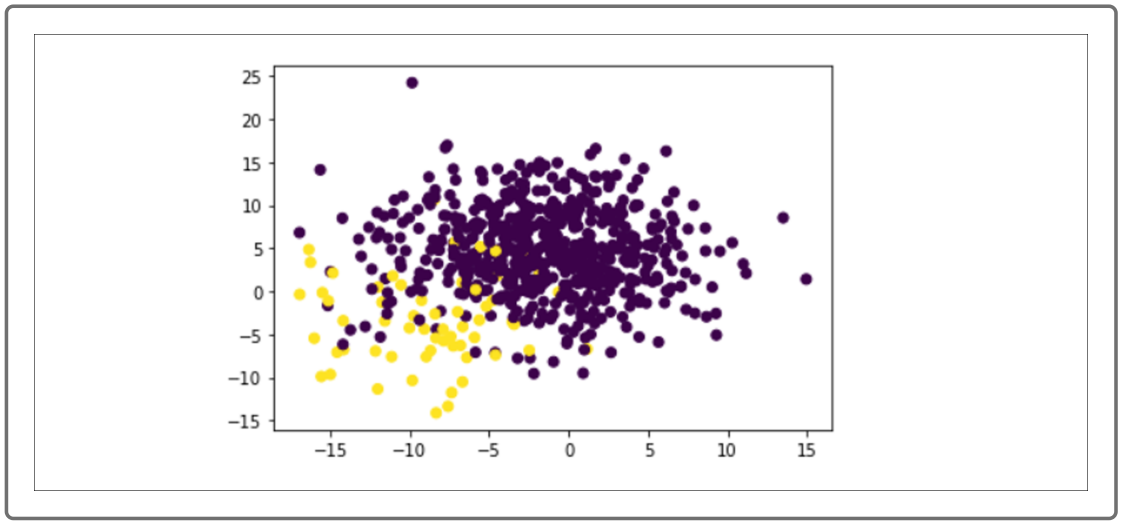
In **random oversampling,** instances of the minority class are randomly selected and added to the training set until the majority and minority classes are balanced. The Python implementation is simple.

To get started, download the necessary files.

Download 17-10-1-oversampling.zip (https://courses.bootcampspot.com/courses/138/files/22518/download?wrap=1)

Open the downloaded Jupyter Notebook. After importing dependencies, an unbalanced dataset with two classes is artificially created and plotted, as shown in the following code blocks and resulting chart.

```
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
from collections import Counter

X, y = make_blobs(n_samples=[600, 60], random_state=1, cluster_std=5)
plt.scatter(X[:, 0], X[:, 1], c=y)
plt.show()
```



The visualization confirms the imbalance—the purple class visibly outnumbers the yellow class.

Next, the dataset is split into training and testing sets.

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=1)
Counter(y_train)
```

Python's `Counter` module confirms the imbalance in the training set. There are 451 samples from the purple class and 44 samples from the yellow class.

```
Counter({0: 451, 1: 44})
```

Next, we randomly oversample the minority class with the `imblearn` library.

```
from imblearn.over_sampling import RandomOverSampler
ros = RandomOverSampler(random_state=1)
X_resampled, y_resampled = ros.fit_resample(X_train, y_train)
```

Let's break down what's happening here:

- An instance of `RandomOverSampler` is instantiated as `ros`.
- The training data (`X_train` and `y_train`) is resampled using the `fit_resample()` method.
- The results are called `X_resampled` and `y_resampled`.

Counting the classes of the resampled target verifies that the minority class has been enlarged.

```
Counter(y_resampled)

Counter({0: 17532, 1: 4968})
```

With a resampled dataset, we can now carry out the familiar pattern of training a model, making predictions, and evaluating the model's performance. For this example, we'll use a `LogisticRegression` model. The following code instantiates and trains the model.

```
from sklearn.linear_model import LogisticRegression
model = LogisticRegression(solver='lbfgs', random_state=1)
model.fit(X_resampled, y_resampled)
```

The model creates predictions. We then generate a `confusion_matrix` with the results.

```
from sklearn.metrics import confusion_matrix
y_pred = model.predict(X_test)
confusion_matrix(y_test, y_pred)
```

To assess the accuracy score of the model, we'll use the `balanced_accuracy_score` module.

```
from sklearn.metrics import balanced_accuracy_score
balanced_accuracy_score(y_test, y_pred)
```

The accuracy score is high at around 90%, but this number can be misleading, especially in an unbalanced dataset. Let's examine the classification report to assess the results further. We'll use the `classification_report_imbalanced` to do so.

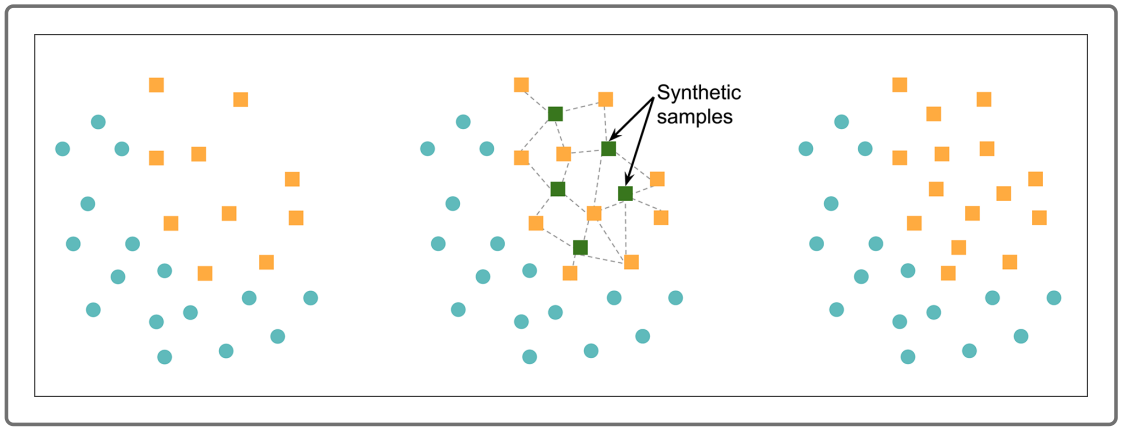
```
from imblearn.metrics import classification_report_imbalanced
print(classification_report_imbalanced(y_test, y_pred))
```

	pre	rec	spe	f1	geo	iba	sup
0	0.99	0.88	0.94	0.93	0.91	0.82	149
1	0.45	0.94	0.88	0.61	0.91	0.83	16
avg / total	0.94	0.88	0.93	0.90	0.91	0.82	165

While precision (“pre” column) and recall (“rec” column) are high for the majority class, precision is low for the minority class.

Synthetic Minority Oversampling Technique

The **synthetic minority oversampling technique (SMOTE)** is another oversampling approach to deal with unbalanced datasets. In SMOTE, like random oversampling, the size of the minority is increased. The key difference between the two lies in how the minority class is increased in size. As we have seen, in random oversampling, instances from the minority class are randomly selected and added to the minority class. In SMOTE, by contrast, new instances are interpolated. That is, for an instance from the minority class, a number of its closest neighbors is chosen. Based on the values of these neighbors, new values are created.



Which of the following best describes the difference between random oversampling and SMOTE?

- Random oversampling generates synthetic observations, whereas SMOTE draws from existing observations.
- Random oversampling draws from existing observations, whereas SMOTE generates synthetic observations.
- There is no difference between the two. SMOTE is the technical name for random oversampling.

Check Answer

Finish ▸

Let's look at SMOTE in action. Note that the following code is contained in the same Jupyter Notebook as the random oversampling example, and that we are using the same training data (`X_train` and `y_train`). We use the `SMOTE` module from the `imblearn` library to oversample the minority class.

```
from imblearn.over_sampling import SMOTE
X_resampled, y_resampled = SMOTE(random_state=1,
sampling_strategy='auto').fit_resample(
   X_train, y_train)
```

The following actions are taking place:

- The `sampling_strategy` argument specifies how the dataset is resampled. By default, it increases the minority class size to equal the majority class's size.
- Again, the `fit_resample()` method is used on the training data to train the SMOTE model and to oversample in a single step.

Counting the number of instances by class verifies that they are now equal in size.

```
Counter(y_resampled)

Counter({0:451, 1:451})
```

We'll again train a `LogisticRegression` model, predict, then assess the accuracy and generate a `confusion_matrix`, as shown in the following code blocks.

```
model = LogisticRegression(solver='lbfgs', random_state=1)
model.fit(X_resampled, y_resampled)

y_pred = model.predict(X_test)
balanced_accuracy_score(y_test, y_pred)

confusion_matrix(y_test, y_pred)

print(classification_report_imbalanced(y_test, y_pred))
```

	pre	rec	spe	f1	geo	iba	sup
0	0.99	0.89	0.94	0.94	0.91	0.83	149
1	0.48	0.94	0.89	0.64	0.91	0.84	16
avg / total	0.94	0.90	0.93	0.91	0.91	0.83	165

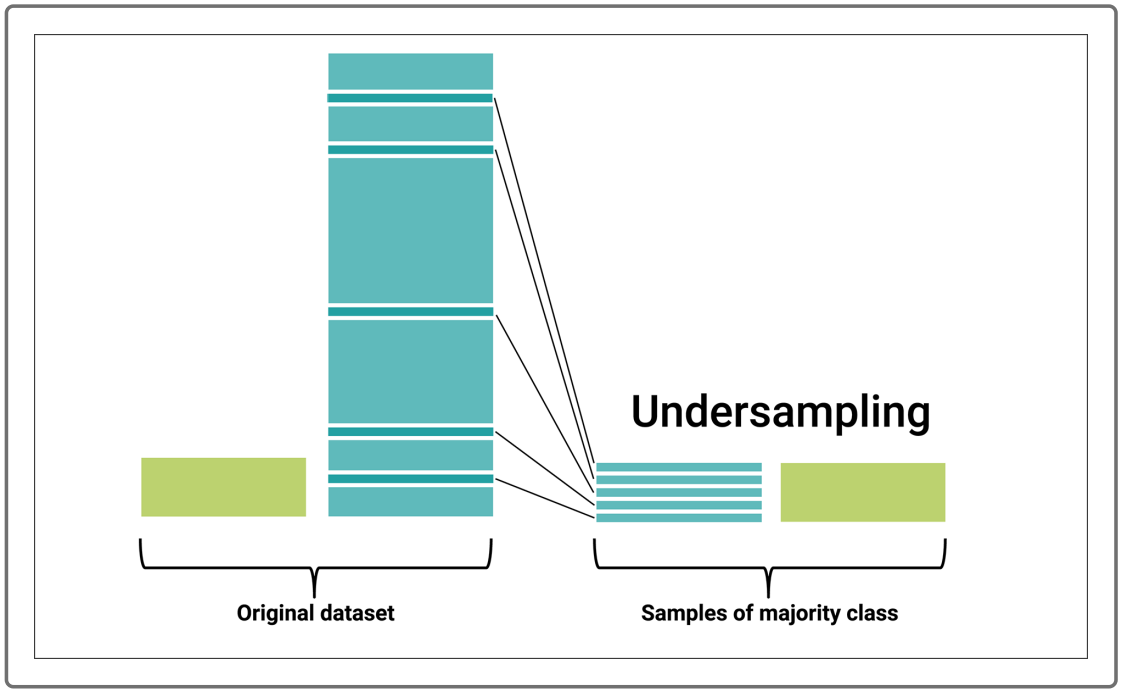
The metrics of the minority class (precision, recall, and F1 score) are slightly improved over those of random oversampling.

It's important to note that although SMOTE reduces the risk of oversampling, it does not always outperform random oversampling. Another deficiency of SMOTE is its vulnerability to outliers. We said earlier that a minority class instance is selected, and new values are generated based on its distance from its neighbors. If the neighbors are extreme outliers, the new values will reflect this. Finally, keep in mind that sampling techniques cannot overcome the deficiencies of the original dataset!

17.10.2 Undersampling

You've learned that in oversampling, the smaller class is resampled to make it larger. Undersampling, in contrast, takes the opposite tack. Jill recommends that you look at random undersampling as well as a synthetic approach.

Undersampling is another technique to address class imbalance. Undersampling takes the opposite approach of oversampling. Instead of increasing the number of the minority class, the size of the majority class is decreased.



Keep in mind that both oversampling and undersampling involve tradeoffs. Oversampling addresses class imbalance by duplicating or mimicking existing data. In contrast, undersampling only uses actual data. On the other hand, undersampling involves loss of data from the majority class. Furthermore, undersampling is practical only when there is enough data in the training set. There must be enough usable data in the undersampled majority class for a model to be useful.

We'll discuss two approaches to undersampling: random and cluster centroid. Both are similar to the oversampling methods we've seen. Download the files and let's look at the examples together.

Download 17-10-2-undersampling.zip (https://courses.bootcampspot.com/courses/138/files/22525/download?wrap=1)

Random Undersampling

In random undersampling, randomly selected instances from the majority class are removed until the size of the majority class is reduced, typically

to that of the minority class. The dataset used in this example contains information on credit card default.

```
import pandas as pd
from path import Path
from collections import Counter

data = Path('../Resources/cc_default.csv')
df = pd.read_csv(data)
df.head()
```

	ID	ln_balance_limit	sex	education	marriage	age	default_next_month
0	1	9.903488	1	2	0	24	1
1	2	11.695247	1	2	1	26	1
2	3	11.407565	1	2	1	34	0
3	4	10.819778	1	2	0	37	0
4	5	10.819778	0	2	0	57	0

The following legend explains the values used in the columns:

- ln_balance_limit: maximum balance limit on a card
- sex: 1 = female, 0 = sex
- education: 1 = graduate school, 2 = university, 3 = high school, 4 = others
- marriage: 1 = married, 0 = single
- age: age of credit card holder
- default_next_month: 1 = yes, 0 = no

The target variable, `default_next_month`, describes whether a credit card holder defaults during the next month. The features are all columns in the dataset, except `ID` and `default_next_month`, with the former not a useful predictor of default status and the latter the target. We can drop these

columns as we have done before, or we can use a list comprehension, as shown below.

```
x_cols = [i for i in df.columns if i not in ('ID', 'default_next_month')]
X = df[x_cols]
y = df['default_next_month']
```

The rest of the code used for undersampling is very similar to that used for oversampling. We first split the dataset into training and testing sets.

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=1)
```

Next, we use `imblearn`'s `RandomUnderSampler` module to train the `RandomUnderSampler` instance, then undersample the majority class. Counting the class verifies that both classes are the same size.

```
from imblearn.under_sampling import RandomUnderSampler
ros = RandomUnderSampler(random_state=1)
X_resampled, y_resampled = ros.fit_resample(X_train, y_train)
Counter(y_resampled)
```

A `LogisticRegression` model will be used again on the dataset. We first train it.

```
from sklearn.linear_model import LogisticRegression
model = LogisticRegression(solver='lbfgs', random_state=1)
model.fit(X_resampled, y_resampled)
```

Then, we make predictions and generate a `confusion_matrix`.

```
from sklearn.metrics import confusion_matrix
y_pred = model.predict(X_test)
confusion_matrix(y_test, y_pred)

array([[3732, 2100],
[ 740, 928]])
```

We calculate the `balanced_accuracy_score`, which is 0.598.

```
from sklearn.metrics import balanced_accuracy_score
balanced_accuracy_score(y_test, y_pred)
```

Finally, we print the classification report.

```
from imblearn.metrics import classification_report_imbalanced
print(classification_report_imbalanced(y_test, y_pred))
```

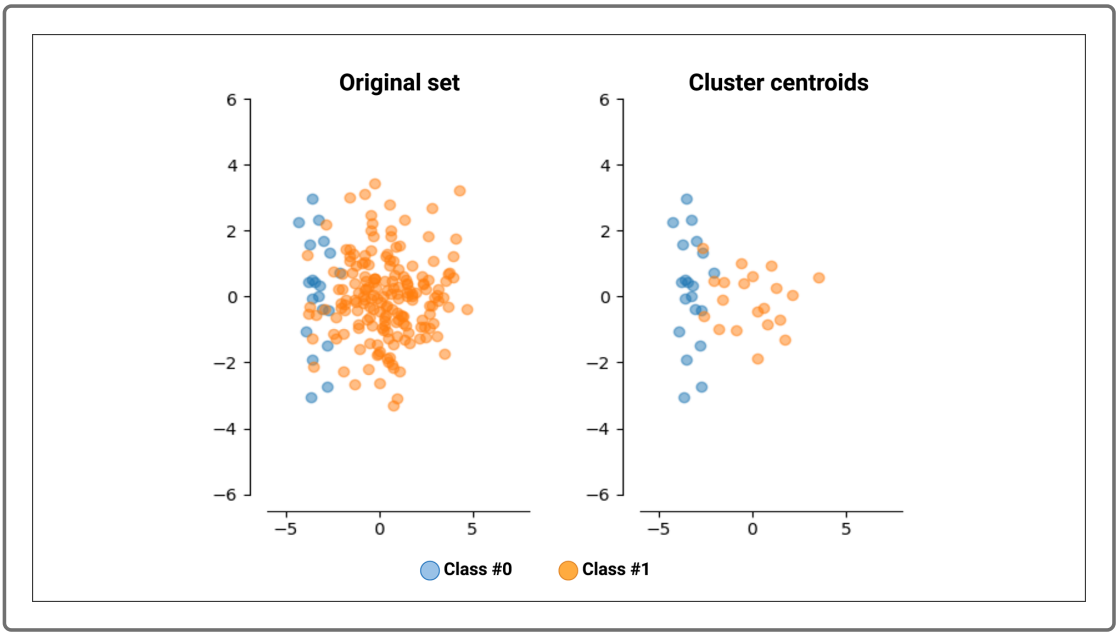
	pre	rec	spe	f1	geo	iba	sup
0	0.83	0.72	0.47	0.77	0.58	0.35	5832
1	0.32	0.47	0.72	0.38	0.58	0.33	1668
avg / total	0.71	0.66	0.53	0.68	0.58	0.34	7500

Generally, the results are unimpressive, especially for predicting defaults. Let's see whether another undersampling technique will improve the metrics.

Cluster Centroid Undersampling

Cluster centroid undersampling is akin to SMOTE. The algorithm identifies clusters of the majority class, then generates synthetic data points, called

centroids, that are representative of the clusters. The majority class is then undersampled down to the size of the minority class.



To implement this technique in Python, we'll use `imblearn`'s `ClusterCentroids` module. The process is much the same as before:

1. Fit and resample the training data.
2. Train a logistic regression model.
3. Create predictions.
4. Assess the results.

NOTE

> Some of these steps are computationally intensive and may take several minutes to complete.

First, instantiate the resampling module and use it to resample the data.

```
from imblearn.under_sampling import ClusterCentroids
cc = ClusterCentroids(random_state=1)
```

```
X_resampled, y_resampled = cc.fit_resample(X_train, y_train)
```

Then instantiate and train a logistic regression model.

```
from sklearn.linear_model import LogisticRegression
model = LogisticRegression(solver='lbfgs', random_state=1)
model.fit(X_resampled, y_resampled)
```

Next, generate the metrics.

```
from sklearn.metrics import confusion_matrix
y_pred = model.predict(X_test)
confusion_matrix(y_test, y_pred)

from sklearn.metrics import balanced_accuracy_score
balanced_accuracy_score(y_test, y_pred)

from imblearn.metrics import classification_report_imbalanced
print(classification_report_imbalanced(y_test, y_pred))
```

	pre	rec	spe	f1	geo	iba	sup
0	0.82	0.48	0.64	0.60	0.55	0.30	5832
1	0.26	0.64	0.48	0.37	0.55	0.31	1668
avg / total	0.70	0.51	0.61	0.55	0.55	0.30	7500

These results are worse than those from random undersampling! This underscores an important point: While resampling can attempt to address imbalance, it does not guarantee better results.

SKILL DRILL

Perform oversampling on this dataset and compare results. Which resampling technique yields better

results?

17.10.3 Combination Sampling With SMOTEENN

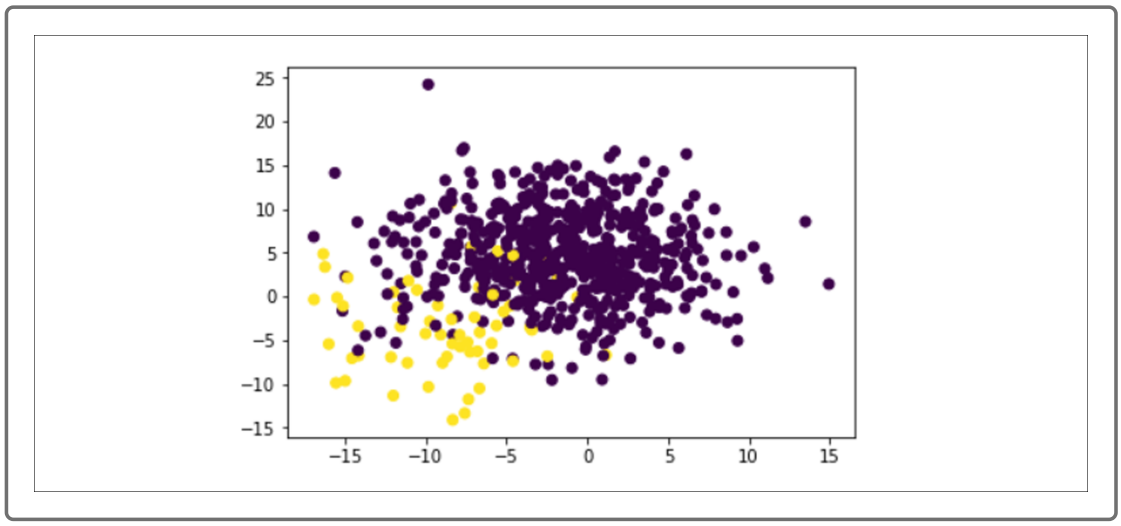
You discuss the results of oversampling and undersampling with Jill. When you point out to her that the improvements seem to be modest, she explains that incremental improvements are usually more realistic than drastic ones. Jill also tells you that such small improvements, in tandem with other tweaks, can add up to make a significant difference. For now, however, she suggests learning about SMOTEENN, an approach to resampling that combines aspects of both oversampling and undersampling.

As previously discussed, a downside of oversampling with SMOTE is its reliance on the immediate neighbors of a data point. Because the algorithm doesn't see the overall distribution of data, the new data points it creates can be heavily influenced by outliers. This can lead to noisy data. With downsampling, the downsides are that it involves loss of data and is not an option when the dataset is small. One way to deal with these challenges is to use a sampling strategy that is a combination of oversampling and undersampling.

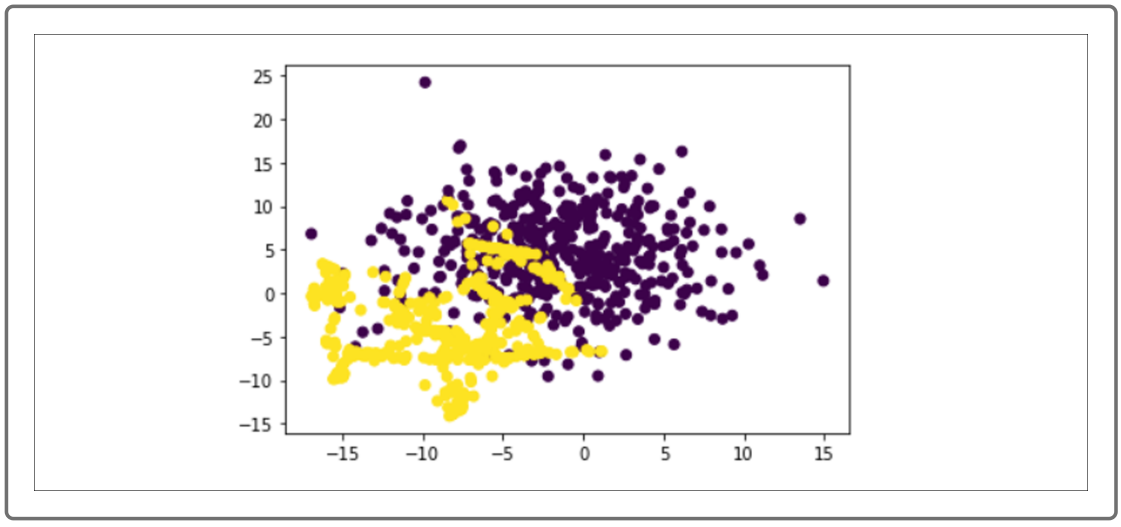
SMOTEENN combines the SMOTE and Edited Nearest Neighbors (ENN) algorithms. SMOTEENN is a two-step process:

1. Oversample the minority class with SMOTE.
2. Clean the resulting data with an undersampling strategy. If the two nearest neighbors of a data point belong to two different classes, that data point is dropped.

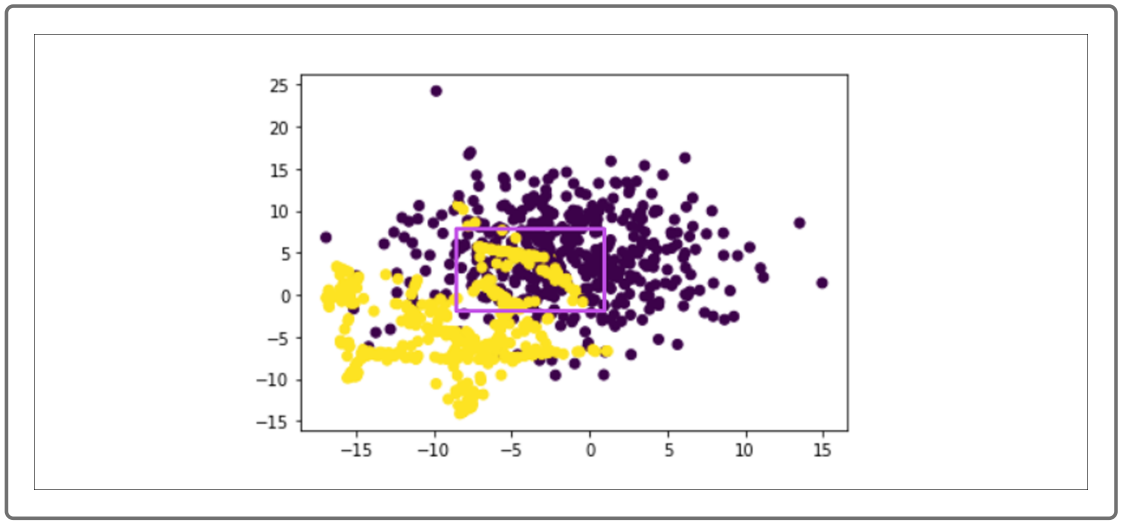
The series of images below help illustrate the SMOTEENN technique. The first image represents a synthetically generated dataset (using the `make_blobs` module) and shows two classes: purple as the majority class and yellow as the minority class.



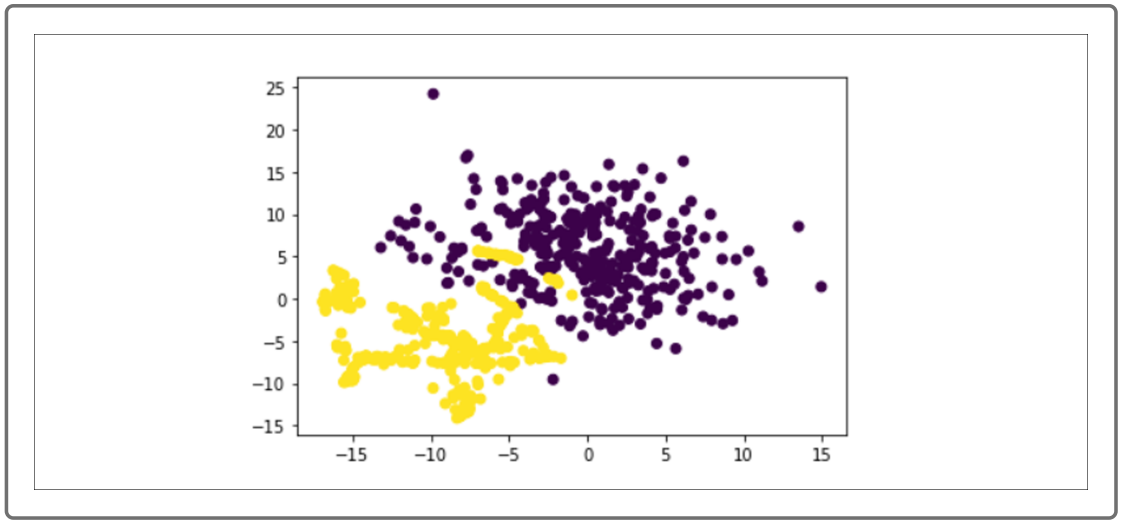
In the following image, the minority class is oversampled with SMOTE.



Note that the two classes significantly overlap, as the box indicates below. This overlap makes classification difficult.



In the next image, SMOTEENN is applied, instead of SMOTE. As with SMOTE, the minority class is oversampled; however, an undersampling step is added, removing some of each class's outliers from the dataset. The result is that the two classes are separated more cleanly.



Let's apply SMOTEENN to the credit card default dataset and compare its results.

[Download 17-10-3-combination_sampling.zip (https://courses.bootcampspot.com/courses/138/files/22532/download?wrap=1)](https://courses.bootcampspot.com/courses/138/files/22532/download?wrap=1)

The code is much the same as before. The only difference is in using the SMOTEENN module.

```
import pandas as pd
from path import Path
from collections import Counter

data = Path('../Resources/cc_default.csv')
df = pd.read_csv(data)
df.head()
```

	ID	ln_balance_limit	sex	education	marriage	age	default_next_month
0	1	9.903488	1	2	0	24	1
1	2	11.695247	1	2	1	26	1
2	3	11.407565	1	2	1	34	0
3	4	10.819778	1	2	0	37	0
4	5	10.819778	0	2	0	57	0

Again, the `ID` and `default_next_month` columns are filtered to create the features dataset. The `default_next_month` column is defined as the target dataset.

```
x_cols = [i for i in df.columns if i not in ('ID', 'default_next_month')]
X = df[x_cols]
y = df['default_next_month']
```

The dataset is split into training and testing sets.

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=1)
```

Next, we import the `SMOTEENN` module and create an instance of `SMOTEENN`, which resamples the dataset.

```
from imblearn.combine import SMOTEENN
smote_enn = SMOTEENN(random_state=0)
X_resampled, y_resampled = smote_enn.fit_resample(X, y)
```

Again, we use a `LogisticRegression` model to generate predictions.

```
from sklearn.linear_model import LogisticRegression
model = LogisticRegression(solver='lbfgs', random_state=1)
model.fit(X_resampled, y_resampled)
```

As before, the evaluation metrics are generated. First, the `confusion_matrix` is generated.

```
from sklearn.metrics import confusion_matrix
y_pred = model.predict(X_test)
confusion_matrix(y_test, y_pred)
```

Next, the `balanced_accuracy_score` is generated.

```
from sklearn.metrics import balanced_accuracy_score
balanced_accuracy_score(y_test, y_pred)
```

Finally, we print the classification report.

```
from imblearn.metrics import classification_report_imbalanced
print(classification_report_imbalanced(y_test, y_pred))
```

	pre	rec	spe	f1	geo	iba	sup
0	0.83	0.64	0.56	0.72	0.60	0.36	5832
1	0.31	0.56	0.64	0.40	0.60	0.35	1668
avg / total	0.72	0.62	0.57	0.65	0.60	0.36	7500

Resampling with SMOTEENN did not work miracles, but some of the metrics show an improvement over undersampling.

Which of the following best describes the difference between SMOTE and SMOTEENN?

- SMOTE generates synthetic observations, whereas SMOTEENN does not.
- SMOTE involves only oversampling, whereas SMOTEENN involves only undersampling.
- SMOTE does not involve undersampling.

Check Answer

Finish ▸

Module 17 Challenge

Submit Assignment

Due May 10 by 11:59pm **Points** 100 **Submitting** a text entry box or a website url

> Jill commends you for all your hard work. Piece by piece, you have been building up your skills in data preparation, statistical reasoning, and machine learning.
>
> You are now ready to apply machine learning to solve real-world challenges.

In this challenge, you'll build and evaluate several machine learning models to assess credit risk, using data from LendingClub; a peer-to-peer lending services company.

Background

Credit risk is an inherently unbalanced classification problem, as the number of good loans easily outnumber the number of risky loans. Therefore, you'll need to employ different techniques to train and evaluate models with unbalanced classes. Jill asks you to use imbalanced-learn and scikit-learn libraries to build and evaluate models using resampling. Your final task is to evaluate the performance of these models and make a recommendation on whether they should be used to predict credit risk.

Objectives

The goals of this challenge are for you to:

- Implement machine learning models.
- Use resampling to attempt to address class imbalance.
- Evaluate the performance of machine learning models.

Instructions

You'll use the imbalanced-learn library to resample the data and build and evaluate logistic regression classifiers using the resampled data. Download the files you'll need, which include starter code and the dataset:

Download Module -17-Challenge-Resources.zip

You will:

1. Oversample the data using the RandomOverSampler and SMOTE algorithms.
2. Undersample the data using the cluster centroids algorithm.
3. Use a combination approach with the SMOTEENN algorithm.

For each of the above, you'll:

1. Train a logistic regression classifier (from Scikit-learn) using the resampled data.
2. Calculate the balanced accuracy score using `balanced_accuracy_score` from `sklearn.metrics`.
3. Generate a `confusion_matrix`.
4. Print the classification report (`classification_report_imbalanced` from `imblearn.metrics`).

Lastly, you'll write a brief summary and analysis of the models' performance. Describe the precision and recall scores, as well as the balanced accuracy score. Additionally, include a final recommendation on

the model to use, if any. If you do not recommend any of the models, justify your reasoning.

Extension

For the extension, you'll train and compare two different ensemble classifiers to predict loan risk and evaluate each model. Note that you'll use the following modules, which you have not used before. They are very similar to ones you've seen: `BalancedRandomForestClassifier` and `EasyEnsembleClassifier`, both from `imblearn.ensemble`. These modules combine resampling and model training into a single step. Consult the following documentation for more details:

- **Section 5.1.2. Forest of randomized trees (https://imbalanced-learn.readthedocs.io/en/stable/ensemble.html#forest)**
- **imblearn.ensemble.EasyEnsembleClassifier (https://imbalanced-learn.readthedocs.io/en/stable/generated/imblearn.ensemble.EasyEnsembleClassifier.**

Use 100 estimators for both classifiers, and complete the following steps for each model:

1. Train the model and generate predictions.
2. Calculate the balanced accuracy score.
3. Generate a confusion matrix.
4. Print the classification report (`classification_report_imbalanced` from `imblearn.metrics`).
5. For the `BalancedRandomForestClassifier`, print the feature importance, sorted in descending order (from most to least important feature), along with the feature score.

Lastly, you'll write a brief summary and analysis of the models' performance. Describe the precision and recall scores, as well as the balanced accuracy score. Additionally, include a final recommendation on

the model to use, if any. If you do not recommend any of the models, justify your reasoning.

Submission

Host your challenge assignment on GitHub, including:

- Your Jupyter Notebook file(s) with your code and analysis.
- Your dataset.
- A `README.md` file describing your project.

Submit a link to your repository through Canvas.

Note: You are allowed to miss up to two Challenge assignments and still earn your certificate. If you complete all Challenge assignments, your lowest two grades will be dropped. If you wish to skip this assignment, click Submit then indicate you are skipping by typing “I choose to skip this assignment” in the text box.

Rubric

Please **download the detailed rubric** to access the assessment criteria.

Module 17 Rubric

Criteria	Ratings					Pts
Summary and Analysis Please see detailed rubric linked in Challenge description.	**40.0 pts Mastery**	**30.0 pts Approaching Mastery**	**20.0 pts Progressing**	**10.0 pts Emerging**	**0.0 pts Incomplete**	40.0 pts
Oversample the data using the random oversampler and SMOTE algorithms Please see detailed rubric linked in Challenge description.	**20.0 pts Mastery**	**15.0 pts Approaching Mastery**	**10.0 pts Progressing**	**5.0 pts Emerging**	**0.0 pts Incomplete**	20.0 pts
Undersample the data using the cluster centroids algorithm Please see detailed rubric linked in Challenge description.	**20.0 pts Mastery**	**15.0 pts Approaching Mastery**	**10.0 pts Progressing**	**5.0 pts Emerging**	**0.0 pts Incomplete**	20.0 pts
Use a combination approach with the SMOTEENN algorithm Please see detailed rubric linked in Challenge description.	**20.0 pts Mastery**	**15.0 pts Approaching Mastery**	**10.0 pts Progressing**	**5.0 pts Emerging**	**0.0 pts Incomplete**	20.0 pts
						Total Points: 100.0

Module 17
Career Connection

Introduction

Welcome back to another Career Connection! This week you explored machine learning and how algorithms are used in data analytics. You also created training and testing groups from a given dataset and implemented logistic regression. And that's not to mention decision trees, random forests, and support vector machine (SVM) algorithms.

It was a packed week! This material can be tricky, but it's essential to your future career as a data engineer.

NOTE

> Data professionals who work in a machine learning environment are often called machine learning engineers.

There's a lot of buzz around machine learning. After all, it's the core technology behind devices we use every day: smart phones, computers, and smart-home technologies like Alexa, Bixby, and Google Assistant.

If you're interested in a career as a machine learning engineer, there are things you can start doing today to help you get there. Having a solid understanding of this module's material is a good start, but you can also strategically prepare for a technical interview in this field.

Technical Interview Preparation

As we'll be digging deeper into machine learning over the next couple of weeks, we'll begin our technical interview preparation with a few broad, conceptually focused questions.

Answer each of the following questions in two or three sentences. You can use the internet to help guide your answers. Once you've responded to each question, check your answer against the possible answer provided.

Q: What is machine learning?

A: Machine learning is the application of statistics and algorithms to data analysis in order to learn and improve without being explicitly programmed.

Q: What is the difference between supervised and unsupervised machine learning?

A: Supervised machine learning requires labeling of data, while unsupervised learning does not. For example, classification would require that we label the data that we want to use to train the model to classify data.

Q: What is the difference between precision and positive predictive value?

A: There is no difference. The terms "precision" and "positive predictive value" are interchangeable.

Hypothetical Case Study

This week, there is no narrative for the hypothetical case study. Instead, we want you to consider something that could have practical application in your life.

Consider a real-life situation in which the skills you learned this week would apply. Pseudocode a solution for tackling it. You can use the template below or write your own. Complete this assignment in any text editor, or write it in a markdown file that can be uploaded to GitHub.

NOTE

> Pseudocode means to write out in plain language (English or other) the steps you would take in order to solve a specific problem. You do not need to include actual code here; the goal is just to create a kind of roadmap for solving the problem.

Template

Title

Add a short, succinct title that identifies the topic of the narrative.

Problem

Write two or three sentences outlining the problem and why it's important.

Solution

Write two or three sentences outlining the solution. Then write 8–10 steps that you could take in order to solve the problem.

Consider the following example:

> ### NARRATIVE
>
> **Title**: Using unsupervised learning to plan an efficient vacation in London.
>
> **Problem**: We recently announced that we're visiting a new city, and in

> came the flood of suggestions: you should visit x, y, or z; don't forget to eat at this place; the best shops are found in this area; and so on. As we'll only be in London for three days, we want to make the most of our time by visiting places that are near each other. This will cut down on travel time around the city.
>
> **Solution**: It seems like this would be a clustering issue that we can solve with unsupervised machine learning. Here are the first few steps we would need to take:
>
> 1. Gather the geographical data of each of the places we want to visit.
> 2. Pin these to a 2D map. Potentially, Google Map pins could work here.
> 3. Extract the geographical coordinates for each of the places.
> 4. Plot the coordinates on a scatter plot for illustration purposes.

Continue to Hone Your Skills

If you're interested in hearing more about the technical interviewing process and practicing algorithms in a mock interview setting, check out our **upcoming workshops! (https://careerservicesonlineevents.splashthat.com/)**