Certified Artificial Intelligence Practitioner Professional Certificate Lab Instructions

How to Use This Document

This document contains instructions for all labs in the CAIP Specialization. It also includes lab instructions for projects that have a lab component. To access the lab instructions for the particular lab you're working on, navigate to the appropriate course and module, and look for the name of the relevant lab. In the lab instructions, the numbered steps typically give an overview of what you're about to do, whereas the lettered substeps tell you exactly what to do in the lab environment, and provide valuable explanations. Some labs also include discussion questions. You don't need to submit your answers—they're just to get you thinking about the task(s) you're performing. The "Solutions" section at the end of the document gives you some example answers.

Certified Artificial Intelligence Practitioner Professional Certificate Lab Instructions

Course 2: Follow a Machine Learning Workflow

Course Introduction

The following labs are for Course 2: Follow a Machine Learning Workflow.

Modules

The labs in this course pertain to the following modules:

- Module 1: Collect the Dataset
- Module 2: Analyze the Dataset
- Module 3: Prepare the Dataset
- Module 4: Set Up and Train a Model

MODULE 1

Collect the Dataset

The following labs are for Module 1: Collect the Dataset.

LAB 2-1

Examining the Structure of a Machine Learning Dataset

Data File

~/Workflow/housing_data/kc_house_data.csv

Scenario

You are continuing your work on the machine learning tool that CapitalR Real Estate company will use to predict an appropriate sale price for houses. You have proposed a model that will base the price on various attributes such as the size of the home, the number of bathrooms and bedrooms, location, and so forth.

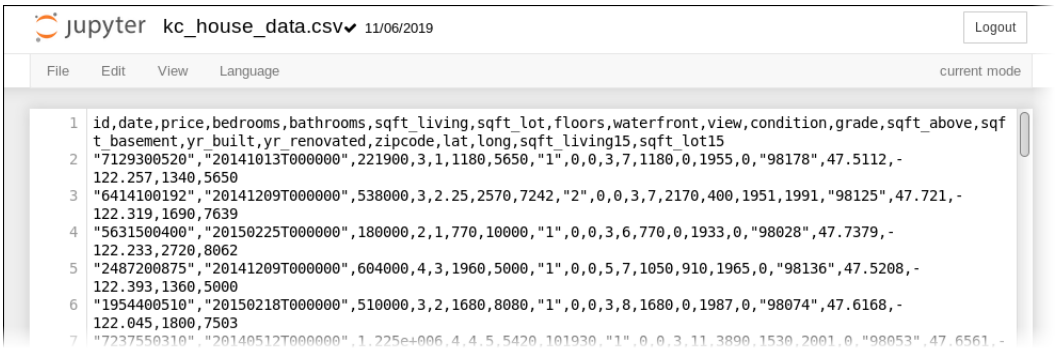
The house's price will be the *output* (the dependent variable). The model will determine (*predict*) the price through multiple *inputs* (independent variables). This seems like an appropriate task for a *regression* model.

You have found a dataset you can use to train the machine learning model. It is a CSV file containing more than 20,000 real estate transactions conducted in King County, Washington. Before you start writing code to load this dataset in Python for analysis and manipulation, you will preview the contents of the data file.

Note: You can obtain a copy of this public domain dataset at **https://www.kaggle.com/harlfoxem/housesalesprediction**.

1. From the Coursera lab page, select **Open Lab** to open a Jupyter Notebook session in another web browser tab.
   - The Jupyter Notebook session shows a listing of directories.
   - You can use this listing to navigate to folders that contain notebooks you want to open.
2. Examine the dataset.
   a) Select **Workflow**.
      The Workflow subdirectory contains a subdirectory named **housing_data** and a notebook file named **Workflow-Housing.ipynb**.
   b) Select **housing_data**.
      The data subdirectory contains one file: **kc_house_data.csv**.

c) Select **kc_house_data.csv**.



- The contents of the file are shown. This is the data you will use to train and test the machine learning model.
- The data is stored in CSV format. Values are separated by commas. Text values, which may include spaces, are enclosed within double quotes.

d) Examine the column labels in the first row.

They include:

- **id**—Unique identifier for each house sold.
- **date**—Date of the house's most recent sale.
- **price**—Price the house most recently sold for.
- **bedrooms**—Number of bedrooms in the house.
- **bathrooms**—Number of bathrooms. A room with a toilet but no shower is counted as 0.5.
- **sqft_living**—Square footage of the house's interior living space.
- **sqft_lot**—Square footage of the lot on which the house is located.
- **floors**—Number of floor levels in the house.
- **waterfront**—Whether the property borders on or contains a body of water. (0 = not waterfront, 1 = waterfront)
- **view**—An index from 0 to 4 representing the subjective quality of the view from the property. The higher the number, the better the view.
- **condition**—An index from 1 to 5 representing the subjective condition of the property. The higher the number, the better the condition.
- **grade**—An index from 0 to 14 representing the quality of the building's construction and design. The higher the number, the better the grade.
- **sqft_above**—The square footage of the interior housing space that is above ground level.
- **sqft_basement**—The square footage of the interior housing space that is below ground level.
- **yr_built**—The year the house was initially built.
- **yr_renovated**—The year of the house's last renovation.
- **zipcode**—What zipcode area the house is located within.
- **lat**—Latitude of the house's location.
- **long**—Longitude of the house's location.
- **sqft_living15**—The square footage of interior housing living space for the nearest 15 neighbors.
- **sqft_lot15**—The square footage of the land lots of the nearest 15 neighbors.

e) Close the **kc_house_data.csv** browser tab.

3. Close the lab browser tab and continue on with the course.

LAB 2-2
Loading the Dataset

Data File
~/Workflow/Workflow-Housing.ipynb

Scenario
You have obtained a dataset in the form of a CSV file containing tens of thousands of real estate transactions made in King County, Washington. Using Python and Jupyter Notebook, you will load this dataset into a pandas DataFrame object, which you can then use to analyze and manipulate the data.

---

1. From the Coursera lab page, select **Open Lab** to open a Jupyter Notebook session in another web browser tab.

2. Run code to import various Python software libraries commonly used in machine learning.
   a) Select **Workflow**.
   b) Select the **Workflow-Housing.ipynb** notebook label.
   The **Workflow-Housing** notebook is opened in Jupyter Notebook.
   c) Examine the Python code beneath the heading **Import software libraries**.
   Scroll, if necessary, to see the entire code listing.

Note: If line numbers are not showing along the left side of the code listing, then select **View→Toggle Line Numbers** to add them. While line numbers are not required to run the code, they will make it easier to refer to the code.

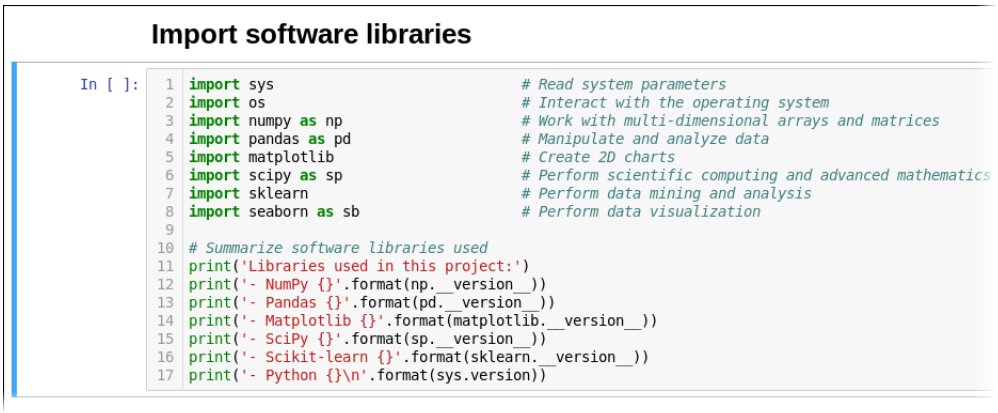
Import software libraries

In [ ]:

```
1  import sys                          # Read system parameters
2  import os                           # Interact with the operating system
3  import numpy as np                  # Work with multi-dimensional arrays and matrices
4  import pandas as pd                 # Manipulate and analyze data
5  import matplotlib                   # Create 2D charts
6  import scipy as sp                  # Perform scientific computing and advanced mathematics
7  import sklearn                      # Perform data mining and analysis
8  import seaborn as sb                # Perform data visualization
9
10 # Summarize software libraries used
11 print('Libraries used in this project:')
12 print('- NumPy {}'.format(np.__version__))
13 print('- Pandas {}'.format(pd.__version__))
14 print('- Matplotlib {}'.format(matplotlib.__version__))
15 print('- SciPy {}'.format(sp.__version__))
16 print('- Scikit-learn {}'.format(sklearn.__version__))
17 print('- Python {}\n'.format(sys.version))
```

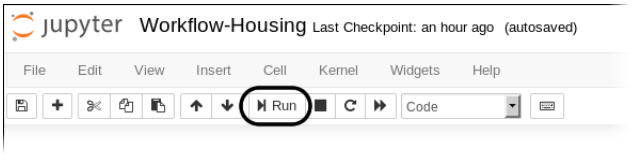
- The `import` statements in lines 1 through 8 enable various software libraries to be used in this program.
- Lines 11 through 17 will print the names of the different libraries, along with the version number of each library installed on the computer.

d) Click within the code listing.



A border is added around the cell that contains the code listing, showing that the cell that contains the code is now selected.

e) Select the **Run** button to run the code in the selected cell.

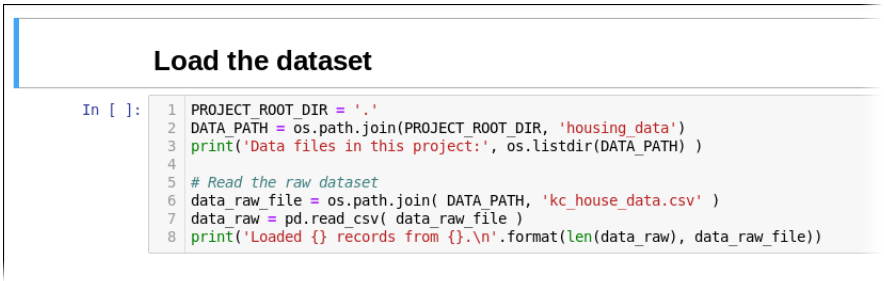


f) Observe the output from the code you just ran.

- Libraries used in the project are listed, along with their version numbers.
- It is important to document which tools and versions you are using, so you can reconstruct the correct environment later, if necessary. Default configurations and settings may also change over time, possibly affecting your results.

3. Load the dataset.

a) Scroll down to view the cell titled **Load the dataset**, and examine the code listing beneath it.



- Lines 1 and 2 identify the location of the directory that contains the dataset.
- Line 3 prints a list of files in the data directory.
- Line 6 constructs a text string that includes the full path and file name of the datafile.
- Line 7 reads data from the datafile into a pandas DataFrame named `data_raw`. Once this statement executes, the dataset will be in memory, where it can be read directly from the `data_raw` variable.
- Line 8 gets the length of (number of rows/records in) `data_raw`, and displays that number in a message.

b) Click within the code listing, and select **Run**.

c) Observe the output.

```
Data files in this project: ['kc_house_data.csv']
Loaded 21613 records from ./housing_data/kc_house_data.csv.
```

- 21,613 records have been loaded from the **kc_house_data.csv** file.
- These records are now loaded in the `data_raw` DataFrame object, from which they can be displayed or manipulated through Python code.

4. Save and close the lab.

a) From the menu, select **File→Save and Checkpoint**.
b) Close the lab browser tab and continue on with the course.

MODULE 2

Analyze the Dataset

The following labs are for Module 2: Analyze the Dataset.

LAB 2-3
Exploring the General Structure of the Dataset

Data File

~/Workflow/Workflow-Housing.ipynb

Scenario

Now that you have loaded the real estate dataset, you can use Python and various software libraries installed in your Python environment to get acquainted with the data. It's helpful to know what kind of information is present in each column and the data types stored in those columns. This will help you to start thinking about which columns will be useful to train your machine learning model, and whether you need to perform any sort of "cleanup" tasks before you set up the model.

---

1. Open the lab and return to where you were in the notebook.
   a) From the Coursera lab page, select **Open Lab** to open a Jupyter Notebook session in another web browser tab.
   b) Open **Workflow/Workflow-Housing.ipynb**.
   c) Your progress from the previous lab should have been saved.

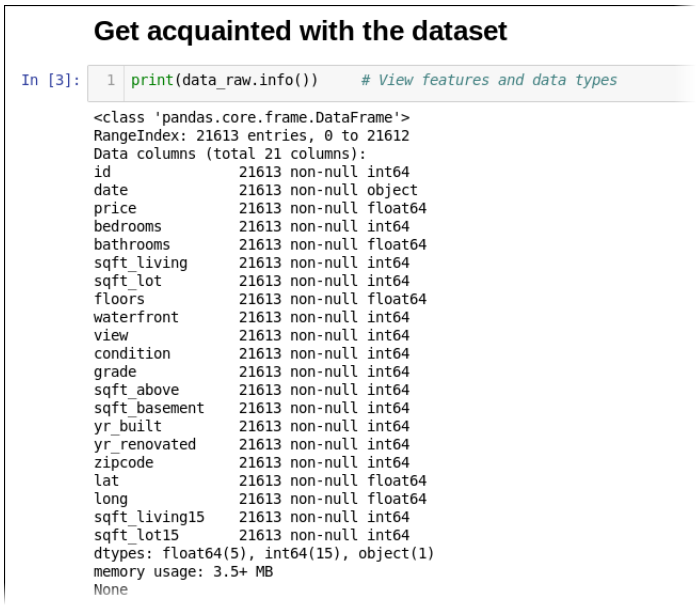
Note: If you encounter errors when running the next code blocks, select the **Get acquainted with the dataset** heading, then select **Cell→Run All Above**.

2. Examine the data types recorded in the dataset.
   a) Scroll down to view the cell titled **Get acquainted with the dataset**, and examine the code listing beneath it.

   Line 1 will output information about the various data types included in the dataset.
   b) Select the cell that contains the code listing, and select **Run**.

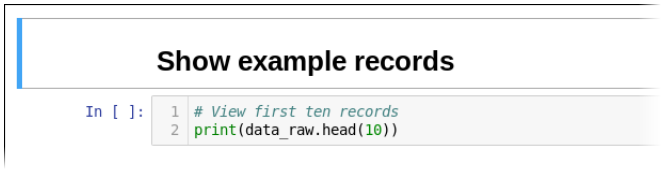
c) Observe the information about the data types used in this dataset.

It's unusual for a large dataset to have no missing values.



- 21,613 records ("entries" regarding a particular house) are in the dataset.
- Each column in the dataset is listed, along with its data type and the number of records that include a data value.
  - Five columns contain floating point number values: `price`, `bathrooms`, `floors`, `lat`, and `long`.
  - Fifteen columns contain integer number values: `id`, `bedrooms`, `sqft_living`, `sqft_lot`, `waterfront`, `view`, `condition`, `grade`, `sqft_above`, `sqft_basement`, `yr_built`, `yr_renovated`, `zipcode`, `sqft_living15`, and `sqft_lot15`.
  - One column (`date`) contains a date value (reported as an "object" value).
- There are no missing data values. Each column contains 21,613 entries.

d) Scroll down to view the cell titled **Show example records**, and examine the code listing beneath it.



This call to the DataFrame's `head()` function will display the first ten rows of data.

e) Select the cell that contains the code listing, and select **Run**.

f) Examine the first ten records in the dataset.

```
           id             date      price  bedrooms  bathrooms  sqft_living  \
0  7129300520  20141013T000000   221900.0         3       1.00         1180
1  6414100192  20141209T000000   538000.0         3       2.25         2570
2  5631500400  20150225T000000   180000.0         2       1.00          770
3  2487200875  20141209T000000   604000.0         4       3.00         1960
4  1954400510  20150218T000000   510000.0         3       2.00         1680
5  7237550310  20140512T000000  1225000.0         4       4.50         5420
6  1321400060  20140627T000000   257500.0         3       2.25         1715
7  2008000270  20150115T000000   291850.0         3       1.50         1060
8  2414600126  20150415T000000   229500.0         3       1.00         1780
9  3793500160  20150312T000000   323000.0         3       2.50         1890

   sqft_lot  floors  waterfront  view  ...  grade  sqft_above  sqft_basement  \
0      5650     1.0           0     0  ...      7        1180              0
1      7242     2.0           0     0  ...      7        2170            400
2     10000     1.0           0     0  ...      6         770              0
3      5000     1.0           0     0  ...      7        1050            910
4      8080     1.0           0     0  ...      8        1680              0
5    101930     1.0           0     0  ...     11        3890           1530
6      6819     2.0           0     0  ...      7        1715              0
7      9711     1.0           0     0  ...      7        1060              0
8      7470     1.0           0     0  ...      7        1050            730
9      6560     2.0           0     0  ...      7        1890              0

   yr_built  yr_renovated  zipcode      lat     long  sqft_living15  \
0      1955             0    98178  47.5112 -122.257           1340
1      1951          1991    98125  47.7210 -122.319           1690
2      1933             0    98028  47.7379 -122.233           2720
3      1965             0    98136  47.5208 -122.393           1360
4      1987             0    98074  47.6168 -122.045           1800
5      2001             0    98053  47.6561 -122.005           4760
6      1995             0    98003  47.3097 -122.327           2238
7      1963             0    98198  47.4095 -122.315           1650
8      1960             0    98146  47.5123 -122.337           1780
9      2003             0    98038  47.3684 -122.031           2390

   sqft_lot15
0        5650
1        7639
2        8062
3        5000
4        7503
5      101930
6        6819
7        9711
8        8113
9        7570

[10 rows x 21 columns]
```

- Because all of the columns can't fit within the page width, they wrap around to display in four chunks. But each listing includes row numbers to help you associate them with each other.
- Even in this small sample, you can see significant variation in prices, number of bedrooms and bathrooms, living space, and so forth.

3. Which attributes do you think might have an influence on price?

A: Some attributes might seem important from a commonsense perspective—such as the lot size (`sqft_lot`), size of the living space, and whether the property is waterfront or has a view. Others such as location (`zipcode`, `lat`, and `long`) might be significant if they correspond to expensive neighborhoods. Other attributes might have a surprising influence on the price. Performing some statistical analysis will help to reveal which values actually correlate with price.

4. Save and close the lab.

a) From the menu, select **File→Save and Checkpoint**.

b) Close the lab browser tab and continue on with the course.

LAB 2-4
Analyzing a Dataset Using Statistical Measures

Data File
~/Workflow/Workflow-Housing.ipynb

Scenario
You have explored the general structure of the dataset and have gotten familiar with the various columns of data, including the data type of each column. Now you will examine various statistical measures as you continue considering which features may be useful to predict a house's optimum price.

---

1. Open the lab and return to where you were in the notebook.
   a) From the Coursera lab page, select **Open Lab** to open a Jupyter Notebook session in another web browser tab.
   b) Open **Workflow/Workflow-Housing.ipynb**.
   c) Your progress from the previous lab should have been saved.

Note: If you encounter errors when running the next code blocks, select the **Examine descriptive statistics** heading, then select **Cell→Run All Above**.

2. View descriptive statistics for the dataset.
   a) Scroll down to view the cell titled **Examine descriptive statistics**, and examine the code listing beneath it.

These two lines of code work together to output a statistical description of the data contained within `data_raw`.

- Line 2 outputs a statistical description of the data contained within the `data_raw` dataframe.
- Line 1 sets the context for the statement in line two, specifying that floating point numbers should be displayed with two decimal places.

   b) Select the cell that contains the code listing, and select **Run**.

c) Examine the statistics describing the dataset.

```
                 id      price  bedrooms  bathrooms  sqft_living   sqft_lot  \
count      21613.00   21613.00  21613.00   21613.00     21613.00   21613.00
mean  4580301520.86  540088.14      3.37       2.11      2079.90   15106.97
std   2876565571.31  367127.20      0.93       0.77       918.44   41420.51
min      1000102.00   75000.00      0.00       0.00       290.00     520.00
25%   2123049194.00  321950.00      3.00       1.75      1427.00    5040.00
50%   3904930410.00  450000.00      3.00       2.25      1910.00    7618.00
75%   7308900445.00  645000.00      4.00       2.50      2550.00   10688.00
max   9900000190.00 7700000.00     33.00       8.00     13540.00 1651359.00

        floors  waterfront     view  condition    grade  sqft_above  \
count 21613.00    21613.00 21613.00   21613.00 21613.00    21613.00
mean      1.49        0.01     0.23       3.41     7.66     1788.39
std       0.54        0.09     0.77       0.65     1.18      828.09
min       1.00        0.00     0.00       1.00     1.00      290.00
25%       1.00        0.00     0.00       3.00     7.00     1190.00
50%       1.50        0.00     0.00       3.00     7.00     1560.00
75%       2.00        0.00     0.00       4.00     8.00     2210.00
max       3.50        1.00     4.00       5.00    13.00     9410.00

       sqft_basement  yr_built  yr_renovated  zipcode      lat     long  \
count       21613.00  21613.00      21613.00 21613.00 21613.00 21613.00
mean          291.51   1971.01         84.40 98077.94    47.56  -122.21
std           442.58     29.37        401.68    53.51     0.14     0.14
min             0.00   1900.00          0.00 98001.00    47.16  -122.52
25%             0.00   1951.00          0.00 98033.00    47.47  -122.33
50%             0.00   1975.00          0.00 98065.00    47.57  -122.23
75%           560.00   1997.00          0.00 98118.00    47.68  -122.12
max          4820.00   2015.00       2015.00 98199.00    47.78  -121.31

       sqft_living15  sqft_lot15
count       21613.00    21613.00
mean         1986.55    12768.46
std           685.39    27304.18
min           399.00      651.00
25%          1490.00     5100.00
50%          1840.00     7620.00
75%          2360.00    10083.00
max          6210.00   871200.00
```

The average (mean) home in this dataset has a price of $540,088.14, 3.37 bedrooms, 2.11 bathrooms, 2,079 square feet of living space, and 1.49 floors.

3. Summarize the mode for non-continuous or categorical data values.

a) Scroll down to view the cell titled **Summarize the most common values**, and examine the code listing beneath it.

Summarize the most common values

```
In [ ]:
1 # Summarize most common values for features with non-continuous or categorical values
2 features_to_summarize = ['view','waterfront','grade','zipcode','bedrooms','bathrooms','floors']
3 data_raw[features_to_summarize].mode()
```

This code displays the mode for selected features.

- Line 1 specifies which features should be summarized.
- Line 2 outputs the mode (most common data value) for each of the specified features in the dataframe.

b) Select the cell that contains the code listing, and select **Run**.

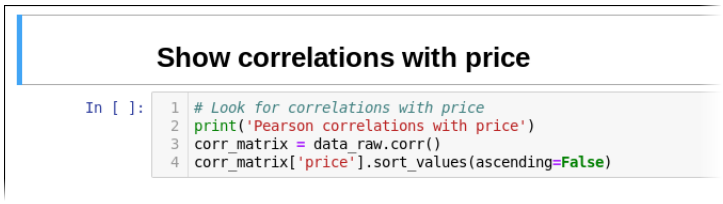
c) Examine the mode values displayed for the various features.

	view	waterfront	grade	zipcode	bedrooms	bathrooms	floors
0	0	0	7	98103	3	2.5	1.0

The typical house:

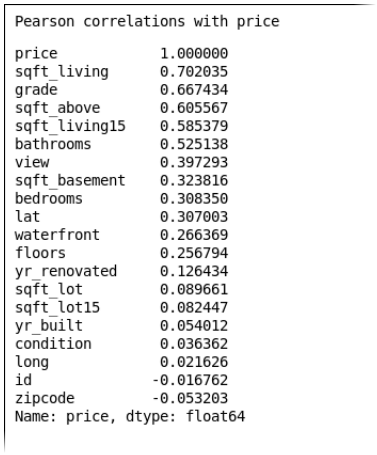
- Does not have a "view" and is not on the waterfront.
- Has a grade of 7.
- Has a zipcode of 98103.
- Has 3 bedrooms, 2.5 bathrooms, and 1 floor level.

4. Look for values that seem to correlate with price.
   a) Scroll down to view the cell titled **Show correlations with price**, and examine the code listing beneath it.



   - Line 2 uses the pandas software library to find the standard (Pearson) correlation coefficient between each feature in the dataset.
   - Line 3 outputs the correlations for the price feature, sorting to show the results in order by how closely they correlate with price. Items that correlate with price most closely will have the highest values, and will appear at the top of the list.

   b) Select the cell that contains the code listing, and select **Run**.
   c) Examine how the various features correlate with `price`.



   - The feature that has the strongest correlation with `price` is `sqft_living`.
   - Interestingly, the reported `condition` of the house does not have a strong correlation with `price`, nor does lot size (`sqft_lot15` and `sqft_lot`) or `waterfront`.
   - You can disregard the perfect 1.000000 standard correlation value for `price`. (Of course, it correlates perfectly with itself.)

5. Save and close the lab.
   a) From the menu, select **File→Save and Checkpoint**.
   b) Close the lab browser tab and continue on with the course.

LAB 2-5
Analyzing a Dataset Using Visualizations

Data File

~/Workflow/Workflow-Housing.ipynb

Scenario

You have used various statistical measures to examine values in the dataset. Now you will use visualizations (various types of charts) to gain additional insights regarding the data.

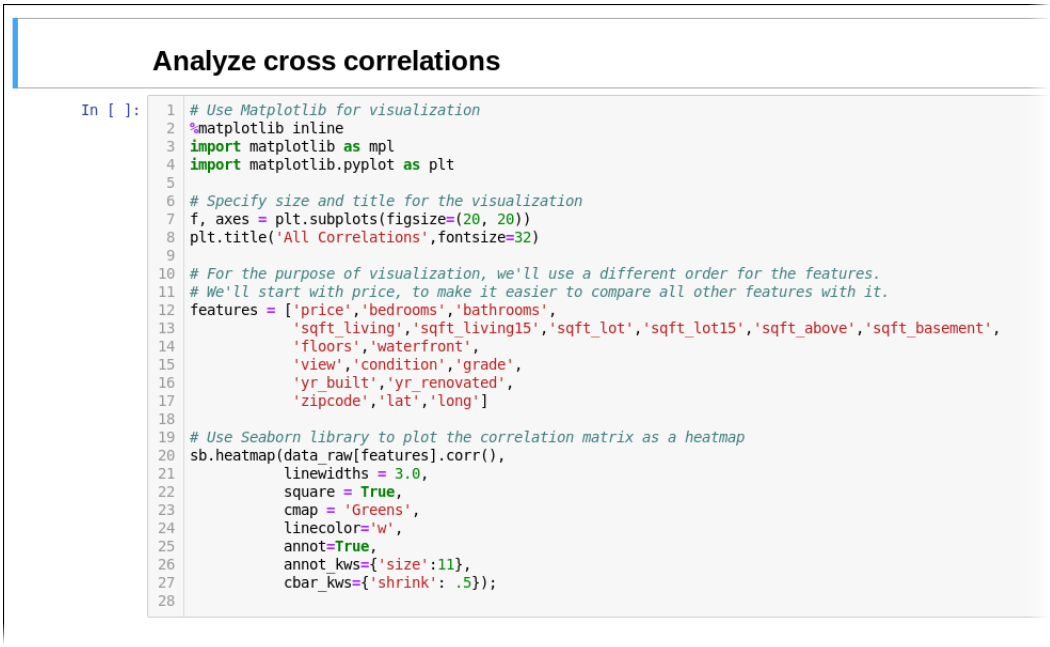
---

1. Open the lab and return to where you were in the notebook.
   a) From the Coursera lab page, select **Open Lab** to open a Jupyter Notebook session in another web browser tab.
   b) Open **Workflow/Workflow-Housing.ipynb**.
   c) Your progress from the previous lab should have been saved.

Note: If you encounter errors when running the next code blocks, select the **Analyze cross correlations** heading, then select **Cell→Run All Above**.

2. Analyze cross correlations.

a) Scroll down to view the cell titled **Analyze cross correlations**, and examine the code listing beneath it.



Analyze cross correlations

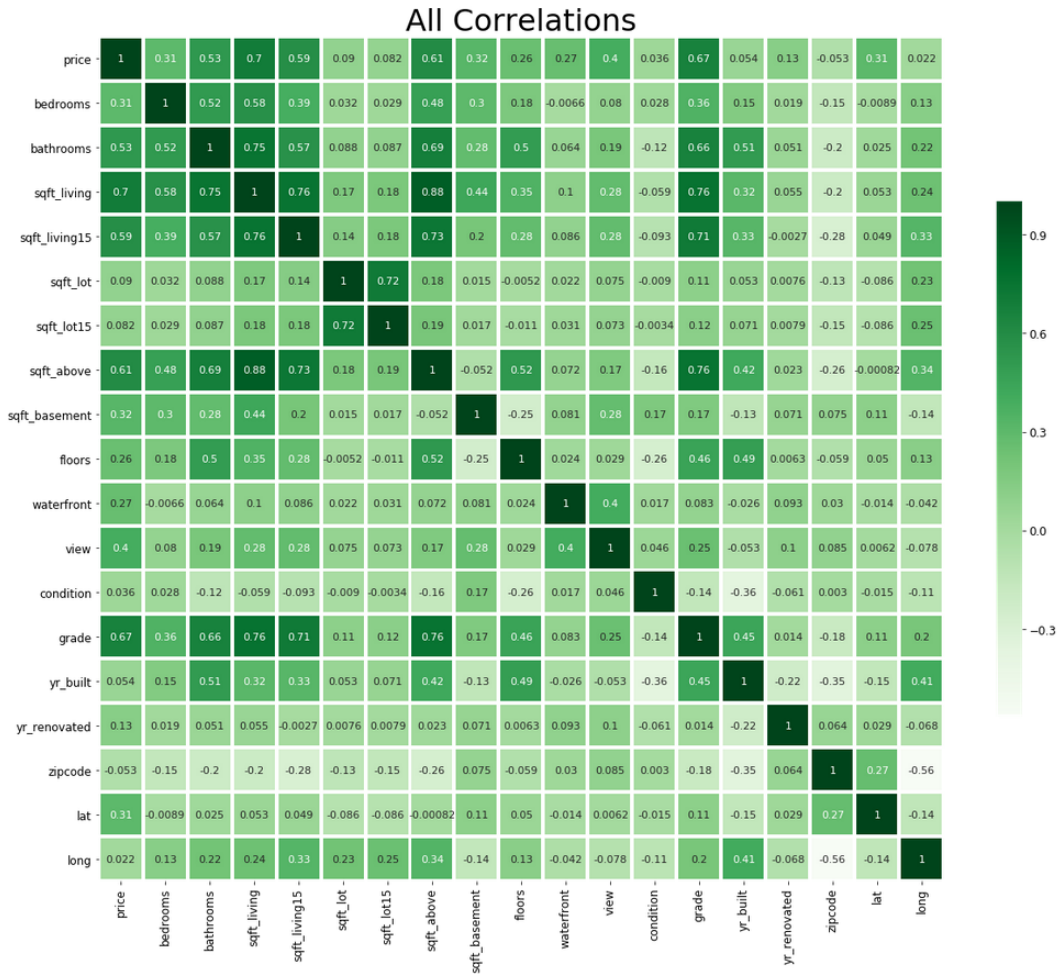
In [ ]:

```
1  # Use Matplotlib for visualization
2  %matplotlib inline
3  import matplotlib as mpl
4  import matplotlib.pyplot as plt
5
6  # Specify size and title for the visualization
7  f, axes = plt.subplots(figsize=(20, 20))
8  plt.title('All Correlations',fontsize=32)
9
10 # For the purpose of visualization, we'll use a different order for the features.
11 # We'll start with price, to make it easier to compare all other features with it.
12 features = ['price','bedrooms','bathrooms',
13             'sqft_living','sqft_living15','sqft_lot','sqft_lot15','sqft_above','sqft_basement',
14             'floors','waterfront',
15             'view','condition','grade',
16             'yr_built','yr_renovated',
17             'zipcode','lat','long']
18
19 # Use Seaborn library to plot the correlation matrix as a heatmap
20 sb.heatmap(data_raw[features].corr(),
21            linewidths = 3.0,
22            square = True,
23            cmap = 'Greens',
24            linecolor='w',
25            annot=True,
26            annot_kws={'size':11},
27            cbar_kws={'shrink': .5});
28
```

- Lines 7 and 8 prepare to plot the visualization, specifying its size and title.
- The statement in lines 12 through 17 identifies the columns that will be cross correlated and the order in which they will appear.
- The statement in lines 20 through 27 calls the Seaborn library's `heatmap()` function, passing in the correlation values to be displayed in the heatmap.

b) Select the cell that contains the code listing, and select **Run**.

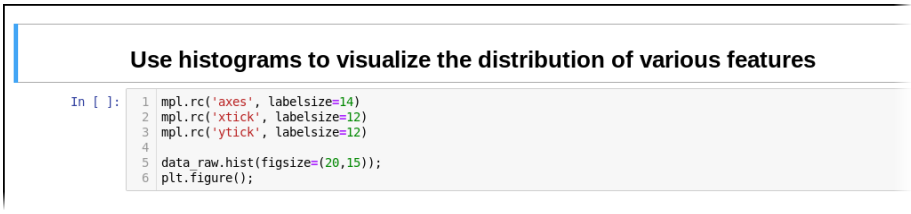
c) Examine how the various features correlate with each other.



- This heatmap visualization shows correlations between different features in the dataset as numeric values, but enhances them with color coding that helps you quickly see which values correlate the most.
- Each feature is shown on the x-axis and y-axis.
- At each intersection, the correlation coefficient is shown for the combination of features represented on the two axes.
- Darker tones highlight values with a high correlation coefficient. The darkest values appear diagonally where features intersect with themselves.

3. Show how the various features are distributed.

a) Scroll down to view the cell titled **Use histograms to visualize the distribution of various features**, and examine the code listing beneath it.

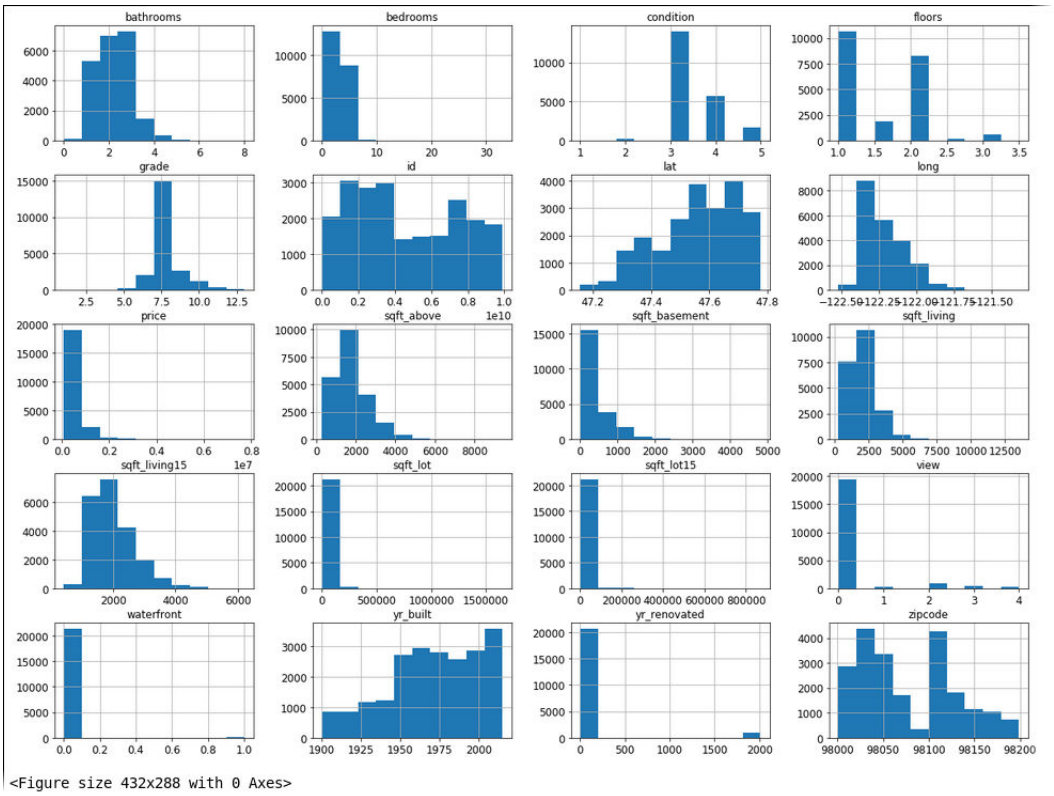


- Lines 1 through 3 specify the font size to be used for the histograms' axis titles and tick mark labels.
- Line 5 calls the pandas DataFrame object's `hist()` method to create a histogram for each column of numeric data in the dataset.
- Line 6 plots the entire figure using the settings configured in lines 1 through 3.

b) Select the cell that contains the code listing, and select **Run**.

Note: It may take several seconds for processing of the code to finish.

c) Examine the distributions shown in the figures.



- Some features, such as `bathrooms`, `bedrooms`, and `sqft_living` are fairly close to a normal distribution. Others, such as `zipcode` are not.
- The `yr_built` (year built) values likely follow the general pattern for house building over the years. More houses have been built in recent years, and there was a decline in building during the Great Depression years (late 1930s).
- The central tendency can be observed for several values, such as `condition` (3), `bathrooms` (2.5), `grade` (7), and `floors` (1).
- The x-axis of the `bedrooms` histogram extends out to more than 30. Charts typically adjust the x- and y-axes to show the range of values being plotted, so there may be some outlier data values in the `bedroom` column that you should investigate. Something similar is happening with the `price` histogram.

4. Use a map visualization to gain insights regarding the relationship between price and location.

a) Scroll down to view the cell titled **Visualize with a geographic map to gain insights regarding location**, and observe the code listing beneath it.

Visualize with a geographic map to gain insights regarding location

```
# To avoid overwhelming the visualization tool, we'll only plot every Nth house.
n_homes = 20
data_raw_subset = data_raw.sort_values(by =['price'], ascending = False)[::n_homes]

# Output highest house price
max_price = data_raw_subset.loc[data_raw_subset['price'].idxmax()]['price']
print(f'The highest home price in this dataset is ${max_price:,.0f}')

# Descriptions of the building grades used in King County
```

This is a large chunk of code. It will be easier to understand what the code does if you first run it see the map it produces.

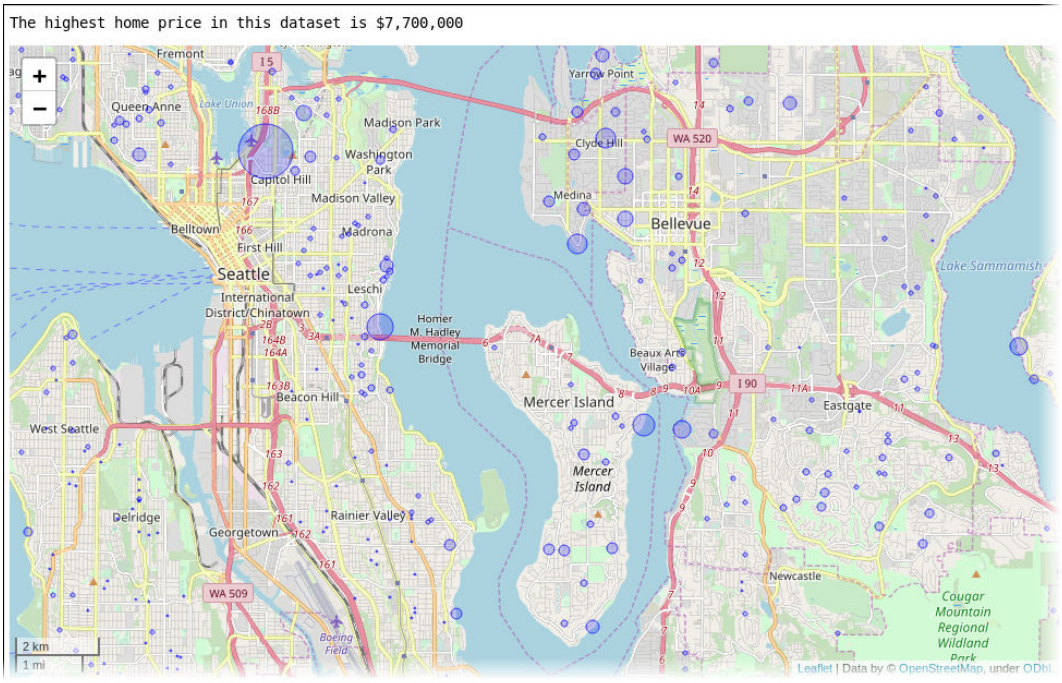
b) Select the cell that contains the code listing, and select **Run**.

Plotting the map may take several seconds to complete.

c) In the upper-left corner of the map, select the **+** button to zoom in. After the map is redrawn, select the **+** button again to zoom in further.

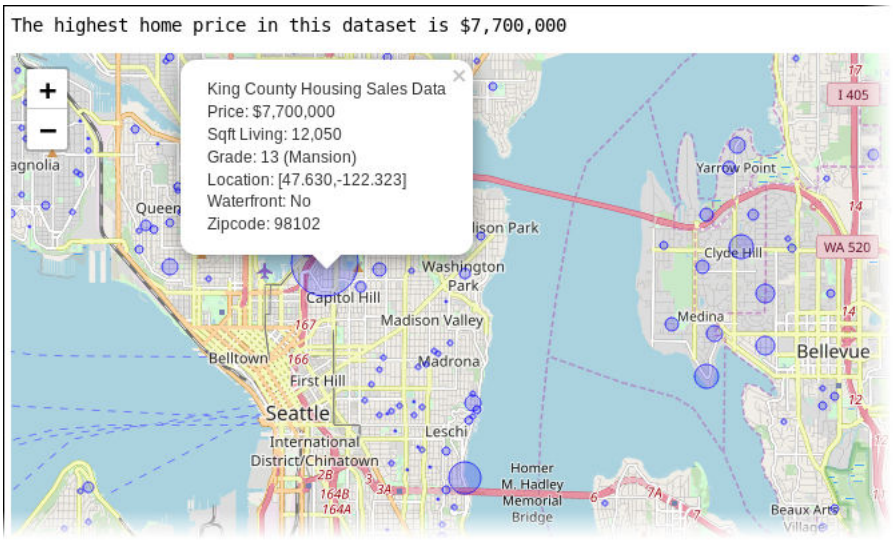
The ability to zoom in and out can be useful as you analyze the map to look for patterns of pricing.

d) Drag the map so Seattle and Bellevue are centered with Mercer Island in the middle, as shown. Observe the map.



- Above the map is a message that shows the highest house price.
- The map shows houses in King County, Washington.
- Each dot on the map represents a house from the dataset. The size of the dot corresponds to the price of the house.

e) Select the largest dot, which is located west of Washington Park and north of Capitol Hill.



- When you select the dot, popup text shows some of the relevant data for this house.
- This is the $7,700,000 house identified in the message above the map.

f) Refer to the map as you explore the relationship between location and housing prices.

	Note: Drag to adjust the location and zoom in and out as needed.

5. What patterns seem to exist regarding house prices and location?

A: There seems to be a cluster of expensive homes located around the lakes in Seattle and Bellevue (Lake Washington and Lake Sammamish). But there are also some expensive homes away from the lakes. Less expensive homes seem to be located in areas away from Seattle and Bellevue, such as the southern part of the county (Federal Way, Auburn, Covington, Maple Valley, and Enumclaw).

6. Examine the code that produced the map visualization.

a) Scroll up and view lines 2 through 7 of the code that produced the map.

Visualize with a geographic map to gain insights regarding location

```
1  # To avoid overwhelming the visualization tool, we'll only plot every Nth house.
2  n_homes = 20
3  data_raw_subset = data_raw.sort_values(by =['price'], ascending = False)[::n_homes]
4
5  # Output highest house price
6  max_price = data_raw_subset.loc[data_raw_subset['price'].idxmax()]['price']
7  print(f'The highest home price in this dataset is ${max_price:,.0f}')
8
9  # Descriptions of the building grades used in King County
```

- Lines 2 and 3 create a temporary copy of the dataset that sorts the homes by price, but filters the records to only include every 20th house in the dataset. While this sample should still be adequate to visually depict any price trends based on location, it also ensures that the mapping tool won't be overwhelmed with data.
- Lines 6 and 7 output the message identifying the highest house price.

Note: If you wanted to see more houses, you could reduce the number assigned in line 2. Bear in mind, however, that including more houses will increase the time needed to plot the map, and may bog down the entire Jupyter Notebook application.

b) View lines 11 through 13 of the code that produced the map.

```
9  # Descriptions of the building grades used in King County
10 # Values obtained from http://www5.kingcounty.gov/sdc/FGDCDocs/resbldg_extr_faq.htm
11 bldg_grades = ['Unknown','Cabin','Substandard','Poor','Low','Fair',
12                'Low Average','Average','Good','Better',
13                'Very Good','Excellent','Luxury','Mansion','Exceptional Properties']
```

- Lines 11 through 13 provide a lookup for the numeric building grades in the dataset. This information came from the King County website. In line 29, `bldg_grades` is used to generate the popup text.
- In the popup text, it is more meaningful to show the description than to just show the number. For example, the most expensive house's popup text showed that it was grade 13, which is described as a "Mansion".

c) View the remaining lines of code.

```
15 # Use Folium library to plot values on a map.
16 import folium
17
18 # Generate the base map, centering on King County.
19 base_map = folium.Map(location = [47.5300, -122.2000],
20                       control_scale = True,
21                       max_zoom = 20,
22                       zoom_start = 10,
23                       zoom_control = True)
24
25 # Plot homes by price.
26 for index, row in data_raw_subset.iterrows():
27
28     # Get the grade description for this row.
29     grade_desc = bldg_grades[row['grade']]
30     waterfront_desc = "Yes" if (row['waterfront'] == 1) else "No"
31
32     # Add popup text. Click each point to show details.
33     popup_text = '<br>'.join(['King County Housing Sales Data',
34                               'Price: ${:,.0f}',
35                               'Sqft Living: {:,.0f}',
36                               'Grade: {} ({})',
37                               'Location: [{:.3f},{:.3f}]',
38                               'Waterfront: {}',
39                               'Zipcode: {}'])
40
41     popup_text = popup_text.format(row['price'],
42                                    row['sqft_living'],
43                                    row['grade'], grade_desc,
44                                    row['lat'], row['long'],
45                                    waterfront_desc,
46                                    row['zipcode'])
47
48     # Add each home to the map, but show larger dots for higher prices.
49     scaling_value = (row['price'] / max_price)      # 1.0 for highest price.
50     folium.CircleMarker([row['lat'], row['long']],
51                         radius = 25 * scaling_value,
52                         weight = 1,
53                         fill = True,
54                         fillColor = '#0000FF',
55                         fillOpacity = 0.7,
56                         color = '#0000FF',
57                         opacity = 0.7,
58                         popup = popup_text).add_to(base_map)
59
60 base_map
```

- Line 16 imports the Folium software library, which is used to generate the map.
- Lines 19 through 23 create a `Map` object, which will be referred to as `base_map`. This will generate the street map, but subsequent code will have to add markers for each house.
- Lines 26 through 58 provide a loop that iterates through each row (house) in the dataset, constructing the popup text (lines 29 through 46), and creating a scaled dot (lines 49 through 58) for each house.
- Up to this point, the map exists in memory, but is not yet plotted on the page. Line 60 outputs the map.

7. Save and close the lab.

a) From the menu, select **File→Save and Checkpoint**.

b) Close the lab browser tab and continue on with the course.

MODULE 3

Prepare the Dataset

The following lab is for Module 3: Prepare the Dataset.

LAB 2-6

Splitting the Training and Testing Datasets and Labels

Data File

~/Workflow/Workflow-Housing.ipynb

Scenario

The housing dataset is already in pretty good shape. You have already observed that all rows contain data, so no missing values need to be added. But you have identified some attributes that you won't use to train the model.

Features You Won't Use	*Rationale*
`id` and `date`	The `id` and `date` values just identify the record and the date the house was sold. They have no bearing on `price`.
`sqft_above`	The `sqft_living` and `sqft_above` features showed a high correlation with each other (0.88), so it may be redundant to use both of these features in the model. The `sqft_living` feature showed a 0.70 correlation with `price`, whereas `sqft_above` showed only a 0.61 correlation with `price`. So of these two features, `sqft_living` will be the more useful one to predict the price.
`sqft_living15` and `sqft_lot15`	These features summarize the size of the house and lot for the 15 nearest properties. They don't directly describe the property itself. While this might provide some indication of the neighborhood, the house's own measurements (`sqft_living` and `sqft_lot`) show a stronger correlation with `price` than these measures of the house's neighbors.

Attributes that remain include `sqft_living`, `grade`, `bathrooms`, `view`, `sqft_basement`, `bedrooms`, `lat`, `waterfront`, `floors`, `yr_renovated`, `sqft_lot`, `yr_built`, `condition`, `long`, and `zipcode`. Later you can also drop any of these attributes if you decide not to use them.

Before you get any further into the project, you should split your testing data from your training data. You also need to separate the house prices from the dataset. Since price is the output the model should provide (the dependent variable), it should not be included in the training or testing dataset. However, you'll need to keep a copy of the labels so you can verify your results.

---

1. Open the lab and return to where you were in the notebook.
   a) From the Coursera lab page, select **Open Lab** to open a Jupyter Notebook session in another web browser tab.
   b) Open **Workflow/Workflow-Housing.ipynb**.
   c) Your progress from the previous lab should have been saved.

Note: If you encounter errors when running the next code blocks, select the **Split the data into training and testing sets and labels** heading, then select **Cell→Run All Above**.

2. Split the data into training and testing sets and labels.

a) Scroll down to view the cell titled **Split the data into training and testing sets and labels**, and examine the code listing beneath it.

Split the data into training and testing sets and labels

```
In [11]:
1  from sklearn.model_selection import train_test_split
2
3  # Price is the dependent variable (value to be predicted), so it will be
4  # removed from the training data and put into a separate dataframe for labels.
5
6  label_columns = ['price']
7
8  training_columns = ['sqft_living',
9                      'grade',
10                     'bathrooms',
11                     'view',
12                     'sqft_basement',
13                     'bedrooms',
14                     'lat',
15                     'waterfront',
16                     'floors',
17                     'yr_renovated',
18                     'sqft_lot',
19                     'yr_built',
20                     'condition',
21                     'long',
22                     'zipcode']
23
24 # Split independent and dependent variables.
25 data_train,data_test,data_train_labels,data_test_labels = train_test_split(data_raw[training_columns],
26                                                                            data_raw[label_columns],
27                                                                            random_state = 42)
28
29 # Compare the number of rows and columns in the original data to the training and testing sets
30 print(f'Original Set:      {data_raw.shape}')
31 print('-----------------------------')
32 print(f'Training Features: {data_train.shape}')
33 print(f'Testing Features:  {data_test.shape}')
34 print(f'Training Labels:   {data_train_labels.shape}')
35 print(f'Testing Labels:    {data_test_labels.shape}')
```

- Line 1 imports the scikit-learn `train_test_split()` method, which will be used to split the raw dataset into separate training and testing datasets.
- Line 6 identifies the column that you'll include in the DataFrame that will contain the labels for machine learning. In this project, `price` is the *dependent* variable, which the model should learn to predict. So the `price` column will provide the labels that you'll use to train the model.
- Lines 8 through 22 identify the columns that you'll include in the DataFrame that contains the attributes (the *independent* variables) the model will use to determine the price. As you train and test the model, you may find that you need to combine, drop, or transform some of these to produce better results.
- Lines 25 through 27 call the `train_test_split()` method to split various columns from the original dataset into four separate datasets:
  - `data_train` contains the independent variables that will be used to train the model.
  - `data_test` contains the independent variables that will be used to test the model after it has been trained.
  - `data_train_labels` contains the dependent variable (`price`) for each row used to train the model.
  - `data_test_labels` contains the dependent variable (`price`) for each row used to test the model.
- The seed value provided in line 27 (`42` in this example, provided as the `random_state` parameter) is used by the random number generator in the `train_test_split()` method to randomly shuffle the records.

The labels function as a sort of "answer key." Just as students practicing math problems shouldn't be provided with the answer until they have worked out the problem on their own, you need to keep labels separate from the data used to train the model.

b) Examine the code that summarizes the split datasets.

- Lines 30 through 35 compare the number of rows and columns in the original data to those in the training and testing sets.
- By default, the `train_test_split()` method will include 75% of the rows in the training set and 25% of the rows in the testing set. Since this code example doesn't specify percentages for the two sets, the default proportions will be used.

3. Run the code and view the results.

a) Select the cell that contains the code listing, and select **Run**.

b) Examine the output.

The datasets have been split as shown.

```
Original Set:      (21613, 21)
-----------------------------
Training Features: (16209, 15)
Testing Features:  (5404, 15)
Training Labels:   (16209, 1)
Testing Labels:    (5404, 1)
```

- The training features and labels include 16,209 of the original 21,613 records, whereas the testing features and labels include only 5,404 records.
- The features datasets include 15 of the original 21 columns, whereas the labels include only 1 column (for price).

4. Save and close the lab.

a) From the menu, select **File→Save and Checkpoint**.

b) Close the lab browser tab and continue on with the course.

MODULE 4

Set Up and Train a Model

The following labs are for Module 4: Set Up and Train a Model.

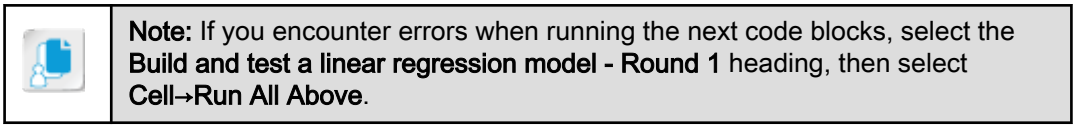
LAB 2-7
Setting Up a Machine Learning Model

Data File
~/Workflow/Workflow-Housing.ipynb

Scenario
You have analyzed the dataset and are now ready to create your first pass at a machine learning model—your hypothesis, which you will use for experimentation and testing. You will start by setting up a linear regression model.

---

1. Open the lab and return to where you were in the notebook.
   a) From the Coursera lab page, select **Open Lab** to open a Jupyter Notebook session in another web browser tab.
   b) Open **Workflow/Workflow-Housing.ipynb**.
   c) Your progress from the previous lab should have been saved.

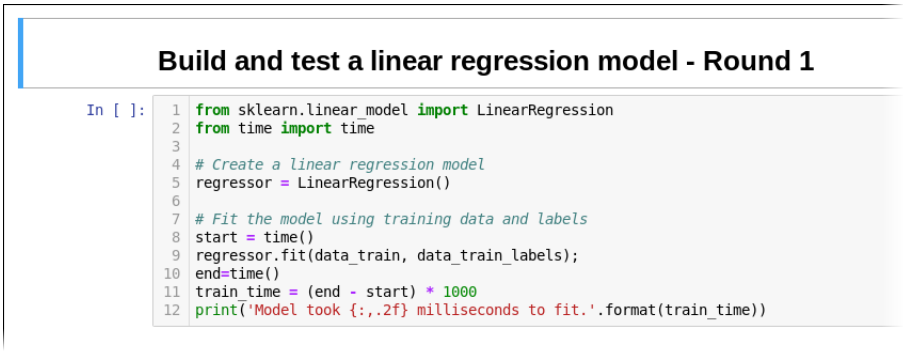
   

2. **Why would you use a linear regression algorithm to produce a real estate price estimator?**

   A: The purpose of the model is to predict the price at which a house with particular attributes will sell. This is a regression type of outcome. While other algorithms can also be used to produce a regression outcome, linear regression is simple, relatively easy to implement, and can produce a good result in many cases.

3. Create a linear regression model and fit it using the training features and labels.

a) Scroll down to view the cell titled **Build and test a linear regression model - Round 1**, and examine the code listing beneath it.



- Line 1 imports the scikit-learn `LinearRegression` library, which will be used to create a linear regression model.
- Line 2 imports the `time` library. You will use this to profile the model's performance, timing how long it takes the model to train.
- Line 5 creates an object from the `LinearRegression()` class, which provides the algorithm that will create the model.
- Line 8 starts the timer.
- Line 9 trains the regression model, providing the training data and labels.
- Lines 10, 11, and 12 calculate and report the time it takes to fit the model.

b) Select the cell that contains the code listing, and select **Run**.

c) Observe the output.

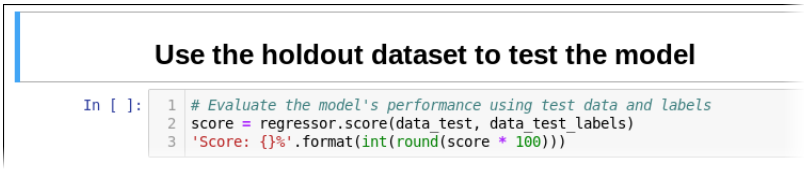
- The model is quickly processed to fit the dataset you provided, with the time it took shown in the output.
- The model now resides in memory (in the `regressor` variable). Of course, now you need to test that the model is actually able to make predictions. You can use the testing dataset for this purpose.

Note: As you can see, except in the case of very large and complicated datasets (which can take several days to process), it may take significantly less time to actually set up the model than it takes to prepare the dataset.

4. Use the holdout dataset to test the model.

a) Scroll down to view the cell titled **Use the holdout dataset to test the model**, and examine the code listing beneath it.

More about the `score()` function will be covered in a later course.



- Line 2 calls the `LinearRegression` object's `score()` function to run a set of predictions on each row of house data in the testing features dataset (`data_test`). It compares those predictions to the actual prices of those houses, which are in `data_test_labels`.
- The `score()` function uses an appropriate algorithm to rate the performance of the regression model. The best possible score that can be returned is 100%. The score can be negative.
- Line 3 outputs the score.

b) Select the cell that contains the code listing, and select **Run**.

c) Examine the score for your linear regression model.

```
'Score: 70%'
```

You may be able to make some adjustments to improve on this score.

5. Compare predicted prices to the actual prices.

a) Scroll down to view the cell titled **Compare predicted values to actual values**, and examine the code listing beneath it.

Compare predicted values to actual values

In [ ]:
```
1 predicted_prices = regressor.predict(data_test)
2 predictions = data_test_labels.copy()
3 predictions['predicted'] = predicted_prices
4
5 # View examples comparing actual prices to predicted prices
6 with pd.option_context('float_format', '${:,.2f}'.format): print( predictions.head(10) )
```

- Line 1 calls the `predict()` function to generate a new DataFrame containing predicted prices for the houses in the test dataset.
- Line 2 makes a copy of the FataFrame that contains the labels (correct answers), and line 3 then adds the predicted prices to the FataFrame as a new column. Having both columns side by side in the same DataFrame will make it easy to compare them.
- Line 6 will display the first 10 rows of the DataFrame, so you can see a sample of the results comparing actual prices to the predicted prices. It applies formatting to show the numbers as U.S. dollars with commas and two decimal places.

b) Select the cell that contains the code listing, and select **Run**.

c) Examine the output.

```
             price     predicted
735     $365,000.00   $451,576.87
2830    $865,000.00   $745,528.73
4106  $1,038,000.00 $1,234,144.11
16218 $1,490,000.00 $1,659,505.67
19964   $711,000.00   $737,851.90
1227    $211,000.00   $284,352.79
18849   $790,000.00   $832,187.44
19369   $680,000.00   $490,462.65
20164   $384,500.00   $392,922.89
7139    $605,000.00   $471,310.48
```

- The first ten actual prices can be compared with the predicted prices.
- While the predicted prices and actual prices correlate roughly, the current level of accuracy may be inadequate. The model could use some improvement.

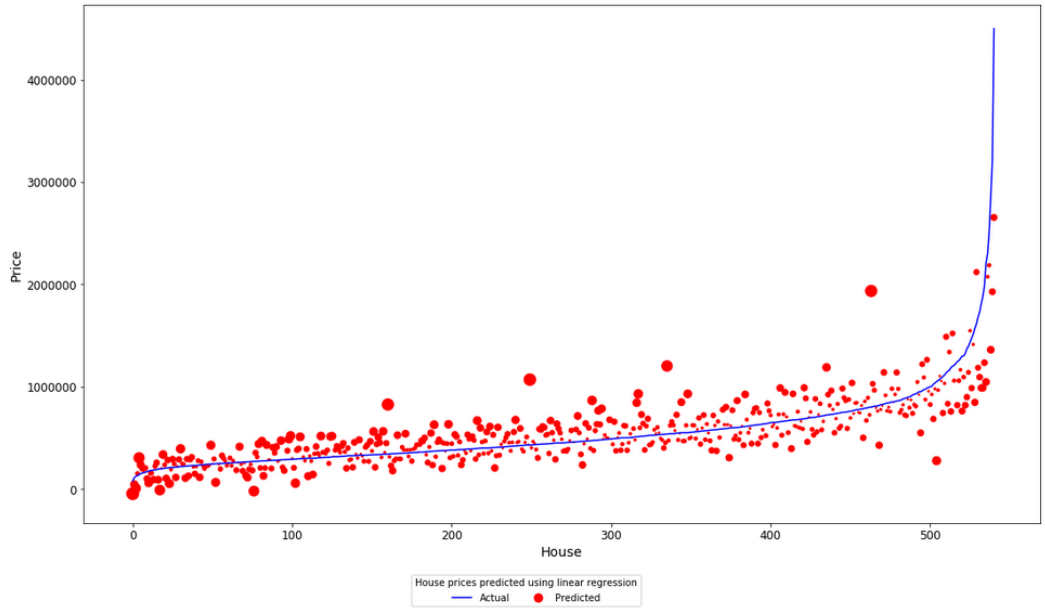
6. Create a chart to compare the predictions to the actual values.

a) Select the next code cell, and examine the code.

```
In [ ]:
1  def compare_pred_to_actual(chart_description):
2
3      N = 10 # Plot every Nth value to save time and space
4      pred_df = predictions.sort_values('price')[::N]
5
6      pred_df['diff'] = pred_df['price'] - pred_df['predicted']
7      pred_df['recnum'] = np.arange(len(pred_df))
8      pred_df['error_pct'] = abs(pred_df['diff']/pred_df['price'])*100
9
10     ax = plt.figure(figsize=[18,10])
11     plt.ylabel('Price')
12     plt.xlabel('House')
13     plt.plot(pred_df['recnum'], pred_df['price'], color='blue');
14     plt.scatter(pred_df['recnum'],
15                 pred_df['predicted'],
16                 pred_df['error_pct'],
17                 color='red');
18
19     ax.legend(['Actual','Predicted'],
20               loc='lower center',
21               ncol=2,
22               title=chart_description)
23
24     plt.show()
25
26 # Compare the predicted prices to actual prices
27 compare_pred_to_actual('House prices predicted using linear regression')
```

- The `compare_pred_to_actual()` function is defined in lines 1 through 24. This function generates a combination line/scatter chart that will compare predicted prices (shown as dot markers) to actual prices (plotted in a line).
- Lines 3 and 4 will create a subset of house records that samples every 10th record. This will remove some of the detail from the chart, but will save processing time and memory, and will still give a good impression of the model's effectiveness.
- Lines 6 through 8 add columns to the dataset, calculate the difference between the predicted and actual price, adding a record number which can be plotted along the x-axis, and calculating the percent of difference between the actual and predicted price.
- The statements in lines 10 through 17 plot the line chart and scatter chart.
- The statement in lines 19 through 22 generates a legend. The chart description passed into the function is shown as the legend's title.
- Line 24 displays the chart.
- Line 27 calls the function defined in lines 1 through 24, passing in a chart description that notes the chart shows prices predicted using a linear regression model.

b) Select **Run**.



- The chart shows a blue line representing actual prices and red dots showing predicted prices.
- The greater the percent of error, the larger the dot.
- You can use this chart as a baseline to visually compare the quality of predictions as you improve the model. As the model improves, the dots should get smaller and clustered more closely to the blue line.

7. Save and close the lab.

a) From the menu, select **File→Save and Checkpoint**.

b) Close the lab browser tab and continue on with the course.

LAB 2-8
Dealing with Outliers

Data File

~/Workflow/Workflow-Housing.ipynb

Scenario

One way to improve a model is to check your training data for noise such as outliers, which may mislead the algorithm and hinder the fit. When you find faulty data values, you can determine the best way to handle them—for example, by correcting them or by simply dropping them from the dataset.

When you observed the histograms for price and bedrooms, there was a lot of white space on the right side of the chart. You'll investigate these features to see if they have outliers in the high end of the range.

---

1. Open the lab and return to where you were in the notebook.
   a) From the Coursera lab page, select **Open Lab** to open a Jupyter Notebook session in another web browser tab.
   b) Open **Workflow/Workflow-Housing.ipynb**.
   c) Your progress from the previous lab should have been saved.

Note: If you encounter errors when running the next code blocks, select the **Identify outliers** heading, then select **Cell→Run All Above**.

2. Identify outliers that you must deal with.
   a) Scroll down to view the cell titled **Identify outliers**, and examine the code listing beneath it.

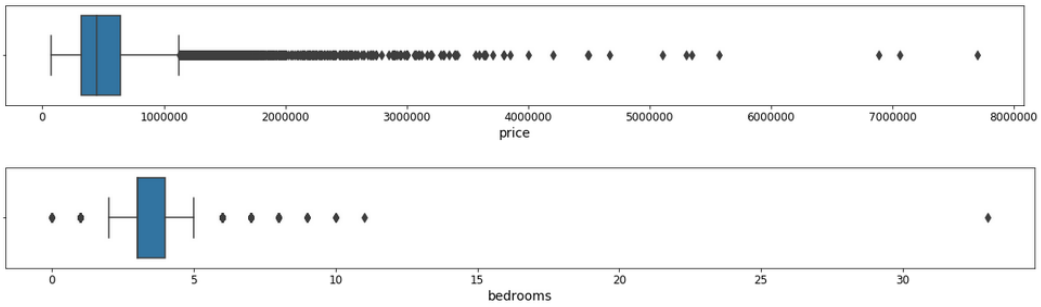
Identify outliers

In [ ]:
```
1 feature_list = ['price','bedrooms']
2
3 for feature in feature_list:
4     plt.figure(figsize=(20,2))
5     bplot = sb.boxplot(x=feature, data=data_raw, orient="h", fliersize=7)
```

- Line 1 identifies the features that you will include in a box plot to find outliers.
- Line 3 iterates through each feature in the list you defined in line 1, repeating the steps in lines 4 and 5 for each feature.
- Line 4 specifies the figure size available to the box plot, and line 5 actually generates it, specifying the name of the feature to be plotted, the DataFrame that holds the data, and formatting options for the box plot.

   b) Select the cell that contains the code listing, and select **Run**.

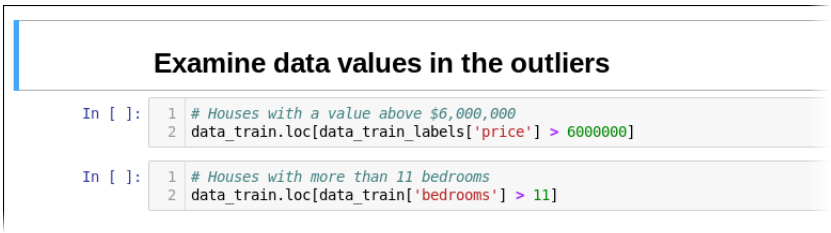
c) Examine the output.



- In both boxplots, the colored boxes show where the middle 50% of the values occur. The vertical lines on either side of the boxes (the "whiskers") show the boundaries of the minimum (`Q1 - 1.5 × IQR`) and maximum (`Q3 + 1.5 × IQR`) for the distributions.
- The upper boxplot shows the distribution of `price` values in the original dataset. There is a long tail on the right, with the distribution tapering off around $5,500,000, followed by a gap, and three more outliers on the far right.
- The lower boxplot shows the distribution of `bedrooms` values in the original dataset. The distribution tapers off around 11 bedrooms, and then there is a single outlier with more than 30 bedrooms. That value is likely an error.

3. Examine the data values for the outliers.

a) Scroll down to view the cell titled **Examine data values in the outliers**, and examine the two code cells beneath it.



- In the first code cell, Line 2 contains a statement that will return a subset of records from `data_train`. For any row in the labels where the price is more than 6 million dollars, the corresponding row from the training dataset will be returned.
- Line 2 in the second code cell does something similar, returning records from the training set where the number of bedrooms is more than 11.
- When run, the statements in these two cells will display lists of the outlier values you saw in the boxplots.

b) Select the first (upper) code cell, and select **Run**.

c) Select **Run** again to run the second cell.

d) Examine the data values for the outliers.

```
# Houses with a value above $6,000,000
data_train.loc[data_train_labels['price'] > 6000000]
```

	sqft_living	grade	bathrooms	view	sqft_basement	bedrooms	lat	waterfront	floors	yr_renovated	sqft_lot	yr_built	condition	long	zipcoc
3914	10040	11	4.50	2	2360	5	47.6500	1	2.0	2001	37325	1940	3	-122.214	980(
9254	9890	13	7.75	4	1030	6	47.6305	0	2.0	0	31374	2001	3	-122.240	980:
7252	12050	13	8.00	3	3480	6	47.6298	0	2.5	1987	27600	1910	4	-122.323	981(

```
# Houses with more than 11 bedrooms
data_train.loc[data_train['bedrooms'] > 11]
```

	sqft_living	grade	bathrooms	view	sqft_basement	bedrooms	lat	waterfront	floors	yr_renovated	sqft_lot	yr_built	condition	long	zipcc
15870	1620	7	1.75	0	580	33	47.6878	0	1.0	0	6000	1947	5	-122.331	981

Note: You may have to scroll slightly to the right and left to see all of the columns.

- In the upper listing, the `price` values are not shown since the prices are stored in the separate labels dataset, but you can see that all three of these (most expensive) houses have a high grade, large number of bedrooms, bathrooms, and high square footage.
- The lower listing shows a house with 33 bedrooms. It only has 1,620 square feet of living space, so this value is almost certainly incorrect. You could either fix the value or just drop the record from the training set.

Note: Because the dataset is large, these outliers may not have much impact on the model, but it won't hurt to remove them from the training set either. Doing so in this lab will enable you to see how it's done in Python.

4. Drop outliers from the training dataset.

a) Scroll down to view the cell titled **Drop outliers from the training dataset**, and examine the code cell beneath it.

Drop outliers from the training dataset

In [ ]:

```
1  print(f'{len(data_train):6d} houses in the training dataset')
2
3  # Keep only the rows for houses priced $6M or less
4  data_train = data_train.loc[data_train_labels['price'] <= 6000000]
5  data_train_labels = data_train_labels.loc[data_train_labels['price'] <= 6000000]
6  print(f'{len(data_train):6d} houses remain after dropping those priced over $6M')
7
8  # Keep only the rows for houses with 11 or fewer bedrooms
9  data_train_labels = data_train_labels.loc[data_train['bedrooms'] <= 11]
10 data_train = data_train.loc[data_train['bedrooms'] <= 11]
11 print(f'{len(data_train):6d} houses remain after dropping those with more than 11 bedrooms')
```

- Line 4 filters the training dataset to include only records whose price is less than six million dollars.
- Line 5 performs the same filter operation on the training labels. It is important to keep these two sets parallel, dropping exactly the same rows from each.
- Lines 9 and 10 are similar to lines 4 and 5, dropping records from the training dataset and labels to remove any records for houses with more than 11 bedrooms.
- The print statements in lines 1, 6, and 11 provide a before-and-after summary so you can see the results of dropping the outliers.

b) Select the cell that contains the code listing, and select **Run**.

c) Examine the output.

```
16209 houses in the training dataset
16206 houses remain after dropping those priced over $6M
16205 houses remain after dropping those with more than 11 bedrooms
```

After dropping the four outliers, 16,205 records remain in the dataset.

5. Save and close the lab.

a) From the menu, select **File→Save and Checkpoint**.
b) Close the lab browser tab and continue on with the course.

LAB 2-9
Scaling and Normalizing Features

Data File
~/Workflow/Workflow-Housing.ipynb

Scenario
As you continue with your optimization of the model, you will look for additional problems with the training data. You will now look for data values that should be adjusted to produce a better linear regression model.

---

1. Open the lab and return to where you were in the notebook.
   a) From the Coursera lab page, select **Open Lab** to open a Jupyter Notebook session in another web browser tab.
   b) Open **Workflow/Workflow-Housing.ipynb**.
   c) Your progress from the previous lab should have been saved.

Note: If you encounter errors when running the next code blocks, select the **Show statistics for the training features** heading, then select **Cell→Run All Above**.

2. Compare the ranges of `sqft_living` and `price`.
   a) Scroll down to view the cell titled **Show statistics for the training features**, and examine the code listing beneath it.

   - Line 2 sets the context for lines 3 and 4, so number values will be shown with two decimal places.
   - Line 3 outputs descriptive statistics for the values in the `sqft_living` column.
   - Line 4 outputs descriptive statistics for the values in the `price` column.

   b) Select the cell that contains the code listing, and select **Run**.

c) Examine the output.

```
count   16205.00
mean     2071.71
std       899.46
min       290.00
25%      1427.00
50%      1910.00
75%      2544.00
max      9640.00
Name: sqft_living, dtype: float64

count     16205.00
mean     536227.60
std      348666.79
min       75000.00
25%      320000.00
50%      450000.00
75%      640000.00
max     5110800.00
Name: price, dtype: float64
```

- The minimum and maximum values for `sqft_living` are 290 and 9,640.
- The minimum and maximum values for `price` are 75,000 and 5,110,800.
- The scale of these values is quite different.

3. Examine the distribution of `sqft_living` and `price`.

a) Scroll down to view the cell titled **Compare the scale and distribution of price and sqft_living**, and examine the code listing beneath it.

Compare the scale and distribution of price and sqft_living

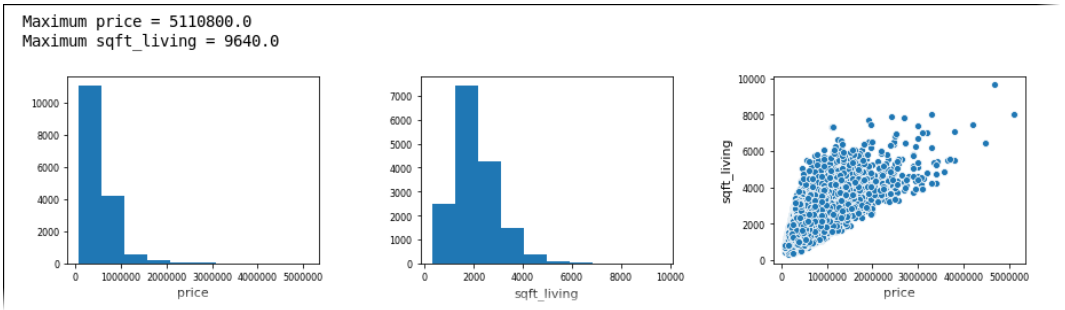
In [ ]:

```
1  # Compare scale and distribution of price and sqft_living
2  def compare_price_sqft():
3
4      print('Maximum price =', data_train_labels.loc[data_train_labels['price'].idxmax()]['price']);
5      print('Maximum sqft_living =', data_train.loc[data_train['sqft_living'].idxmax()]['sqft_living']);
6
7      fig = plt.figure(figsize=(15,3))
8      fig.subplots_adjust(wspace=.4)
9
10     plt.rc('axes', titlesize=9)     # fontsize of the axes title
11     plt.rc('axes', labelsize=11)    # fontsize of the x and y labels
12     plt.rc('xtick', labelsize=8)    # fontsize of the tick labels
13     plt.rc('ytick', labelsize=8)    # fontsize of the tick labels
14
15     ax1=fig.add_subplot(1, 3, 1)
16     plt.xlabel('price')
17     plt.hist(data_train_labels['price'], label='price');
18
19     ax2=fig.add_subplot(1, 3, 2)
20     plt.xlabel('sqft_living')
21     plt.hist(data_train['sqft_living'], label='sqft_living');
22
23     # View relationship between price and sqft_living
24     ax2=fig.add_subplot(1, 3, 3)
25     sb.scatterplot(x=data_train_labels['price'], y=data_train['sqft_living']);
26
27 compare_price_sqft()
```

- Lines 4 and 5 show the maximum values for `price` and `sqft_living`.
- Lines 7 through 13 prepare to create a visualization.
- Lines 15 through 17 plot a histogram to show the distribution of `price` values.
- Lines 19 through 21 plot a histogram to show the distribution of `sqft_living` values.
- Lines 24 and 25 plot a scatterplot to show the relationship between `sqft_living` and `price`.
- All of the lines described above are contained within the `compare_price_sqft()` function, which is called in line 27 to generate the charts.

b) Select the cell that contains the code listing, and select **Run**.

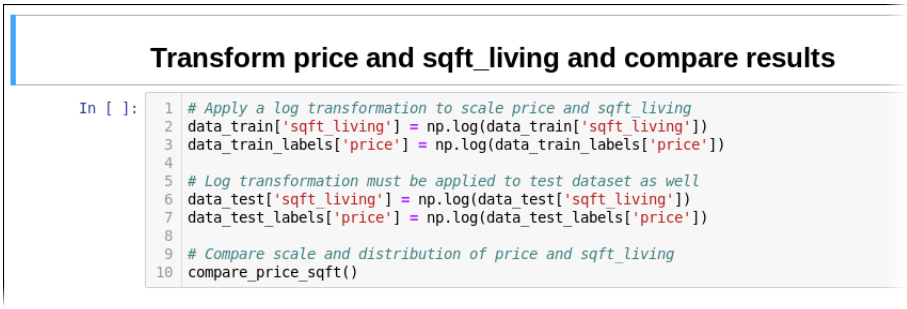
c) Examine the output.



- The maximum `price` is 5,110,800, while the maximum `sqft_living` is 9,640. These values are in very different scales.
- The left two charts show the distributions of `price` and `sqft_living` skewing to the right.
- The chart on the right, the scatterplot, shows a cone-shaped dispersion. This occurs because the values for `sqft_living` and `price` spread differently. Statisticians call this condition *heteroscedasticity*, and it limits the effectiveness of linear regression.
- To optimize the linear regression model, you should transform the values in these two columns to be more similar in scale and spread.

4. Transform `price` and `sqft_living` to be more similar in scale and spread.

a) Scroll down to view the cell titled **Transform price and sqft_living and compare results**, and examine the code listing beneath it.



Transform price and sqft_living and compare results

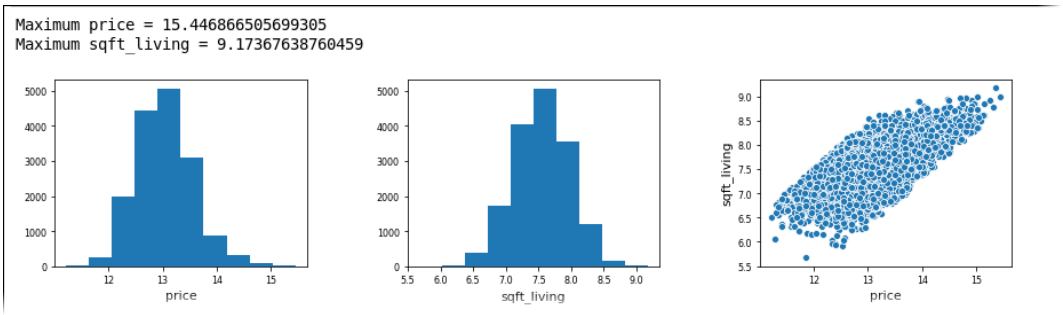
In [ ]:

```
1  # Apply a log transformation to scale price and sqft_living
2  data_train['sqft_living'] = np.log(data_train['sqft_living'])
3  data_train_labels['price'] = np.log(data_train_labels['price'])
4
5  # Log transformation must be applied to test dataset as well
6  data_test['sqft_living'] = np.log(data_test['sqft_living'])
7  data_test_labels['price'] = np.log(data_test_labels['price'])
8
9  # Compare scale and distribution of price and sqft_living
10 compare_price_sqft()
```

- Lines 2 and 3 apply a log transformation to the two columns in the training dataset.
- Lines 6 and 7 apply the log transformation to the same columns in the testing dataset.
- Line 10 calls the `compare_price_sqft()` function again, so you can see the results of the log transformation on the two columns.

b) Select the cell that contains the code listing, and select **Run**.

c) Examine the output.



- After transformation, the maximum `price` is about 15.45, and the maximum `sqft_living` is 9.17. These values are now in a much more similar scale than they were before.
- Both `price` and `sqft_living` now show a normal distribution curve, with the skew eliminated.
- The rightmost chart, the scatterplot, is no longer cone-shaped. The highest and lowest edges are roughly parallel. This makes it more suitable for linear regression.

5. Save and close the lab.

a) From the menu, select **File→Save and Checkpoint**.

b) Close the lab browser tab and continue on with the course.

LAB 2-10
Refitting and Testing the Model

Data File
~/Workflow/Workflow-Housing.ipynb

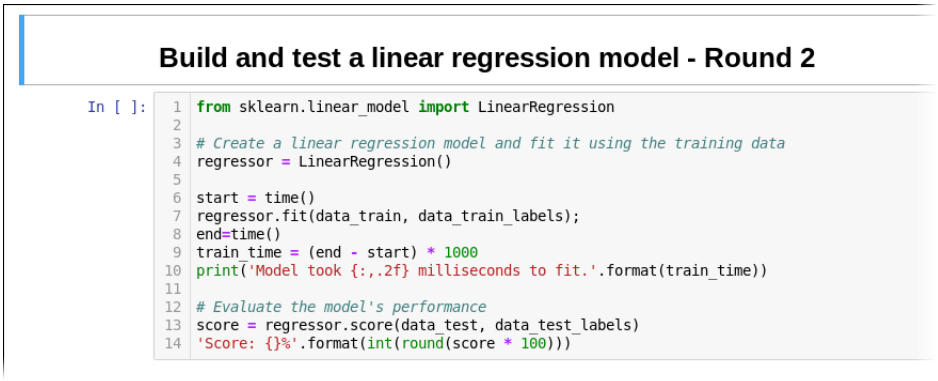
Scenario
You will now test the model again, to see whether the adjustments you made to the datasets will improve the results.

---

1. Open the lab and return to where you were in the notebook.
   a) From the Coursera lab page, select **Open Lab** to open a Jupyter Notebook session in another web browser tab.
   b) Open **Workflow/Workflow-Housing.ipynb**.
   c) Your progress from the previous lab should have been saved.

Note: If you encounter errors when running the next code blocks, select the **Build and test a linear regression model - Round 2** heading, then select **Cell→Run All Above**.

2. Compare the scaling of `sqft_living` and `price`.
   a) Scroll down to view the cell titled **Build and test a linear regression model - Round 2**, and examine the code listing beneath it.



Build and test a linear regression model - Round 2

In [ ]:

```
1  from sklearn.linear_model import LinearRegression
2
3  # Create a linear regression model and fit it using the training data
4  regressor = LinearRegression()
5
6  start = time()
7  regressor.fit(data_train, data_train_labels);
8  end=time()
9  train_time = (end - start) * 1000
10 print('Model took {:,.2f} milliseconds to fit.'.format(train_time))
11
12 # Evaluate the model's performance
13 score = regressor.score(data_test, data_test_labels)
14 'Score: {}%'.format(int(round(score * 100)))
```

   - Lines 1 through 14 create and fit a linear regression model similar to the one you created in Round 1.
   - The model will be re-fit using the updated training data.

   b) Select the cell that contains the code listing, and select **Run**.

c) Examine the output.

```
Model took 14.45 milliseconds to fit.

'Score: 77%'
```

- The score has increased from 70% to 77%.
- Improvements you made to the dataset have led to improved results.

Note: In addition to the size and complexity of the dataset and the complexity of the algorithm, other factors affect the amount of time needed to fit a model, including the processor speed, background processes currently running, and so forth. Your lab session may have taken a different amount of time to fit the model than what is shown here.

3. Compare predicted prices to the actual prices.

a) Scroll down to view the cell titled **Compare the first ten predictions to actual values**, and examine the code listing beneath it.

Compare the first ten predictions to actual values

In [ ]:

```
# y_pred is the predicted prices that will be produced by testing
predicted_prices = regressor.predict(data_test)
predictions = data_test_labels.copy()
predictions['predicted'] = predicted_prices

# View examples of the transformed prices
with pd.option_context('float_format', '${:,.2f}'.format): print( predictions.head(10) )
```

- Again, this code is similar to code you ran in Round 1.
- This will show you the first ten rows of actual prices compared to predicted prices, so you can see how well the new model is making its predictions.

b) Select the cell that contains the code listing, and select **Run**.

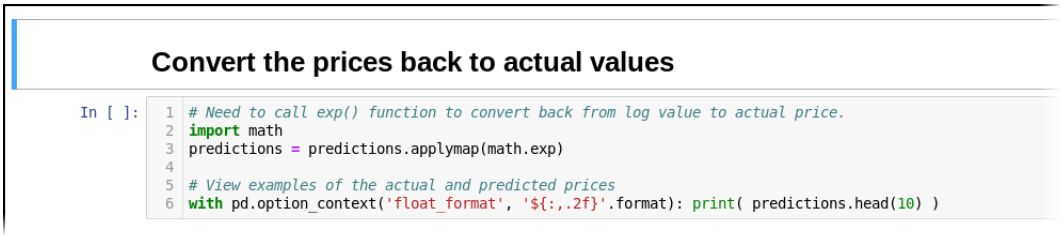
c) Examine the output.

```
       price  predicted
735   $12.81     $12.95
2830  $13.67     $13.44
4106  $13.85     $14.01
16218 $14.21     $14.58
19964 $13.47     $13.45
1227  $12.26     $12.62
18849 $13.58     $13.59
19369 $13.43     $13.04
20164 $12.86     $12.92
7139  $13.31     $12.90
```

- Because the prices were scaled using a log transformation, the values don't reflect the actual prices.
- You will have to transform these prices back to actual values. Python's `math` library provides an `exp()` function, which you can use to reverse the `log()` function.

4. Convert prices back to actual values.

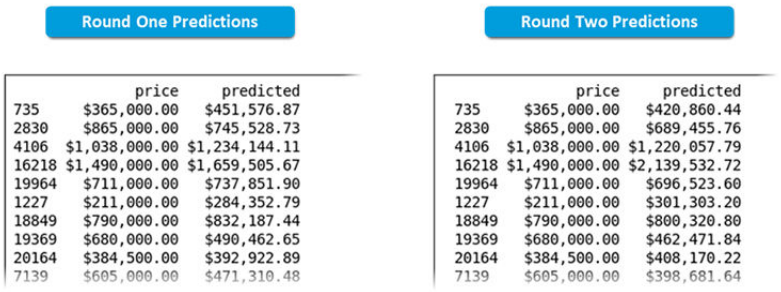
a) Scroll down to view the cell titled **Convert the prices back to actual values**, and examine the code listing beneath it.



- Line 2 imports the `math` library.
- Line 3 calls the pandas library's `applymap()` function to apply the `exp()` function to every data item in the predictions DataFrame.
- Line 6 outputs the first ten rows of predictions, showing the values with formatting for U.S. dollars.

b) Select the cell that contains the code listing, and select **Run**.

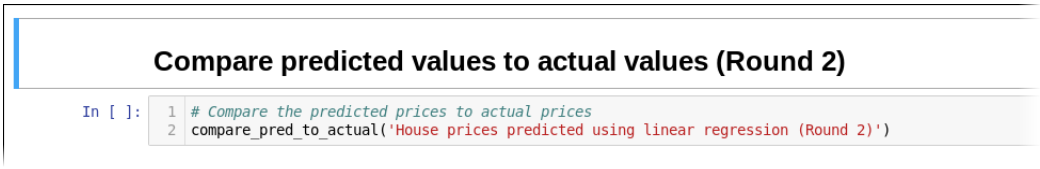
c) Examine the output.



- The prices have been transformed back to reflect real prices.
- Although the overall score has improved, by comparing individual predictions in the two rounds, it may not be easy to see that the predicted prices are now generally closer to the actual prices.

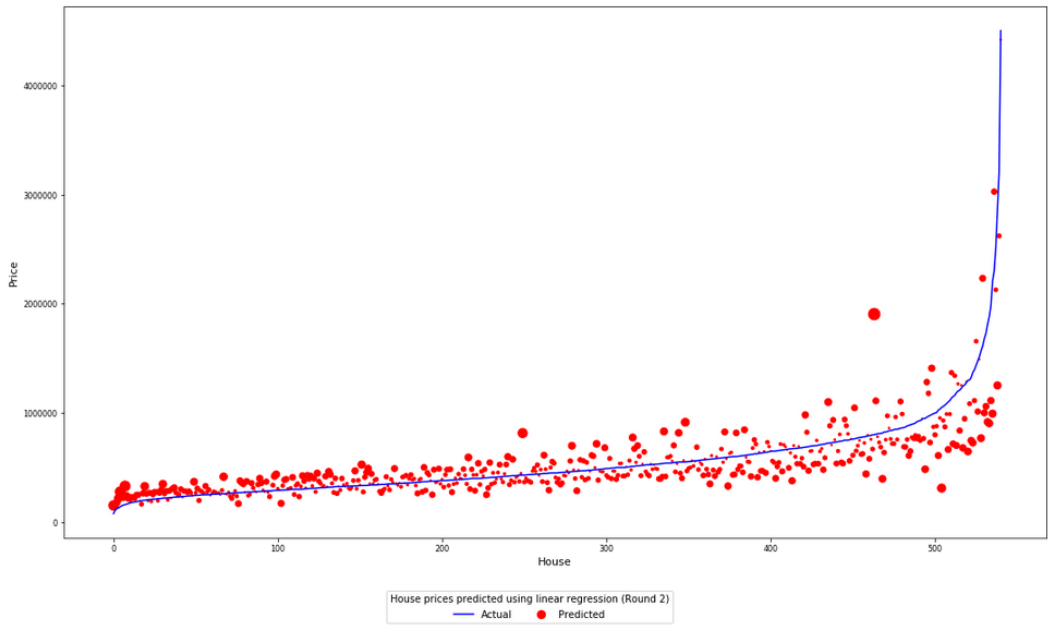
5. Create a visualization to show the accuracy of the predictions.

a) Scroll down to view the cell titled **Compare predicted values to actual values (Round 2)**, and examine the code listing beneath it.



Line 2 calls the `compare_pred_to_actual()` function that was previously defined, to generate a new chart.

a) Select the cell that contains the code listing, and select **Run**.



- If you compare back to the previous chart, you can see that there has been some improvement.
- The dots are a bit more clustered toward the blue line, and generally smaller than before.

6. Try a different algorithm.

a) Scroll down to view the cell titled **Try a different algorithm**, and examine the code listing beneath it.

Try a different algorithm

In [ ]:

```
# Create a model using the random forest algorithm.
from sklearn.ensemble import RandomForestRegressor

rnd_forest = RandomForestRegressor(n_estimators=100,random_state=0)

start = time()
rnd_forest.fit(data_train, data_train_labels)
end=time()
train_time = (end - start) * 1000
print('Model took {:,.2f} milliseconds to fit.'.format(train_time))

score = rnd_forest.score(data_test, data_test_labels)
print('Score: {}%'.format(int(round(score * 100))))
```

This code is similar to the code you used to create a linear regression model, only now you are creating a model using a random forest class that employs a combination of various regression algorithms to come up with the best fit.

This is another algorithm that can produce a prediction outcome. At this point in the course the objective is simply to show that a workflow might involve trying out several different algorithms. The random forest algorithm is covered in detail in a later course.

b) Select the cell that contains the code listing, and select **Run**.

	Note: Be patient, as the random forest algorithm will take much longer to finish than the linear regression. This operation may take half a minute or more to complete.

a) Examine the output.

```
Model took 33,572.89 milliseconds to fit.
Score: 89%
```

- Although this model took longer to fit, it produced a much better score than the linear regression model.
- Often, you'll have to make tradeoffs between time and performance, deciding whether measures like accuracy and recall outweigh the need for producing quick results.

7. Create a visualization to show the accuracy of the random forest predictions.

a) Scroll down to view the cell titled **View examples of the actual and predicted prices**, and examine the code listing beneath it.

View examples of the actual and predicted prices

In [ ]:

```
predicted_prices = rnd_forest.predict(data_test)

predictions = data_test_labels.copy()
predictions['predicted'] = predicted_prices

# Scale the prices back to actual values.
predictions = predictions.applymap(math.exp)

# View examples of the actual and predicted prices
with pd.option_context('float_format', '${:,.2f}'.format): print( predictions.head(10) )
```

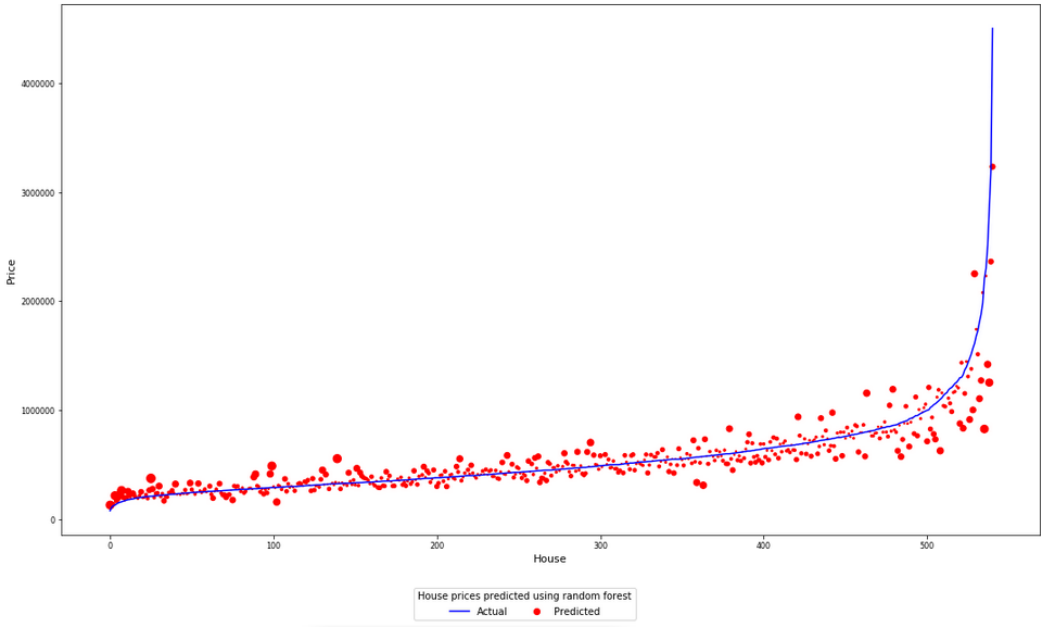
b) Select the cell that contains the code listing, and select **Run**.

c) Examine the output.

```
             price     predicted
735     $365,000.00   $367,229.33
2830    $865,000.00   $797,830.56
4106  $1,038,000.00 $1,112,227.86
16218 $1,490,000.00 $1,866,465.53
19964   $711,000.00   $709,144.08
1227    $211,000.00   $251,855.28
18849   $790,000.00   $870,082.54
19369   $680,000.00   $562,736.77
20164   $384,500.00   $413,123.06
7139    $605,000.00   $534,529.34
```

These predicted prices tend to be more accurate than those you produced with the linear regression model.

d) Run the last code cell and examine the output.



The last code cell generates an updated bar chart comparing the actual prices with the prices predicted by the random forest model. The prices in this model tend to be closer than those shown in the chart for the linear regression model.

8. Shut down the notebook and close the lab.

a) From the menu, select **Kernel→Shutdown**.

b) In the **Shutdown kernel?** dialog box, select **Shutdown**.

c) Close the lab browser tab and continue on with the course.

MODULE 5

Project

The following lab is for the Course 2 Project.

PROJECT 2

Following a Machine Learning Workflow to Predict Demand for Bicycle Rentals

Data File

~/Projects/Workflow.ipynb

Scenario

A company that rents bicycles on the streets of Seoul, South Korea is looking to improve its logistics. In particular, the company wants to provide an adequate number of bikes to meet the demands of customers, without providing more bikes than is necessary. Demand for bike rentals fluctuates based on several factors, including time of day, temperature, and weather patterns. So, you've been asked to create a machine learning model that can help the company predict how many bikes to allocate at any given time so the city has a stable supply for its citizens.

The bike company has provided you with a dataset of historical records that span nearly two years, where each record represent an hour in which the rental service operated. The following are the attributes of the dataset:

Attribute	*Description*
`bikes_rented`	Total number of bikes rented in an hour. This is the label you need to predict.
`temp`	Ambient temperature in Celsius (°C).
`humidity`	Relative humidity expressed as a percentage.
`wind_speed`	Wind speed in meters per second (m/s).
`visibility`	Average visibility in 10s of meters.
`dew_temp`	Dew point temperature in Celsius (°C).
`solar_rad`	Solar radiation in millijoules per square meter (mJ/m2).
`rainfall`	Rainfall in millimeters (mm).
`snowfall`	Snowfall in centimeters (cm).

Note: There is no "correct" answer for every line of code, as there are many ways to write code to do the same thing. You can write the code however you feel most comfortable, as long as it accomplishes the tasks set out for you.

Note: If you're stuck and need help, you can "borrow" code from existing notebooks, and/or look up documentation on how to use the various Python libraries.

Note: Sometimes, when you run code containing logic errors or bugs, you may corrupt the data contained in variables you created in previous code cells. To clear out such problems, you can select **Kernel→Restart & Clear Output**, then run each code cell again.

1. Open the lab and notebook.

a) From the Coursera lab page, select **Open Lab** to open a Jupyter Notebook session in another web browser tab.
b) Select **Projects/Workflow.ipynb** to open it.
c) Observe the notebook.

Placeholder code cells have been provided in which you can add your own code. Comments provide hints on tasks you might perform in each code cell. The first code cell has already been completely programmed for you.

2. Import software libraries.
   a) Select the code listing under **Import software libraries**.
   b) Run the code and examine the results.

3. Load the dataset.
   a) In the code block under **Load the dataset**, write code to load the dataset located in **./Projects/seoul_bike_data/seoul_bike_data.csv** into a pandas `DataFrame`.
   b) Run the code cell and verify that the dataset was loaded.

4. Get acquainted with the dataset.
   a) In the code block under **Get acquainted with the dataset**, write code to show information about the dataset, like its dimensions, data types, and any missing values.
   b) Run the code cell and verify that information about the dataset is displayed.

5. Show example records.
   a) In the code block under **Show example records**, write code to show the first 10 records of the dataset.
   b) Run the code and verify that the first 10 records are shown.

6. Examine a general summary of statistics.
   a) In the code block under **Examine a general summary of statistics**, write code to print summary statistics for the dataset (mean, standard deviation, min, max, etc.).
   b) Run the code and verify that the summary statistics are shown.

7. Look for columns that correlate with `bikes_rented`.
   a) In the code block under **Look for columns that correlate with `bikes_rented`**, write code to display the correlations between the features and `bikes_rented`.
   b) Run the code and verify that the correlations are shown.

8. Visually analyze cross correlations.
   a) In the code block under **Visually analyze cross correlations**, write code to display the feature correlations visually using Seaborn.

Note: Seaborn's `heatmap()` function is useful for this.

   b) Run the code and verify that a heatmap of the correlations is shown.

9. Use histograms to visualize the distribution of all features.
   a) In the code block under **Use histograms to visualize the distribution of all features**, write code to draw histograms using Matplotlib.
   b) Run the code and verify that the distribution histograms are shown for all features.

10. Split the data into training and testing sets and labels.

a) In the code block under **Split the data into training and testing sets and labels**, write code to split the training and test datasets and their labels. To confirm the split, print the shape of the training set, test set, training labels, and test labels.
b) Run the code cell and verify that the data is being split into a training set, test set, training label set, and test label set.

11. Build and test an initial linear regression model.

a) In the code block under **Build and test an initial linear regression model**, write code to construct a linear regression class object, then fit the training data to that object.
b) Run the code.

12. Use the holdout dataset to test the model.

a) In the code block under **Use the holdout dataset to test the model**, write code to generate a score for the initial linear regression model.
b) Run the code cell and verify the initial model's score.

13. Compare the first ten predictions to actual values.

a) In the code block under **Compare the first ten predictions to actual values**, write code to compare the first ten predictions for bikes rented to their actual values.
b) Run the code cell and verify that the predictions were printed alongside their actual values.

14. Identify outliers.

a) In the code block under **Identify outliers**, write code to display box plots of `bikes_rented` and `wind_speed`.
b) Run the code cell and verify box plots for both variables are shown.

15. Examine data values in the outliers.

a) In the first code block under **Examine data values in the outliers**, write code to retrieve only rows that exceed 3,500 bikes rented.
b) Run the code cell and verify that the expected row(s) were returned.
c) In the second code block under **Examine data values in the outliers**, write code to retrieve only rows that exceed 6 m/s in wind speed.
d) Run the code cell and verify that the expected row(s) were returned.

16. Drop outliers from the training dataset.

a) In the code block under **Drop outliers from the training dataset**, write code to drop the outliers identified in the previous step. Remember to drop these outliers from both the training and test sets.
b) Run the code.

17. Compare the scale and distribution of `bikes_rented` and `wind_speed`.

a) In the code block under **Compare the scale and distribution of `bikes_rented` and `wind_speed`**, write code that defines a function that compares the scale and distribution of bikes rented and wind speed using Matplotlib. Then, call the function.
b) Run the code and verify that the comparison is shown.

18. Transform `bikes_rented` and `wind_speed`, and compare the results.

a) In the code block under **Transform `bikes_rented` and `wind_speed`, and compare the results**, write code to apply a logarithmic transformation to both bikes rented and wind speed.
b) Run the code cell and verify the transformation's effect on the features' distributions.

19. Build and test a new linear regression model.

a) In the code block under **Build and test a new linear regression model**, write code to build a new linear regression model using the data you just transformed as training input. Then, print the regressor's score.

b) Run the code and verify the change in score.

20. Compare the first ten predictions to actual values for the new model.

a) In the code block under **Compare the first ten predictions to actual values for the new model**, write code to compare the first ten predictions for bikes rented to their actual values using the new linear regression model.

b) Run the code cell and verify that the predictions were printed alongside their actual values.

21. Convert the bike rentals back to their initial scale

a) In the code block under **Convert the bike rentals back to their initial scale**, write code to apply an exponential transformation to the predicted data so that the bike rental counts are back to their initial scale. Then, display the prediction comparisons using these converted values.

b) Run the code cell and verify that the data is back to its initial scale.

22. Save and download the notebook.

a) From the menu, select **File→Save and Checkpoint**.

b) Select **File→Download as→Notebook (.ipynb)**.

c) Save the file to your local drive if it wasn't automatically.

d) Find the saved file and rename it using the following convention: **FirstnameLastname-Workflow-Project.ipynb**.

For example: **JohnSmith-Workflow-Project.ipynb**.

Note: Make sure not to lose this file. This is the project you'll be uploading and sharing to your peers for grading.

23. Shut down the notebook and close the lab.

a) Back in Jupyter Notebook, select **Kernel→Shutdown**.

b) In the **Shutdown kernel?** dialog box, select **Shutdown**.

c) Close the lab browser tab and continue on to the peer review.

Course 3: Build Regression, Classification, and Clustering Models

Course Introduction

The following labs are for Course 3: Build Regression, Classification, and Clustering Models.

Modules

The labs in this course pertain to the following modules:

- Module 1: Build Linear Regression Models Using Linear Algebra
- Module 2: Build Regularized and Iterative Linear Regression Models
- Module 3: Train Classification Models
- Module 4: Evaluate and Tune Classification Models
- Module 5: Build Clustering Models
- Apply What You've Learned

MODULE 1

Build Linear Regression Models Using Linear Algebra

The following lab is for Module 1: Build Linear Regression Models Using Linear Algebra.

LAB 3-1

Building a Regression Model Using Linear Algebra

Data Files

~/Linear Regression/LinearRegression-PowerPlant.ipynb

~/Linear Regression/power_plant_data/cc_power_plant_data.csv

Scenario

IOT Company supplies network-connected sensors used to measure a variety of factors in industrial settings. One application of these sensors is in power plants, where they measure ambient conditions inside and outside the plant, like temperature and humidity. These conditions can have an impact on the efficiency of the power plant's energy output.

You've been given a dataset with hourly measurements over a six-year period, and are tasked with predicting the energy output of the plant given certain measurements. Because the energy output is measured in megawatts (MW), a numeric value, you'll use a simple linear model to make these predictions.

Note: This dataset was retrieved from the UCI Machine Learning Repository at: **https://archive.ics.uci.edu/ml/datasets/combined+cycle+power+plant**.

---

1. Open the lab and notebook.
   a) From the Coursera lab page, select **Open Lab** to open a Jupyter Notebook session in another web browser tab.
   b) Select **Linear Regression/LinearRegression-PowerPlant.ipynb** to open it.

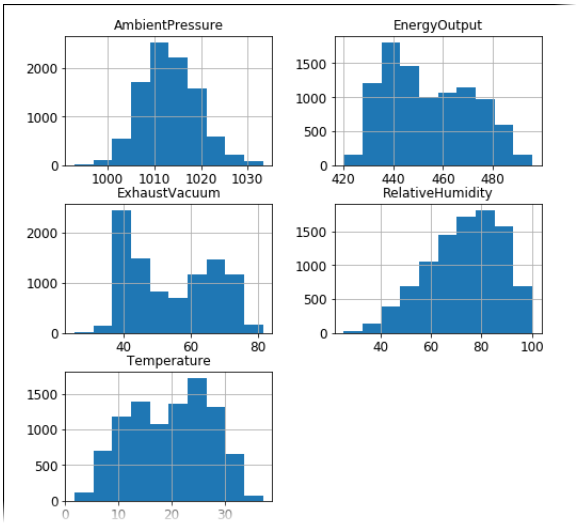
2. Import the relevant libraries and load the dataset.
   a) View the cell titled **Import software libraries and load the dataset**, and examine the code listing below it.
   b) Select the cell that contains the code listing, then select **Run**.
   c) Verify that **cc_power_plant_data.csv** was loaded with 9,568 records.

3. Get acquainted with the data.
   a) Scroll down and view the cell titled **Get acquainted with the dataset**, and examine the code listing below it.
   b) Select the cell that contains the code listing, then select **Run**.
   c) Examine the output.
      - The training set includes 9,568 rows and 5 columns.
      - All of the columns contain float values.
      - There is no missing data; all rows have values for every column.
      - Each column refers to a particular measurement taken from sensors placed around the power plant:
        - `Temperature` is the temperature of the system in Celsius.
        - `ExhaustVacuum` measures the pressure of air as it is pushed out of the system.
        - `AmbientPressure` is the air pressure surrounding the system.
        - `RelativeHumidity` measures humidity at a given temperature.

- `EnergyOutput` is the net electrical energy output by the system in megawatts.
- Each row represents an hourly average for each measurement over a six-year period.
- The `EnergyOutput` column will be treated as the label the model will try to predict.

4. Examine the distribution of the features.

a) Scroll down and view the cell titled **Examine the distribution of the features**, and examine the code listing below it.
b) Select the cell that contains the code listing, then select **Run**.
c) Examine the output.



- The distribution for `AmbientPressure` is roughly symmetrical.
- The distribution for `RelativeHumidity` appears left-skewed.
- The other features' distributions are more varied.

5. Examine a general summary of statistics.

a) Scroll down and view the cell titled **Examine a general summary of statistics**, and examine the code listing below it.
b) Select the cell that contains the code listing, then select **Run**.
c) Examine the output.

```
       Temperature  ExhaustVacuum  AmbientPressure  RelativeHumidity  \
count      9568.00        9568.00          9568.00           9568.00
mean         19.65          54.31          1013.26             73.31
std           7.45          12.71             5.94             14.60
min           1.81          25.36           992.89             25.56
25%          13.51          41.74          1009.10             63.33
50%          20.34          52.08          1012.94             74.97
75%          25.72          66.54          1017.26             84.83
max          37.11          81.56          1033.30            100.16

       EnergyOutput
count       9568.00
mean         454.37
std           17.07
min          420.26
25%          439.75
50%          451.55
75%          468.43
max          495.76
```

- Compared to the other features, `AmbientPressure` and `EnergyOutput` seem to exhibit a low amount of variance, as their minimum and maximum values are relatively close together.
- Likewise, the scale of these two features seems out of alignment with the other features. However, feature scaling does not improve a simple linear model's skill, so you'll leave the features as they are.

6. Look for columns that correlate with `EnergyOutput`.

a) Scroll down and view the cell titled **Look for columns that correlate with `EnergyOutput`**, and examine the code listing below it.

b) Select the cell that contains the code listing, then select **Run**.

c) Examine the output.

```
Correlations with EnergyOutput
EnergyOutput        1.000000
AmbientPressure     0.518429
RelativeHumidity    0.389794
ExhaustVacuum      -0.869780
Temperature        -0.948128
Name: EnergyOutput, dtype: float64
```

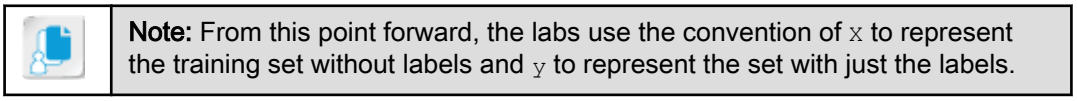
All four of the features have a decent amount of correlation, with `AmbientPressure` having the highest positive correlation (i.e., as the pressure goes up, so does the energy output) and `ExhaustVacuum` and `Temperature` having the highest negative correlation (i.e., as they increase, the energy output decreases). You'll therefore use all of these features during training.

7. Split the datasets.

a) Scroll down and view the cell titled **Split the datasets**, and examine the code listing below it.

This code will split the dataset and training labels.

b) Select the cell that contains the code listing, then select **Run**.



c) Examine the output.

```
Original set:        (9568, 5)
-----------------------------
Training features:   (7176, 4)
Test features:       (2392, 4)
Training labels:     (7176, 1)
Test labels:         (2392, 1)
```

The original training dataset has now been split into two: one set to continue using as training, the other to use for testing. Note that `EnergyOutput` label has been removed from the `X` matrices and placed into its own `y` vector.

8. Create a linear regression model.

a) Scroll down and view the cell titled **Create a linear regression model**, and examine the code listing below it.

Create a linear regression model

In [ ]:

```
from sklearn.linear_model import LinearRegression

lin_reg = LinearRegression(fit_intercept = False)
start = time()
lin_reg.fit(X_train, np.ravel(y_train))
end = time()
train_time = (end - start) * 1000

# Score using the test data.
score = lin_reg.score(X_test, y_test)

print('Linear regression model took {:.2f} milliseconds to fit.'.format(train_time))
print('Variance score on test set: {:.0f}%'.format(score * 100))
```

The `LinearRegression()` class is being used to fit a simple linear model. The `fit_intercept` argument determines whether or not the intercept is calculated; in this case, it is being set to `False` for the sake of simplicity.

b) Select the cell that contains the code listing, then select **Run**.

c) Examine the output.

```
Linear regression model took 25.64 milliseconds to fit.
Variance score on test set: 91%
```

By default, the `score()` method for a `LinearRegression()` object returns the R^2 value of the prediction, also known as the coefficient of determination. It summarizes the amount of variance that the model explains; so, in this case, 91% of the variance in the dependent variable (`EnergyOutput`) is explainable.

Since there are better ways of measuring a model's skill—by using a cost function to measure prediction error, for example—you won't be using R^2 as the desired metric.

9. Compare the first ten predictions to actual values.

a) Scroll down and view the cell titled **Compare the first ten predictions to actual values**, and examine the code listing below it.

This code will generate new columns for the predicted and actual energy output, and show a sample of ten records for comparison.

b) Select the cell that contains the code listing, then select **Run**.

c) Examine the output.

	Temperature	ExhaustVacuum	AmbientPressure	RelativeHumidity	PredictedEnergyOutput	ActualEnergyOutput
7466	25.68	70.32	1009.08	90.48	436.20	431.32
8561	17.94	62.10	1019.81	82.65	457.51	453.55
7777	10.60	41.17	1018.38	87.92	474.29	478.47
176	12.28	40.55	1005.72	98.56	464.26	472.47
6342	9.75	52.72	1026.03	78.53	477.28	473.41
74	10.25	41.46	1018.67	84.41	475.27	479.28
2155	23.73	63.94	1010.70	87.10	442.35	441.78
9057	16.84	39.63	1004.67	82.98	457.90	466.06
1675	24.78	68.63	1013.96	43.70	445.15	448.43
771	25.72	59.21	1012.37	66.62	443.16	437.72

10. Compare the error between predicted and actual values.

a) Scroll down and view the cell titled **Calculate the error between predicted and actual values**, and examine the code listing below it.

b) Select the cell that contains the code listing, then select **Run**.

c) Examine the output.

```
Cost (mean squared error): 25.87
```

The result is the mean squared error (MSE) for the model. The lower the MSE, the higher the model's predictive skill. Different models based on different datasets will lead to different MSE values, so there is not one objectively "correct" value to shoot for.

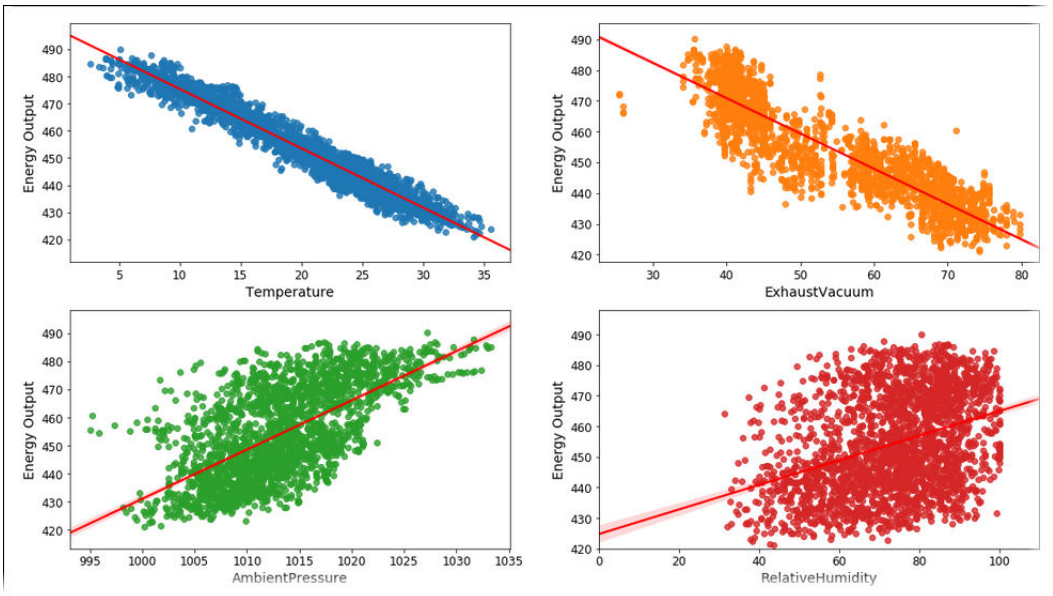
11. Plot lines of best fit for all features.

a) Scroll down and view the cell titled **Plot lines of best fit for all features**, and examine the code listing below it.

This code will produce four plots enabling you to view the lines of best fit for temperature, exhaust vacuum, ambient pressure, and relative humidity.

b) Select the cell that contains the code listing, then select **Run**.

c) Examine the output.



- Each feature is plotted against the `EnergyOutput` label, with a line of best fit generated from the linear regression.
- Note that the appearance of each scatter plot seems to align with the correlations identified earlier. For example, `Temperature` and `ExhaustVacuum` both exhibited high negative correlation with the `EnergyOutput`, so their data points support a straight line fit rather well. On the other hand, `RelativeHumidity` had a much lower correlation, and its data points are scattered in such a way that a straight line does not fit as tightly.

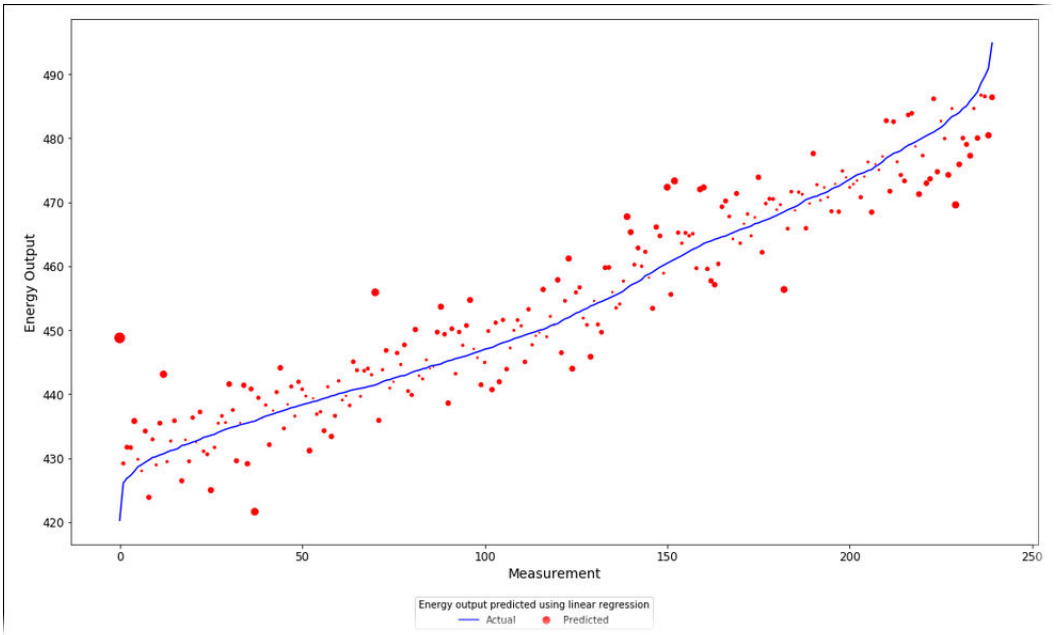
12. Compare predicted values to actual values.

a) Scroll down and view the cell titled **Compare predicted values to actual values**, and examine the code listing below it.

This code will generate a combination line chart and scatter plot to visually compare the predicted values to actual values.

b) Select the cell that contains the code listing, then select **Run**.

c) Examine the output.



- The dataset has been sorted by the actual energy output per hourly measurement, which is shown as the blue line.
- The predicted energy output at each hourly measurement is shown as a red dot in a scatter plot. The larger the dot, the higher the amount of error between predicted and actual values.
- For the most part, the predicted output corresponds rather closely to the actual output.

13. Retrieve the model parameters.

a) Scroll down and view the cell titled **Retrieve the model parameters**, and examine the code listing below it.

This code will print out the model parameters that the linear regression model uses to make its predictions.

b) Select the cell that contains the code listing, then select **Run**.

c) Examine the output.

```
Temperature coefficient: -1.6668474737282652
ExhaustVacuum coefficient: -0.27545747113877944
AmbientPressure coefficient: 0.5025758077163675
RelativeHumidity coefficient: -0.09689119750793702
```

These coefficients are the parameters that were derived from the linear regression model. They are used by the model to make predictions on input data.

14. Manually calculate the model parameters using the normal equation.

a) Scroll down and view the cell titled **Manually calculate the model parameters using the normal equation**, and examine the code listing below it.

This code manually forms the normal equation that is used in simple linear regression.

b) Select the cell that contains the code listing, then select **Run**.

c) Examine the output.

```
[[-1.66684747]
 [-0.27545747]
 [ 0.50257581]
 [-0.0968912 ]]
```

As you can see, plugging the training data into the normal equation gives you the exact same model parameters. This is because the `LinearRegression()` class uses the normal equation to generate a predictive model.

15. Shut down the notebook and close the lab.

a) From the menu, select **Kernel→Shutdown**.
b) In the **Shutdown kernel?** dialog box, select **Shutdown**.
c) Close the lab browser tab and continue on with the course.

MODULE 2

Build Regularized and Iterative Linear Regression Models

The following labs are for Module 2: Build Regularized and Iterative Linear Regression Models.

LAB 3-2
Building a Regularized Linear Regression Model

Data File

~/Linear Regression/LinearRegression-Boston.ipynb

Scenario

You want to apply your house valuation models to more than just the examples provided in the King County dataset. You also have access to a dataset that includes data about houses in Boston, Massachusetts. Once again, you want to be able to train a model to predict the value of a home in this area given several factors. However, simple linear regression is not necessarily the best way to address this problem. You want to avoid running into issues where the model overfits to the training data, making it less useful in generalizing to new data. So, you'll apply the technique of regularization to your linear models to help simplify the models and avoid overfitting. Rather than just arbitrarily choosing one type of regularization, you'll evaluate all three (ridge, lasso, and elastic net), and then choose which one performs the best according to your requirements.

---

1. Open the lab and notebook.
   a) From the Coursera lab page, select **Open Lab** to open a Jupyter Notebook session in another web browser tab.
   b) Select **Linear Regression/LinearRegression-Boston.ipynb** to open it.

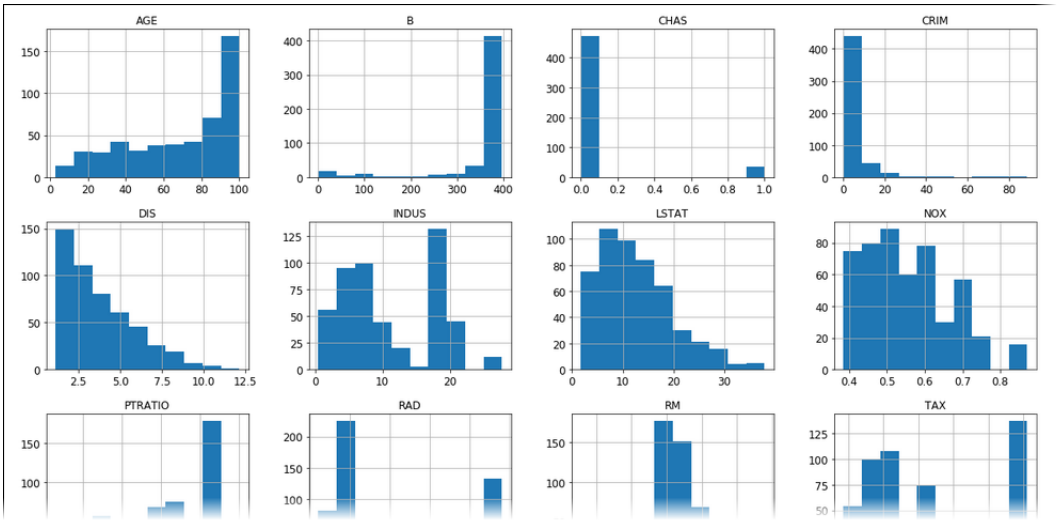
2. Import the relevant libraries and load the dataset.
   a) Select the code cell beneath the **Import software libraries and load the dataset** title, then select **Run**.
   b) Verify that 506 records were loaded.

      This dataset actually comes pre-packaged with scikit-learn, and can be easily loaded into an array through the `load_boston()` function. There's no need to load a file stored on the local machine.

3. Get acquainted with the dataset.
   a) Scroll down and select the code cell beneath the **Get acquainted with the dataset** title, then select **Run**.
   b) Examine the output.
      - The training set includes 506 rows and 14 columns.
      - All of the columns contain float values.
      - There is no missing data; all rows have values for every column.
      - Each column measures some factor about a house in Boston. Some examples include:
        - `CRIM` is the per-capita crime rate of the area.
        - `CHAS` refers to the "Charles River dummy variable"—if 1, the land bounds the river; if 0, it does not.
        - `NOX` is the level of nitrogen oxides (NO_2) in the area. High levels of NO_2 can cause health issues.
        - `RM` is the average number of rooms per house.
        - `AGE` is the proportion of occupied units built before 1940.
        - `DIS` is the weighted mean distance to several employment centers around Boston.
        - `TAX` is the property tax rate per $10,000.
        - `PTRATIO` is the ratio of pupils (students) to teachers in the school district.
        - `target` also refers to `MEDV`, the median value of the house in thousands of dollars.

Note: If you're curious about the rest of the features in this dataset, consider going to: **https://www.cs.toronto.edu/~delve/data/boston/bostonDetail.html**.

4. Examine the distribution of various features.
   a) Select the code cell beneath the **Examine the distribution of various features** title, then select **Run**.
   b) Examine the output.



Some highlights include:

- The age of the houses seems to be left-skewed, indicating that most houses in the dataset are quite old.
- The mean distance to employment centers is right-skewed, indicating that most houses are close to such centers.
- The NO_2 levels seem to fluctuate, though most tend to be on the low end.
- The number of rooms seems to form a roughly symmetrical distribution.
- The median house value (i.e., `target`) also seems to have a reasonably symmetrical distribution, with perhaps a few high outliers.

5. Examine a general summary of statistics.
   a) Select the code cell beneath the **Examine a general summary of statistics** title, then select **Run**.

b) Examine the output.

```
        CRIM     ZN  INDUS   CHAS    NOX     RM    AGE    DIS    RAD    TAX  \
count 506.00 506.00 506.00 506.00 506.00 506.00 506.00 506.00 506.00 506.00
mean    3.61  11.36  11.14   0.07   0.55   6.28  68.57   3.80   9.55 408.24
std     8.60  23.32   6.86   0.25   0.12   0.70  28.15   2.11   8.71 168.54
min     0.01   0.00   0.46   0.00   0.39   3.56   2.90   1.13   1.00 187.00
25%     0.08   0.00   5.19   0.00   0.45   5.89  45.02   2.10   4.00 279.00
50%     0.26   0.00   9.69   0.00   0.54   6.21  77.50   3.21   5.00 330.00
75%     3.68  12.50  18.10   0.00   0.62   6.62  94.07   5.19  24.00 666.00
max    88.98 100.00  27.74   1.00   0.87   8.78 100.00  12.13  24.00 711.00

       PTRATIO      B  LSTAT  target
count   506.00 506.00 506.00  506.00
mean     18.46 356.67  12.65   22.53
std       2.16  91.29   7.14    9.20
min      12.60   0.32   1.73    5.00
25%      17.40 375.38   6.95   17.02
50%      19.05 391.44  11.36   21.20
75%      20.20 396.23  16.96   25.00
max      22.00 396.90  37.97   50.00
```

The feature scales seem to be disproportionate, especially when comparing lower-scale features like `CRIM` (whose mean is 3.61) and higher-scale features like `TAX` (whose mean is 408.24). Unlike with simple linear regression, regularized linear regression will benefit from some feature scaling. When features are scaled, the regularization penalty can be applied equally to the data, rather than giving undue weight to certain features. Scaling can also help reduce the effect of outliers on regularized linear regression.

6. Look for columns that correlate with `target` (median house value).

a) Select the code cell beneath the **Look for columns that correlate with `target` (median house value)** title, then select **Run**.

b) Examine the output.

```
Correlations with median house value
target     1.000000
RM         0.695360
ZN         0.360445
B          0.333461
DIS        0.249929
CHAS       0.175260
AGE       -0.376955
RAD       -0.381626
CRIM      -0.388305
NOX       -0.427321
TAX       -0.468536
INDUS     -0.483725
PTRATIO   -0.507787
LSTAT     -0.737663
Name: target, dtype: float64
```

It looks like many of these features have at least some correlation with the median house value, whether positive or negative. The lowest correlation is `CHAS`, the categorical variable that indicates whether a house's property borders the Charles River. Because the correlation appears to be weak, and because the feature is categorical and not numeric (like the other features), you'll drop this feature from training.

7. Split the label from the dataset.

a) Select the code cell beneath the **Split the label from the dataset** title, then select **Run**.

b) Examine the output.

Rather than using the holdout method to split the datasets into training set and validation/test set, you'll be training the data using cross-validation. Cross-validation can help improve the model's ability to generalize to new data. For now, you're just extracting the label from the training set (`X`) and placing it in its own vector (`y`).

8. Drop columns that won't be used for training.

a) Select the code cell beneath **Drop columns that won't be used for training**, then select **Run**.

b) Examine the output and observe that the `CHAS` column was dropped.

```
Columns before drop:

['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE', 'DIS', 'RAD', 'TAX', 'PTRATIO', 'B', 'LSTAT']

Columns after drop:

['CRIM', 'ZN', 'INDUS', 'NOX', 'RM', 'AGE', 'DIS', 'RAD', 'TAX', 'PTRATIO', 'B', 'LSTAT']
```

9. Standardize the features.

a) Scroll down and view the cell titled **Standardize the features**, and examine the code listing below it.

Standardize the features

In [ ]:

```
def standardize(X):
    result = X.copy()

    for feature in X.columns:
        result[feature] = (X[feature] - X[feature].mean()) / X[feature].std()  # z-score formula.

    return result

X = standardize(X)

print('The features have been standardized.')
```

This function includes the z-score formula, which is used to standardize the data.

b) Select the cell that contains the code listing, then select **Run**.

```
The features have been standardized.
```

c) Scroll down and select the next code listing cell.

```
with pd.option_context('float_format', '{:.2f}'.format):
    print(X.describe())
```

d) Select **Run**.

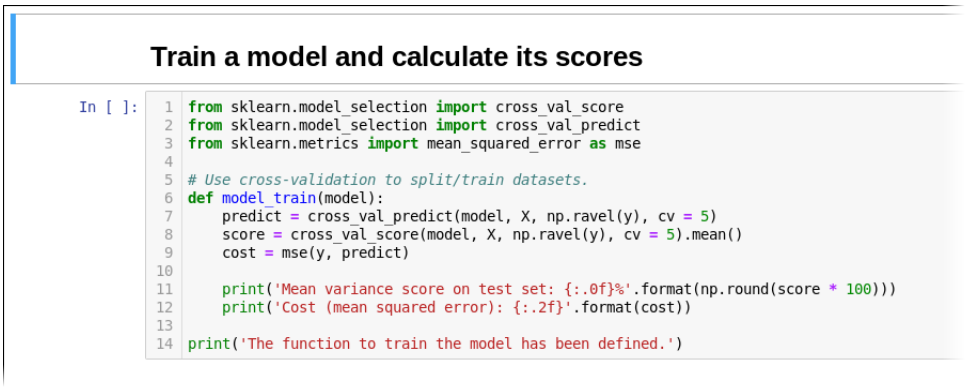
e) Examine the output.

```
        CRIM     ZN  INDUS    NOX     RM    AGE    DIS    RAD    TAX  PTRATIO  \
count 506.00 506.00 506.00 506.00 506.00 506.00 506.00 506.00 506.00   506.00
mean    0.00   0.00  -0.00   0.00  -0.00  -0.00   0.00  -0.00   0.00    -0.00
std     1.00   1.00   1.00   1.00   1.00   1.00   1.00   1.00   1.00     1.00
min    -0.42  -0.49  -1.56  -1.46  -3.88  -2.33  -1.27  -0.98  -1.31    -2.70
25%    -0.41  -0.49  -0.87  -0.91  -0.57  -0.84  -0.80  -0.64  -0.77    -0.49
50%    -0.39  -0.49  -0.21  -0.14  -0.11   0.32  -0.28  -0.52  -0.46     0.27
75%     0.01   0.05   1.01   0.60   0.48   0.91   0.66   1.66   1.53     0.81
max     9.92   3.80   2.42   2.73   3.55   1.12   3.96   1.66   1.80     1.64

           B  LSTAT
count 506.00 506.00
mean    0.00  -0.00
std     1.00   1.00
min    -3.90  -1.53
25%     0.20  -0.80
50%     0.38  -0.18
75%     0.43   0.60
max     0.44   3.55
```

Each feature has been transformed to have a mean value of 0 and a standard deviation of 1. As a result, the values have been scaled down.

10. Train the model and calculate its scores.

a) Scroll down and view the cell titled **Train a model and calculate its scores**, and examine the code listing below it.



Train a model and calculate its scores

In [ ]:

```
1  from sklearn.model_selection import cross_val_score
2  from sklearn.model_selection import cross_val_predict
3  from sklearn.metrics import mean_squared_error as mse
4
5  # Use cross-validation to split/train datasets.
6  def model_train(model):
7      predict = cross_val_predict(model, X, np.ravel(y), cv = 5)
8      score = cross_val_score(model, X, np.ravel(y), cv = 5).mean()
9      cost = mse(y, predict)
10
11     print('Mean variance score on test set: {:.0f}%'.format(np.round(score * 100)))
12     print('Cost (mean squared error): {:.2f}'.format(cost))
13
14 print('The function to train the model has been defined.')
```

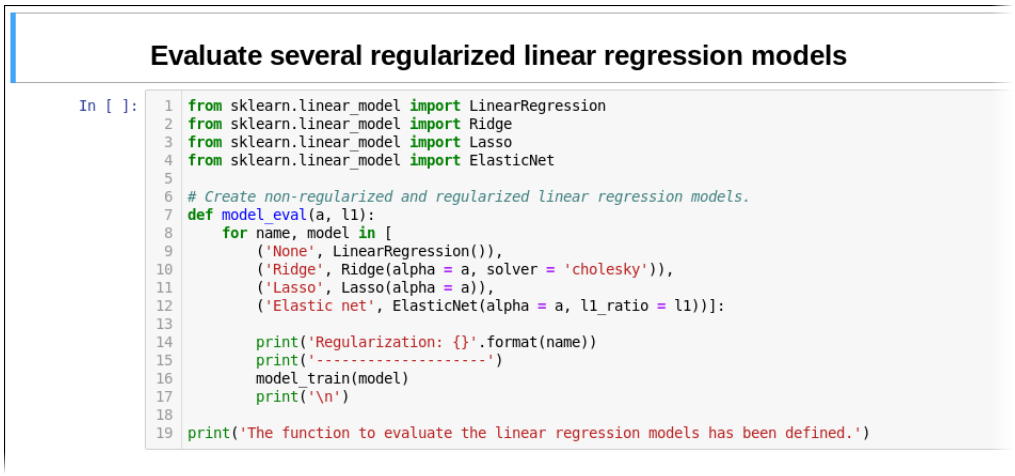
This function both performs cross-validation on the dataset and trains the model:

- On line 6, the function takes in `model` as an argument, which is the class object of whatever machine learning algorithm you wish to use.
- On line 7, rather than using the standard `fit()` method to begin training, the `cross_val_prediction()` method performs both cross-validation and the actual training on the data it splits. The first argument it takes is the model, and the next two arguments are the datasets to train the model on (`X` and `y`).
- The `cv` argument specifies the number of folds to use in cross-validation. In other words, by supplying an integer, k-fold cross-validation will automatically be performed. Specifically, stratified k-fold is used as the default. You can also specify a different kind of cross-validation (like LOOCV), but for this dataset, stratified k-fold appears to be adequate.
- On line 8, the `cross_val_score()` method returns the R^2 score for the training, much like the standard `score()` method. However, it computes a score for all of the folds, so those five scores are being averaged to produce the overall score.
- On line 9, the mean squared error (MSE) for the predictions is being calculated.
- Line 14 prints a message to indicate the function has been defined.

b) Select the cell that contains the code listing, then select **Run**.

11. Evaluate several regularized linear regression models.

a) Scroll down and view the cell titled **Evaluate several regularized linear regression models**, and examine the code listing below it.



Evaluate several regularized linear regression models

In [ ]:

```
1  from sklearn.linear_model import LinearRegression
2  from sklearn.linear_model import Ridge
3  from sklearn.linear_model import Lasso
4  from sklearn.linear_model import ElasticNet
5
6  # Create non-regularized and regularized linear regression models.
7  def model_eval(a, l1):
8      for name, model in [
9          ('None', LinearRegression()),
10         ('Ridge', Ridge(alpha = a, solver = 'cholesky')),
11         ('Lasso', Lasso(alpha = a)),
12         ('Elastic net', ElasticNet(alpha = a, l1_ratio = l1))]:
13
14         print('Regularization: {}'.format(name))
15         print('--------------------')
16         model_train(model)
17         print('\n')
18
19 print('The function to evaluate the linear regression models has been defined.')
```

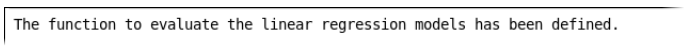
This function calls `model_train()` using four different algorithms:

- Simple linear regression (no regularization).
- Ridge regression.
- Lasso regression.
- Elastic net regression.

In addition:

- On line 7, the function takes two arguments: `a` is the regularization hyperparameter (i.e., λ) that is applied to all regularization algorithms; and `l1` is the regularization penalty ratio to use for elastic net. The lower this value, the closer the penalty is to the ℓ_2 norm (ridge); the higher the value, the closer the penalty is to the ℓ_1 norm (lasso).
- The `solver` argument for ridge regression refers to how the algorithm will minimize the cost function. In this case, `cholesky` is just the standard closed-form solution (the normal equation).

b) Select the cell that contains the code listing, then select **Run**.



```
The function to evaluate the linear regression models has been defined.
```

By placing this code in a function, it can easily be repeated multiple times for comparison, passing in different parameters.

c) Scroll down and select the next code listing cell.

In [ ]:

```
1 model_eval(1, 0.5)
```

d) Select **Run**.

e) Examine the output.

```
1 model_eval(1, 0.5)
```

```
Regularization: None
--------------------
Mean variance score on test set: 34%
Cost (mean squared error): 36.63


Regularization: Ridge
--------------------
Mean variance score on test set: 35%
Cost (mean squared error): 36.37


Regularization: Lasso
--------------------
Mean variance score on test set: 32%
Cost (mean squared error): 40.50


Regularization: Elastic net
--------------------
Mean variance score on test set: 33%
Cost (mean squared error): 39.97
```

- In this function call, you used an arbitrary starting point of 1 as the regularization hyperparameter.
- You used an elastic net penalty of 0.5, which favors neither the ℓ_1 nor ℓ_2 norm.
- For these hyperparameters, ridge regression appears to have both the highest variance score and the lowest MSE. The latter is of particular interest, as this is what you're trying to minimize.
- Lasso and elastic net regression seem to have performed worse than the simple, non-regularized linear model.
- You'll reconfigure these hyperparameters to hopefully get better results.

f) Scroll down and select the next code listing cell.

```
In [ ]: 1 model_eval(0.1, 0.3)
```

g) Select **Run**.

h) Examine the output.

```
model_eval(0.1, 0.3)
```

```
Regularization: None
--------------------
Mean variance score on test set: 34%
Cost (mean squared error): 36.63


Regularization: Ridge
--------------------
Mean variance score on test set: 34%
Cost (mean squared error): 36.60


Regularization: Lasso
--------------------
Mean variance score on test set: 39%
Cost (mean squared error): 35.53


Regularization: Elastic net
--------------------
Mean variance score on test set: 42%
Cost (mean squared error): 33.68
```

- In this function call, you changed the regularization hyperparameter to 0.1.
- You changed the elastic net penalty to 0.3, which favors the ℓ_2 norm (ridge).
- For these hyperparameters, elastic net regression appears to have the lowest MSE—even lower than the ridge regression model from the previous training round. The improvement may be slight, but you won't always see massive gains through regularization.

Note: Learning how to select the optimal hyperparameters is covered in the next course.

12. Are you satisfied with this MSE value? In other words, would you stop there and finalize the model? Why or why not?

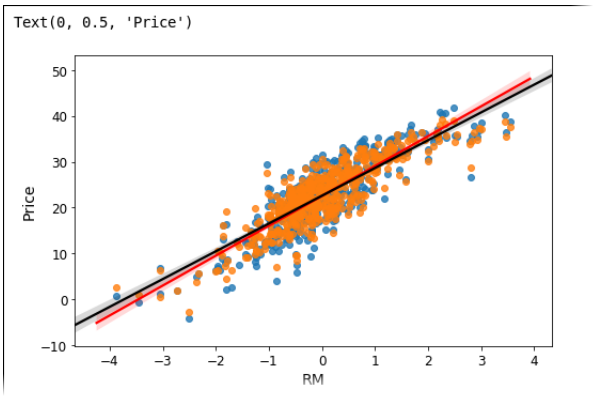
13. Plot lines of best fit for the `RM` (average number of rooms) feature.

a) Scroll down and view the cell titled **Plot lines of best fit for the `RM` (average number of rooms) feature**, and examine the code listing below it.

This code will generate a chart showing the line of best fit for the linear regression model.

b) Select the cell that contains the code listing, then select **Run**.

c) Examine the output.



- `RM` is being demonstrated here because it had one of the strongest correlations with the median house value.
- The red line is the unregularized line of best fit, whereas the black line is the line of best fit using elastic net with the hyperparameters that were deemed "good enough."
- The blue dots are the unregularized predictions, whereas the orange dots are the regularized predictions.
- As you can see, regularized regression produced slightly different predictions with a slightly different line of best fit, which reflects the evaluation you did earlier.

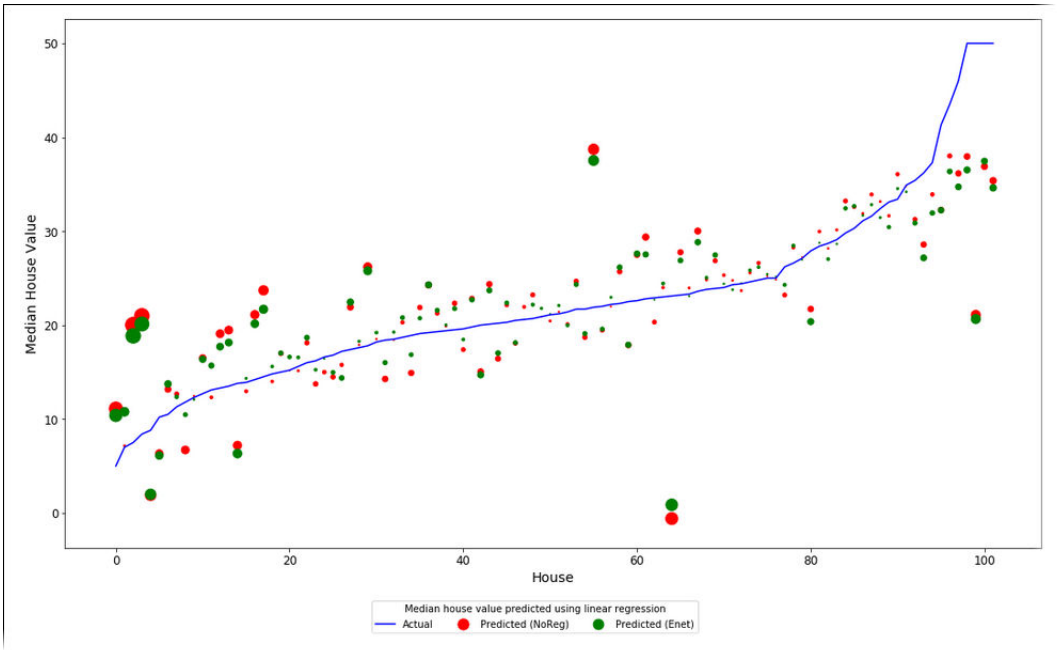
14. Compare predicted values to actual values.

a) Scroll down and view the cell titled **Compare predicted values to actual values**, and examine the code listing below it.

This code will generate a combination line chart and scatter plot to visually compare the predicted values to actual values.

b) Select the cell that contains the code listing, then select **Run**.

c) Examine the output.



- The dataset has been sorted by the actual median house value, which is shown as the blue line.
- The predicted value of a house using unregularized linear regression is shown as a red dot in a scatter plot. The larger the dot, the higher the amount of error between predicted and actual values.
- The predicted value of a house using elastic net regression is shown as a green dot in a scatter plot.
- Each model gives slightly different predictions, and some predictions are closer to the target than others.

15. Shut down the notebook and close the lab.

a) From the menu, select **Kernel→Shutdown**.
b) In the **Shutdown kernel?** dialog box, select **Shutdown**.
c) Close the lab browser tab and continue on with the course.

LAB 3-3
Building an Iterative Linear Regression Model

Data File

~/Linear Regression/LinearRegression-Boston-Iterative.ipynb

Scenario

As you continue expanding the scope of your real estate models, you realize that you'll quickly be training on very large datasets—datasets with hundreds of thousands of examples, and hundreds or perhaps thousands of features. The typical closed-form solution of the normal equation will start being a burden on training performance, slowing down the process to a potentially unacceptable degree. So, you'll try implementing an alternative cost minimization technique like gradient descent to hopefully cut down on training time. You'll begin by retraining your Boston housing model.

---

1. Open the lab and notebook.
   a) From the Coursera lab page, select **Open Lab** to open a Jupyter Notebook session in another web browser tab.
   b) Select **Linear Regression/LinearRegression-Boston-Iterative.ipynb** to open it.

2. Import the relevant libraries and load the dataset.
   a) Select the code cell beneath the **Import software libraries and load the dataset** title, then select **Run**.
   b) Verify that 506 records were loaded.

3. Split the datasets.
   a) Scroll down and select the code cell beneath the **Split the datasets** title, and examine the code listing below it.
   b) Select **Run**.

      You'll be training the following models using a holdout set rather than cross-validation. This is for demonstration purposes, as the stochastic nature of cross-validation in scikit-learn makes it more difficult to compare the results of the two algorithms you'll use in this lab.

4. Drop columns that won't be used for training.
   a) Scroll down and view the cell titled **Drop columns that won't be used for training**, and examine the code listing below it.
   b) Select the cell that contains the code listing, then select **Run**.

5. Standardize the features.
   a) Scroll down and select the code cell beneath the **Standardize the features** title, then select **Run**.

      The data is being prepared in the same way as before. The z-score is applied to the features in order to scale their values.

6. **Why is it important to scale down features, such as through standardization, when using an iterative cost minimization technique like gradient descent?**

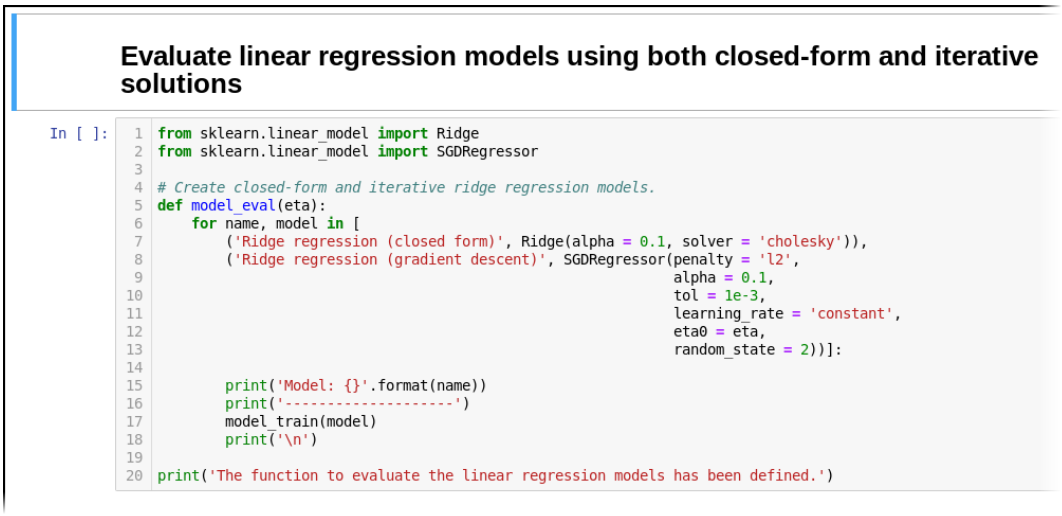
7. Train a model and calculate its scores.

a) Scroll down and select the code cell beneath the **Train a model and calculate its cost** title, then select **Run**.

This is the same `model_train()` function as before; the only differences are that each model's training will be timed, and the variance score is not used.

8. Evaluate linear regression models using both closed-form and iterative solutions.

a) Scroll down and view the code cell beneath the **Evaluate linear regression models using both closed-form and iterative solutions** title.



Evaluate linear regression models using both closed-form and iterative solutions

In [ ]:

```
from sklearn.linear_model import Ridge
from sklearn.linear_model import SGDRegressor

# Create closed-form and iterative ridge regression models.
def model_eval(eta):
    for name, model in [
        ('Ridge regression (closed form)', Ridge(alpha = 0.1, solver = 'cholesky')),
        ('Ridge regression (gradient descent)', SGDRegressor(penalty = 'l2',
                                                             alpha = 0.1,
                                                             tol = 1e-3,
                                                             learning_rate = 'constant',
                                                             eta0 = eta,
                                                             random_state = 2))]:

        print('Model: {}'.format(name))
        print('----------------------')
        model_train(model)
        print('\n')

print('The function to evaluate the linear regression models has been defined.')
```

As before, this function calls `model_train()` using multiple algorithms:

- The ridge regression algorithm. This model uses the closed-form solution to minimize the cost function, with an arbitrary regularization strength (`alpha`) value.
- A similar linear algorithm, but one that uses stochastic gradient descent (SGD) to minimize the cost function.

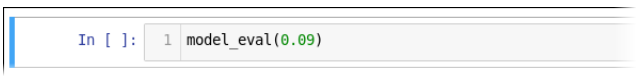
For the `SGDRegressor()` algorithm, the following hyperparameters are configured:

- The `penalty` is the type of regularization to use. The objective is to try to compare the speed of each model using the same or very similar settings, so the ℓ_2 norm (ridge) is being used.
- The `alpha` value is the same so that the regularization strength of both models is comparable.
- The `tol` argument is the stopping criterion; when a value is supplied, the iterations will stop when the amount of loss is greater than the optimal loss minus the `tol` value. The default is 1e–3, which is supplied here.
- The `learning_rate` is actually the scheduling technique used to configure the learning rate during descent. Setting this to `constant` means that the learning rate will stay at its initial value (`eta0`) and won't change. The learning rate determines the number of steps in gradient descent, and the more steps the descent takes, the more time it takes to train the model.
- The `eta0` argument is the initial learning rate. This is what you'll supply to the `model_eval()` function as a way to tune your model.

b) Select the cell that contains the code listing, then select **Run**.

9. Continue evaluating the linear regression models using different initial learning rates.

a) Scroll down and select the next code listing cell.



In [ ]:

```
model_eval(0.09)
```

b) Select **Run**.

c) Examine the output.

```
Model: Ridge regression (closed form)
--------------------
Linear regression model took 5.35 milliseconds to fit.
Cost (mean squared error): 21.88


Model: Ridge regression (gradient descent)
--------------------
Linear regression model took 1.63 milliseconds to fit.
Cost (mean squared error): 65.36
```

- In this function call, you used an arbitrary starting point of 0.09 as the initial learning rate (`eta0`).
- The mean squared error (MSE) for the SGD model appears significantly worse.
- Because the dataset is so small, the time to fit will vary. On larger datasets, where calculating the normal equation consumes too much memory, the SGD model should take less time to fit.

d) Scroll down and select the next code listing cell.

```
In [ ]: 1 model_eval(0.08)
```

e) Select **Run**.

f) Examine the output.

```
Model: Ridge regression (closed form)
--------------------
Linear regression model took 2.56 milliseconds to fit.
Cost (mean squared error): 21.88


Model: Ridge regression (gradient descent)
--------------------
Linear regression model took 1.53 milliseconds to fit.
Cost (mean squared error): 47.05
```

- In this function call, you decreased the learning rate slightly.
- The MSE for the SGD model seems to have improved, but is still worse than the closed-form model.

g) Scroll down and select the next code listing cell.

```
In [ ]: 1 model_eval(0.05)
```

h) Select **Run**.

i) Examine the output.

```
Model: Ridge regression (closed form)
--------------------
Linear regression model took 2.12 milliseconds to fit.
Cost (mean squared error): 21.88


Model: Ridge regression (gradient descent)
--------------------
Linear regression model took 1.09 milliseconds to fit.
Cost (mean squared error): 31.89
```

- The SGD model seems to be gradually improving as you progressively decrease the learning rate.

j) Scroll down and select the next code listing cell.

k) Select **Run**.

l) Examine the output.

```
Model: Ridge regression (closed form)
--------------------
Linear regression model took 2.21 milliseconds to fit.
Cost (mean squared error): 21.88


Model: Ridge regression (gradient descent)
--------------------
Linear regression model took 1.09 milliseconds to fit.
Cost (mean squared error): 25.25
```

- By lowering the learning rate significantly, the SGD model is getting close to the closed-form solution in terms of its ability to minimize error.

10. You could continue to tune the learning rate, but the SGD model will not lead to a lower error than the closed-form solution.

    Why is this?

11. Shut down the notebook and close the lab.

    a) From the menu, select **Kernel→Shutdown**.

    b) In the **Shutdown kernel?** dialog box, select **Shutdown**.

    c) Close the lab browser tab and continue on with the course.

MODULE 3

Train Classification Models

The following labs are for Module 3: Train Classification Models.

LAB 3-4
Training Binary Classification Models

Data Files

~/Classification/Classification-Titanic.ipynb

~/Classification/titanic_data/test.csv

~/Classification/titanic_data/train.csv

Scenario

You work for the History department at a state college. An upcoming lecture focuses on one of the most well-known disasters in history—the sinking of the RMS *Titanic*. The department wants to teach students about the multiple factors that may have influenced who survived and who didn't. Instead of just lecturing students on the subject matter and having them accept it passively, you want them to be able to reach their own conclusions and verify those conclusions through hands-on experience. So, you decide to build a machine learning model that will help students do just that.

You'll use a real-world dataset that includes various statistics about the *Titanic* passengers. The label in this case is whether or not the passenger survived. So, you'll train a couple classification models that will try to predict who survived and who didn't. Ultimately, you want students to be able to feed the model their own data—for example, everyone in the class could be a "passenger" with their own characteristics (age, sex, number of siblings, etc.)—so that you can show them whether or not *they* would survive the disaster. This will hopefully make the lecture a little more "real" to the students, while also being a fun academic exercise.

While you plan to make interacting with the machine learning model more user friendly at some point, you first need to build the code base.

Note: This *Titanic* dataset is very commonly used to teach machine learning concepts.

---

1. Open the lab and notebook.
   a) From the Coursera lab page, select **Open Lab** to open a Jupyter Notebook session in another web browser tab.
   b) Select **Classification/Classification-Titanic.ipynb** to open it.

2. Import the relevant libraries and load the dataset.
   a) Select the code cell beneath the **Import software libraries and load the dataset** title, then select **Run**.
   b) Verify that **test.csv** and **train.csv** are in the project folder, and that the latter was loaded with 891 records.

      There are separate testing and training datasets provided. You'll split the training dataset to create a validation set as well. But first you'll get familiar with the data.

3. Get acquainted with the data.
   a) Select the code cell beneath the **Get acquainted with the dataset** title, then select **Run**.
   b) Examine the output.
      - The training set includes 891 rows and 12 columns.
      - 5 columns contain integer values, 5 contain strings (objects), and 2 contain floats.
      - Most columns have a value in every row, except for `Age` (714 records, or 177 missing), `Cabin` (204 records, or 687 missing), and `Embarked` (889 records, or 2 missing).

- Several columns are self-explanatory, but for those that aren't:
  - `Survived` with a value of 0 means the passenger did not survive; 1 means they did. This is the label of interest.
  - `Pclass` is the class of ticket. In other words, a value of 1 means the passenger was traveling first class, the most expensive and highest quality service. This also acts as a proxy for the passenger's socioeconomic status.
  - `SibSp` refers to the number of siblings plus a spouse that the passenger had onboard.
  - `Parch` refers to the number of parents and children that the passenger had onboard.
  - `Ticket` is the ticket's identification number.
  - `Fare` is the cost of the passenger's ticket, which likely correlates with `Pclass`.
  - `Cabin` refers to where on the ship the passenger lodged.
  - `Embarked` is a letter that refers to the port where the passenger embarked from: "C" for Cherbourg, "Q" for Queenstown, and "S" for Southampton.
- Several columns (`Sex`, `Ticket`, `Cabin`, `Embarked`) contain non-numeric values. These will have to be converted to numeric values somehow if they are to be used by the learning algorithm.
- Some data values shown as `NaN` (not a number) are missing, and will need to be handled somehow.
- The features have different ranges, so they will need to be scaled so that they exert the same influence over the model.
- It might be helpful to investigate the meaning (if any) of the letter/number combinations in `Ticket` and `Cabin`. However, since so many values are missing for `Cabin`, it might be necessary to just drop this feature.

4. Examine a general summary of statistics.

a) Select the code cell beneath the **Examine a general summary of statistics** title, and then select **Run**.

b) Examine the output.

	PassengerId	Survived	Pclass	Age	SibSp	Parch	Fare
count	891.00	891.00	891.00	714.00	891.00	891.00	891.00
mean	446.00	0.38	2.31	29.70	0.52	0.38	32.20
std	257.35	0.49	0.84	14.53	1.10	0.81	49.69
min	1.00	0.00	1.00	0.42	0.00	0.00	0.00
25%	223.50	0.00	2.00	20.12	0.00	0.00	7.91
50%	446.00	0.00	3.00	28.00	0.00	0.00	14.45
75%	668.50	1.00	3.00	38.00	1.00	0.00	31.00
max	891.00	1.00	3.00	80.00	8.00	6.00	512.33

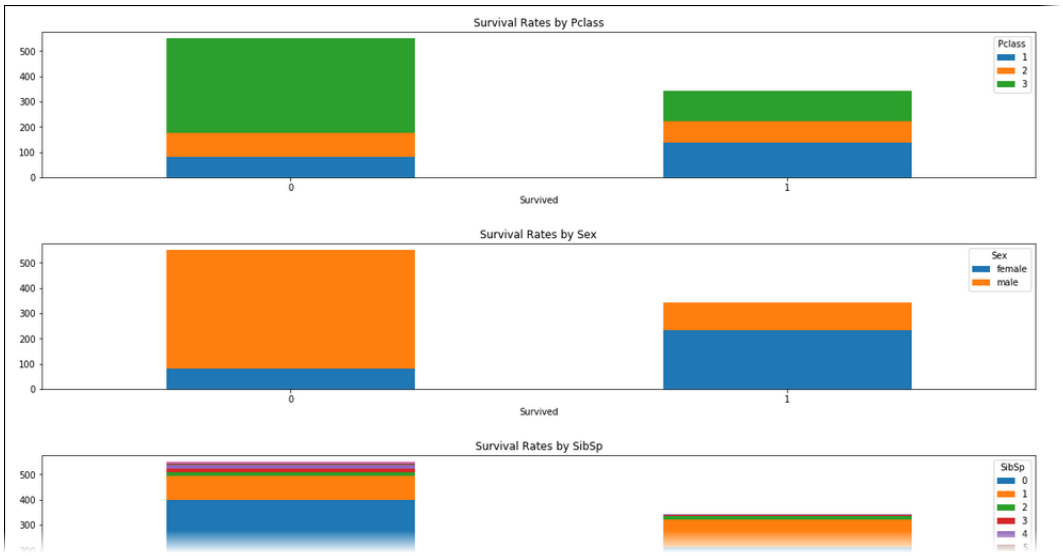
- The mean for `Survived` is 0.38. Since features in this column contain a 1 (survived) or 0 (did not survive), this means that 38% of passengers in the training set survived.
- The minimum `Age` shows that the youngest passenger in the set was less than a year old (0.42). The maximum `Age` was 80 years old.
- The mean age was `29.7` years old.

5. Given what you know about the dataset thus far, what features do you think might influence the survival rates?

6. Unlike with linear regression, attempting to find a correlation between a feature and a classification label is not useful. In other words, there's no need to run code to compare the *Titanic* features to the `Survived` label.

   Why is such a correlation not relevant in classification problems like this one?

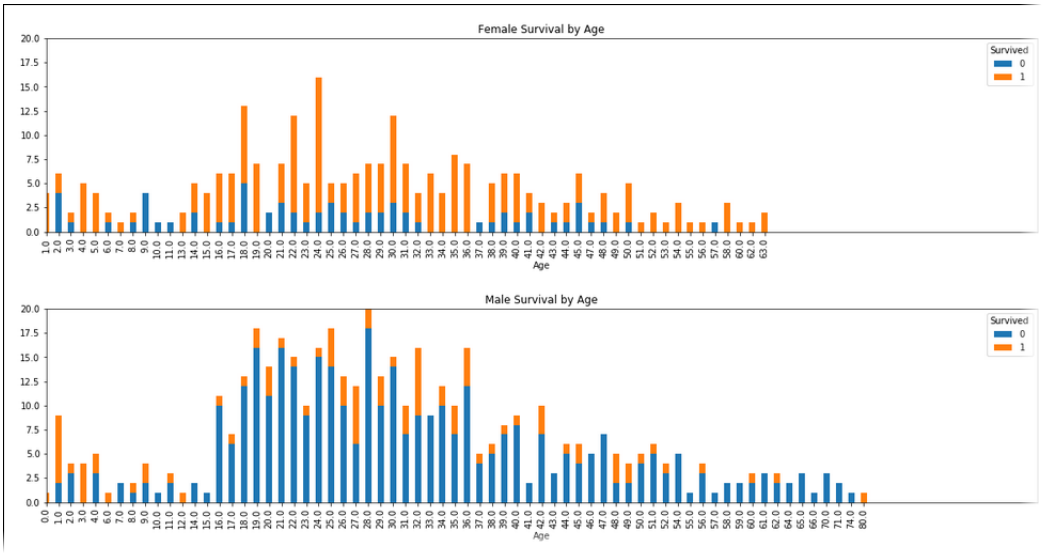
7. Use stacked bar visualization to show survival numbers.
   a) Scroll down and select the code cell beneath the **Use stacked bar visualization to show survival numbers** title, then select **Run**.
   b) Examine the output.



   - In general, passengers were more likely to perish than survive.
   - The majority of people who perished were in third class. However, keep in mind that these graphs are measuring absolute values and not proportions—there may have been more people in third class than in any other class.
   - The majority of people who perished were male.
   - The majority of people who perished had no family with them.
   - People who embarked from Cherbourg were more likely to survive than perish.
   - People who embarked from Southampton were more likely to perish than survive.

8. Look for relationships between survival, age, and sex.
   a) Scroll down and select the code cell beneath the **Look for relationships between survival, age, and sex** title, then select **Run**.

b) Examine the output.



As expected, female passengers had higher survival rates than male passengers. Likewise, fewer of the elderly survived as compared to younger passengers, though this is more obvious for male than female passengers.

9. Identify columns with missing values.

a) Scroll down and select the code cell beneath the **Identify columns with missing values** title, then select **Run**.

b) Examine the output.

```
Number of missing values:
PassengerId      0
Survived         0
Pclass           0
Name             0
Sex              0
Age            177
SibSp            0
Parch            0
Ticket           0
Fare             0
Cabin          687
Embarked         2
dtype: int64
```

`Age` and `Cabin` have many missing values that will need to be resolved somehow.

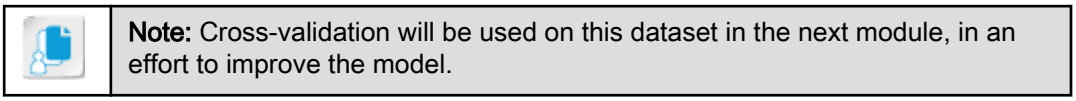
10. Split the datasets.

a) Scroll down and select the code cell beneath the **Split the datasets** title, then select **Run**.

b) Examine the output.

```
Original set:        (891, 12)
------------------------------
Training features:   (668, 11)
Validation features: (223, 11)
Training labels:     (668, 1)
Validation labels:   (223, 1)
```

The original training dataset has now been split into two: one set to continue using as training, the other to use as validation. Note that the `Survived` label has been removed from the `X` matrices and placed into its own `y` vector.



Note: Cross-validation will be used on this dataset in the next module, in an effort to improve the model.

11. Identify columns that should be modified or deleted from the training set.

a) Scroll down and select the code cell beneath the **Identify columns that should be modified or deleted from the training set** title, then select **Run**.

b) Examine the output.

	PassengerId	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
439	440	2	Kvillner, Mr. Johan Henrik Johannesson	male	31.0	0	0	C.A. 18723	10.5000	NaN	S
617	618	3	Lobb, Mrs. William Arthur (Cordelia K Stanlick)	female	26.0	1	0	A/5. 3336	16.1000	NaN	S
242	243	2	Coleridge, Mr. Reginald Charles	male	29.0	0	0	W./C. 14263	10.5000	NaN	S
82	83	3	McDermott, Miss. Brigdet Delia	female	NaN	0	0	330932	7.7875	NaN	Q
398	399	2	Pain, Dr. Alfred	male	23.0	0	0	244278	10.5000	NaN	S

- `PassengerId` and `Name` are not relevant for the model and should be dropped from the training set. However, they may be needed for reporting results, so they should stay in the validation and test sets.
- You'll convert the `Sex` and `Embarked` columns to numeric codes.
- Many of the `Name` values include an honorific (personal title) such as "Mr.", "Mrs.", "Miss.", etc. You'll extract these and convert them to numeric codes.

12. Determine how to handle ticket values.

a) Scroll down and select the code cell beneath the **Determine how to handle ticket values** title, then select **Run**.

b) Examine the output.

It seems as though it would be difficult to convert ticket codes into something useful, so they will be dropped from the training set.

13. Identify all personal titles and embarked port codes.

a) Scroll down and select the code cell beneath the **Identify all personal titles and embarked port codes** title, then select **Run**.

b) Examine the output.

```
Titles:  ['Mr' 'Mrs' 'Miss' 'Master' 'Don' 'Rev' 'Dr' 'Mme' 'Ms' 'Major' 'Lady'
 'Sir' 'Mlle' 'Col' 'Capt' 'the Countess' 'Jonkheer']
Embarked locations:  ['S' 'C' 'Q' nan]
```

You'll convert these to numeric values.

14. Perform common preparation on the training and validation sets.

a) Scroll down and view the cell titled **Perform common preparation on the training and validation sets**, and examine the code listing below it.

   Common cleaning and feature engineering tasks will be performed on the data.

b) Select the cell that contains the code listing, then select **Run**.

c) Examine the output.

   - Missing values for `Age` and `Fare` are being filled in with the median of those values. For `Embarked`, you're using the mode to fill in missing values.
   - Since `SibSp` and `Parch` seem to both be related, you're combining them into a single column, `SizeOfFamily`. Anyone traveling alone has a family size of 1.
   - If a passenger has a personal title, it is being removed from `Name` and placed into its own `Title` column.
   - Each unique `Title`, `Sex`, and `Embarked` value is being encoded with a number value, starting at 1. This will make them easier to work with.
   - Missing `TitleEncoding` values are being filled in with the mode.

15. Preview current training data.

a) Scroll down and select the code cell beneath the **Preview current training data** title, and examine the code listing below it.

b) Select the cell that contains the code listing, then select **Run**.

c) Examine the output.

   - As mentioned previously, `PassengerId` and `Name` don't contain information that will be useful for training the model.
   - `Cabin` contains coding that might be useful, but numerous records are missing this data, so you'll drop it.
   - `Ticket` likewise might contain useful encoding, but you don't have enough information to convert it to a number.
   - Some columns containing text values (`Title`, `Sex`, `Embarked`) have been replaced with a numeric code, and can be dropped.

16. Drop columns that won't be used for training.

a) Scroll down and view the cell titled **Drop columns that won't be used for training**, and examine the code listing below it.

   Unused columns will be dropped from the dataset.

b) Select the cell that contains the code listing, then select **Run**.

c) Examine the output and observe the new set of columns in the `X` matrix (for both training and validation).

```
---- TRAINING -----
Columns before drop:

['PassengerId', 'Pclass', 'Name', 'Sex', 'Age', 'SibSp', 'Parch', 'Ticket', 'Fare', 'Cabin', 'Embarked', 'SizeOfFamily',
'Title', 'SexEncoding', 'EmbarkedEncoding', 'TitleEncoding']

Columns after drop:

['Pclass', 'Age', 'SibSp', 'Parch', 'Fare', 'SizeOfFamily', 'SexEncoding', 'EmbarkedEncoding', 'TitleEncoding']

--- VALIDATION ----
Columns before drop:

['PassengerId', 'Pclass', 'Name', 'Sex', 'Age', 'SibSp', 'Parch', 'Ticket', 'Fare', 'Cabin', 'Embarked', 'SizeOfFamily',
'Title', 'SexEncoding', 'EmbarkedEncoding', 'TitleEncoding']

Columns after drop:

['Pclass', 'Age', 'SibSp', 'Parch', 'Fare', 'SizeOfFamily', 'SexEncoding', 'EmbarkedEncoding', 'TitleEncoding']
```

d) Scroll down and select the next code listing cell.

```
In [ ]:  1  X_train.head()
```

e) Select **Run**.

f) Examine the output and observe the new columns.

	Pclass	Age	SibSp	Parch	Fare	SizeOfFamily	SexEncoding	EmbarkedEncoding	TitleEncoding
439	2	31.0	0	0	10.5000	1	2	1	1
617	3	26.0	1	0	16.1000	2	1	1	2
242	2	29.0	0	0	10.5000	1	2	1	1
82	3	27.5	0	0	7.7875	1	1	3	3
398	2	23.0	0	0	10.5000	1	2	1	4

17. Create a logistic regression model.

a) Scroll down and select the code cell beneath the **Create a logistic regression model** title, then select **Run**.

b) Examine the output.

```
Logistic regression model took 573.28 milliseconds to fit.
Score on validation set: 78%
```

- `LogisticRegression()` is a scikit-learn class that, as the name implies, generates a logistic regression model object. One of the important arguments that it takes is `solver`, which determines the method used to minimize the cost function. In this case, you're using `sag`, which refers to stochastic average gradient (SAG). Recall that this is similar to stochastic gradient descent (SGD), but has a "memory" of previous gradients for faster convergence. By using this `solver`, the algorithm will automatically perform ℓ_2 regularization.
- The `C` argument is the hyperparameter that controls the strength of regularization applied to the model. For now, this is set to a relatively low value, which implements strong regularization.
- The `max_iter` argument specifies the maximum number of iterations that the `solver` will perform in an effort to converge at a minimum. In this case, the default number of iterations (100) is not enough to converge, so the maximum is set higher.
- The score for this model is around 78%.

c) Scroll down and select the next code listing cell.

```
In [ ]:  1  # Use validation set to evaluate.
         2  results_comparison = X_val.copy()
         3  results_comparison['PredictedSurvival'] = log_reg.predict(X_val)
         4  results_comparison['ActualSurvival'] = y_val.copy()
         5  results_comparison['ProbPerished'] = np.round(log_reg.predict_proba(X_val)[:, 0] * 100, 2)
         6  results_comparison['ProbSurvived'] = np.round(log_reg.predict_proba(X_val)[:, 1] * 100, 2)
         7
         8  # View examples of the predictions compared to actual survival.
         9  results_comparison.head(20)
```

d) Select **Run**.

e) Examine the output.

	Pclass	Age	SibSp	Parch	Fare	SizeOfFamily	SexEncoding	EmbarkedEncoding	TitleEncoding	PredictedSurvival	ActualSurvival	ProbPerished
400	3	39.00	0	0	7.9250	1	2	1	1	0	1	83.00
450	2	36.00	1	2	27.7500	4	2	1	1	0	0	82.71
846	3	30.25	8	2	69.5500	11	2	1	1	0	0	97.52
516	2	34.00	0	0	10.5000	1	1	1	2	0	1	51.99
42	3	30.25	0	0	7.8958	1	2	2	1	0	0	78.61
247	2	24.00	0	2	14.5000	3	1	1	2	0	1	53.42
22	3	15.00	0	0	8.0292	1	1	3	3	1	1	35.46
867	1	31.00	0	0	50.4958	1	2	1	1	0	0	63.11
410	3	30.25	0	0	7.8958	1	2	1	1	0	0	82.60
252	1	62.00	0	0	26.5500	1	2	1	1	0	0	71.82

In addition to the existing columns, the classification that was predicted by the model on the validation set is compared to the actual label value. The probabilities for each classification are also displayed in the right-most columns. These probabilities are the results that the logistic regression function generates and uses to determine what class to predict. For example, if `ProbPerished` is higher than `ProbSurvived`, then the model will have predicted a 0 for that particular passenger (i.e., did not survive).

18. Create a *k*-nearest neighbor model.

a) Scroll down and select the code cell beneath the **Create a *k*-nearest neighbor model** title, then select **Run**.

b) Examine the output.

```
Value of k: 27
KNN model took 9.14 milliseconds to fit.
Score on validation set: 76%
```

- This is an alternative to the logistic regression model built in the previous step. Both of these models perform binary classification, but it's useful to build multiple models to see if one performs better than another.
- The score for the k-nearest neighbors model is around 76%. This appears to perform worse than the logistic regression model, so you'll stick with the logistic regression model for the actual test set.
- In this case, k is generated by a bootstrapping method—taking the square root of the total number of data examples in the training set. To help avoid tie votes in a binary classifier, k is incremented by 1 if it is even.

c) Scroll down and select the next code listing cell.

```
In [ ]:
# Use validation set to evaluate.
results_comparison = X_val.copy()
results_comparison['PredictedSurvival'] = knn.predict(X_val)
results_comparison['ActualSurvival'] = y_val.copy()
results_comparison['ProbPerished'] = np.round(knn.predict_proba(X_val)[:, 0] * 100, 2)
results_comparison['ProbSurvived'] = np.round(knn.predict_proba(X_val)[:, 1] * 100, 2)

# View examples of the predictions compared to actual survival.
results_comparison.head(20)
```

d) Select **Run**.

e) Examine the output.

The information here is in the same format as the logistic regression output from before, but the predictions are different because of the different model used.

Note: Because *k*-NN does not generate prediction probabilities in the same way that logistic regression does, the `predict_proba()` method is actually returning the fraction of neighbors that voted for each label. So, in the first row of results, ~93% of the neighbors voted for perished (25 of the total 27).

19. Use the logistic regression model to make predictions on the test data.

a) Scroll down and select the code cell beneath the **Use the logistic regression model to make predictions on the test data** title, then select **Run**.

b) Examine the output.

418 records are loaded from **test.csv**. Recall that this is the test set, separate from the training and validation sets. This test set does not include label values. It's this kind of data that the machine learning model must make predictions for in a production environment. In the context of your History class, this is the set that would include students' information as if they were passengers on the *Titanic*.

c) Scroll down and select the next code cell.

```
In [ ]:
# Prepare the dataset and drop unneeded columns.
print('Preparing test data for prediction\n')
X_test = prep_dataset(X_test_raw.copy())
X_test = drop_unused(X_test.copy())
```

d) Select **Run**.

e) Examine the output.

As with the training and validation sets, you're performing the same data preparation steps to get the data in an optimal state.

f) Scroll down and select the next code listing cell.

```
In [ ]:
# Show example predictions with the original test data.
results_log_reg = X_test_raw.copy()
results_log_reg['PredictedSurvival'] = log_reg.predict(X_test)
results_log_reg['ProbPerished'] = np.round(log_reg.predict_proba(X_test)[:, 0] * 100, 2)
results_log_reg['ProbSurvived'] = np.round(log_reg.predict_proba(X_test)[:, 1] * 100, 2)
results_log_reg.head(20)
```

g) Select **Run**.

h) Examine the output.

	PassengerId	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked	PredictedSurvival	ProbPerished	ProbSurvived
0	892	3	Kelly, Mr. James	male	34.5	0	0	330911	7.8292	NaN	Q	0	74.29	25.71
1	893	3	Wilkes, Mrs. James (Ellen Needs)	female	47.0	1	0	363272	7.0000	NaN	S	0	68.02	31.98
2	894	2	Myles, Mr. Thomas Francis	male	62.0	0	0	240276	9.6875	NaN	Q	0	70.70	29.30
3	895	3	Wirz, Mr. Albert	male	27.0	0	0	315154	8.6625	NaN	S	0	82.30	17.70
4	896	3	Hirvonen, Mrs. Alexander (Helga E Lindqvist)	female	22.0	1	1	3101298	12.2875	NaN	S	0	66.31	33.69
5	897	3	Svensson, Mr. Johan Cervin	male	14.0	0	0	7538	9.2250	NaN	S	0	81.58	18.42
6	898	3	Connolly, Miss. Kate	female	30.0	0	0	330972	7.6292	NaN	Q	1	36.70	63.30
7	899	2	Caldwell, Mr. Albert Francis	male	26.0	1	1	248738	29.0000	NaN	S	0	80.96	19.04
			Abrahim, Mrs. Joseph											

The logistic regression model has predicted values for this test set, fulfilling its ultimate purpose.

20. Shut down the notebook and close the lab.

a) From the menu, select **Kernel→Shutdown**.

b) In the **Shutdown kernel?** dialog box, select **Shutdown**.

c) Close the lab browser tab and continue on with the course.

LAB 3-5
Training a Multi-Class Classification Model

Data File

~/Classification/Classification-Wine.ipynb

Scenario

Your hands-on *Titanic* lab was a success, and now other professors at the college are looking to apply machine learning to their chosen fields. The Chemistry department wants to help its students learn about the interactions between certain chemical compounds. Namely, the students should observe how slight variations in a substance's chemical makeup can change its overall form.

A professor from the Chemistry department has provided you with a dataset that lists various wines, each one with several characteristics. Each wine example is classified as a wine from a specific cultivar. However, unlike with the *Titanic* dataset, the wines can be grouped into one of *three* classes, not just two. So, a binary classification model isn't going to be enough. You'll need to build a multi-class classification model in order to predict the type of wine a particular sample is.

---

1. Open the lab and notebook.
   a) From the Coursera lab page, select **Open Lab** to open a Jupyter Notebook session in another web browser tab.
   b) Select **Classification/Classification-Wine.ipynb** to open it.

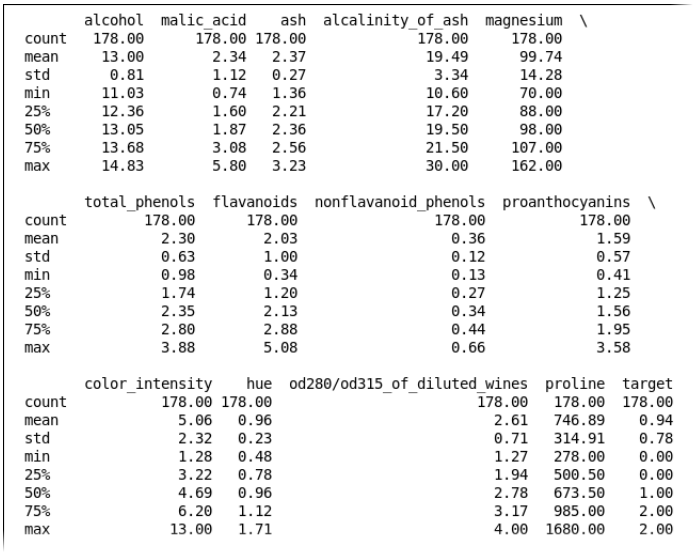
2. Import the relevant libraries and load the dataset.
   a) Scroll down and select the code cell beneath the **Import software libraries and load the dataset** title, then select **Run**.
   b) Verify that 178 records were loaded.

      This dataset actually comes pre-packaged with scikit-learn, and can be easily loaded into an array through the `load_wine()` function. There's no need to load a file stored on the local machine.

3. Get acquainted with the dataset.
   a) Scroll down and select the code cell beneath the **Get acquainted with the dataset** title, then select **Run**.
   b) Examine the output.
      - The training set includes 178 rows and 14 columns.
      - All of the columns contain float values, except for the `target` column, which contains integer values. The `target` column is the label of wine the model must predict, classified as either 0, 1, or 2. With three possible classifications, this dataset presents a multi-class problem.
      - There is no missing data; all rows have values for every column.
      - Most of the columns describe a particular wine's chemical composition, like its alcohol level, magnesium level, number of phenols, etc. Some columns describe visual aspects of the wine, like its color intensity and hue.
      - Ultimately, this is a relatively clean and uniform dataset. However, it may still benefit from a bit of preparation.

4. Examine a general summary of statistics.
   a) Scroll down and select the code cell beneath the **Examine a general summary of statistics** title, then select **Run**.

b) Examine the output.



	alcohol	malic_acid	ash	alcalinity_of_ash	magnesium	\
count	178.00	178.00	178.00	178.00	178.00	
mean	13.00	2.34	2.37	19.49	99.74	
std	0.81	1.12	0.27	3.34	14.28	
min	11.03	0.74	1.36	10.60	70.00	
25%	12.36	1.60	2.21	17.20	88.00	
50%	13.05	1.87	2.36	19.50	98.00	
75%	13.68	3.08	2.56	21.50	107.00	
max	14.83	5.80	3.23	30.00	162.00	

	total_phenols	flavanoids	nonflavanoid_phenols	proanthocyanins	\
count	178.00	178.00	178.00	178.00	
mean	2.30	2.03	0.36	1.59	
std	0.63	1.00	0.12	0.57	
min	0.98	0.34	0.13	0.41	
25%	1.74	1.20	0.27	1.25	
50%	2.35	2.13	0.34	1.56	
75%	2.80	2.88	0.44	1.95	
max	3.88	5.08	0.66	3.58	

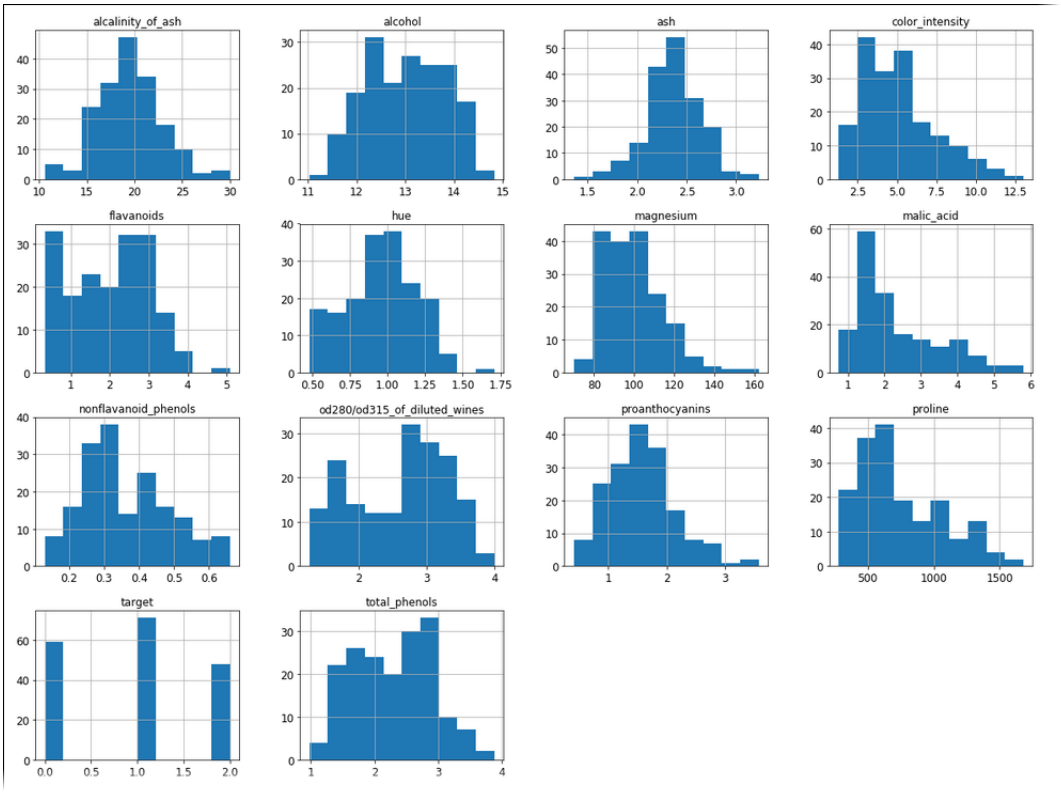
	color_intensity	hue	od280/od315_of_diluted_wines	proline	target
count	178.00	178.00	178.00	178.00	178.00
mean	5.06	0.96	2.61	746.89	0.94
std	2.32	0.23	0.71	314.91	0.78
min	1.28	0.48	1.27	278.00	0.00
25%	3.22	0.78	1.94	500.50	0.00
50%	4.69	0.96	2.78	673.50	1.00
75%	6.20	1.12	3.17	985.00	2.00
max	13.00	1.71	4.00	1680.00	2.00

- Unless you're a wine expert or knowledgeable in the chemistry of alcohol, you may not have much intuition as to which features are the most important in determining the class of wine.
- There don't appear to be many extreme outliers for any of the features, but you'll soon look at a visual distribution to be sure.
- One thing you should be able to identify is that the values in `magnesium` and especially `proline` are much higher than the others. Some scaling might be in order.

5. Examine the distribution of various features.

a) Scroll down and select the code cell beneath the **Examine the distribution of various features** title, then select **Run**.

b) Examine the output.



For the most part, these histograms seem to confirm the idea that there are not many extreme outliers in the dataset. You can also see that the class labels (`target`) are distributed pretty evenly.

6. Split the label from the dataset.

a) Scroll down and select the code cell beneath the **Split the label from the dataset** title, then select **Run**.

b) Examine the output.

Rather than using the holdout method to split the datasets into a training set and a validation/test set, you'll be training the data using cross-validation. For now, you're just extracting the label from the training set (`X`) and placing it in its own vector (`y`).

7. Transform `magnesium` and `proline`.

a) Scroll down and select the code cell beneath the **Transform `magnesium` and `proline`** title, then select **Run**.

You're transforming the two features you identified earlier as being out of scale with the rest of the data.

b) Examine the output.

Transform `magnesium` and `proline`

```
# Apply a log transformation to scale 'magnesium' and 'proline'.
X = X.copy()
X['proline'] = np.log(X['proline'])
X['magnesium'] = np.log(X['magnesium'])

# Examine results of the transformation
with pd.option_context('float_format', '{:.2f}'.format):
    print(X['magnesium'].describe())
    print('\n----------------------')
    print(X['proline'].describe())

X.head()
```

```
count   178.00
mean      4.59
std       0.14
min       4.25
25%       4.48
50%       4.58
75%       4.67
max       5.09
Name: magnesium, dtype: float64

----------------------
count   178.00
mean      6.53
std       0.42
min       5.63
25%       6.22
50%       6.51
75%       6.89
max       7.43
Name: proline, dtype: float64
```

	alcohol	malic_acid	ash	alcalinity_of_ash	magnesium	total_phenols	flavanoids	nonflavanoid_phenols	proanthocyanins	color_intensity	hue	od280/od
0	13.69	3.26	2.54	20.0	4.672829	1.83	0.56	0.50	0.80	5.88	0.96	
1	12.67	0.98	2.24	18.0	4.595120	2.20	1.94	0.30	1.46	2.62	1.23	
2	13.86	1.35	2.27	16.0	4.584967	2.98	3.15	0.22	1.85	7.22	1.01	
3	13.73	1.50	2.70	22.5	4.615121	3.00	3.25	0.29	2.38	5.70	1.19	
4	13.41	3.84	2.12	18.8	4.499810	2.45	2.68	0.27	1.48	4.28	0.91	

- A simple logarithm has helped reduced the numeric range of these values. The maximum value for `magnesium` was 162.00; now it's 5.09. The maximum value for `proline` was 1680.00; now it's 7.43.
- This feature scaling is not entirely useful for standard regression models, but it *is* useful in speeding up gradient descent and for improving regularized regression models, both of which you'll implement.

8. Create a multinomial logistic regression model.

a) Scroll down and select the code cell beneath the **Create a multinomial logistic regression model** title, then select **Run**.

Create a multinomial logistic regression model

```
from sklearn.linear_model import LogisticRegression

log_reg = LogisticRegression(solver='sag', multi_class='multinomial', max_iter=10000)

print('Multinomial logistic regression model created.')
```

```
Multinomial logistic regression model created.
```

- In scikit-learn, the same `LogisticRegression()` class is used for both binary classification and multi-class classification. The `multi_class` argument activates the latter. There are actually two different approaches; `multinomial` ensures that the softmax function is used to predict probabilities.
- The `solver` is using stochastic average gradient (SAG).

9. Train the model using stratified k-fold cross-validation to split the dataset.

a) Scroll down and select the code cell beneath the **Train the model using stratified *k*-fold cross-validation to split the dataset** title, then select **Run**.

b) Examine the output.

Train the model using stratified *k*-fold cross-validation to split the dataset

```
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import cross_val_predict

# Train model and make predictions using test data.
start = time()
predict = cross_val_predict(log_reg, X, np.ravel(y), cv = 5)
end = time()
train_time = (end - start) * 1000

# Retrieve mean score of test folds.
score = cross_val_score(log_reg, X, np.ravel(y), cv = 5).mean()

print('Multinomial logistic regression model took {:.2f} milliseconds to fit.'.format(train_time))
print('Mean score on test sets: {:.0f}%'.format(np.round(score * 100)))
```

```
Multinomial logistic regression model took 349.64 milliseconds to fit.
Mean score on test sets: 95%
```

If you hadn't transformed those two out-of-scale features, the training process would have taken around 5 times longer.

c) Scroll down and select the next code listing cell.

```
In [ ]:
# Retrieve prediction probabilities.
proba = cross_val_predict(log_reg, X, np.ravel(y), cv = 5, method = 'predict_proba')

# Use test set to evaluate.
results_comparison = X.copy()
results_comparison['magnesium'] = np.exp(results_comparison['magnesium'])
results_comparison['proline'] = np.exp(results_comparison['proline'])
results_comparison['PredictedWine'] = predict
results_comparison['ActualWine'] = y.copy()
results_comparison['ProbWine0'] = np.round(proba[:, 0] * 100, 2)
results_comparison['ProbWine1'] = np.round(proba[:, 1] * 100, 2)
results_comparison['ProbWine2'] = np.round(proba[:, 2] * 100, 2)

# View examples of the predictions compared to actual wine.
results_comparison.head(20)
```

d) Select **Run**.

e) Examine the output.

	alcohol	malic_acid	ash	alcalinity_of_ash	magnesium	total_phenols	flavanoids	nonflavanoid_phenols	proanthocyanins	color_intensity	hue	od280/od
0	13.69	3.26	2.54	20.0	107.0	1.83	0.56	0.50	0.80	5.88	0.96	
1	12.67	0.98	2.24	18.0	99.0	2.20	1.94	0.30	1.46	2.62	1.23	
2	13.86	1.35	2.27	16.0	98.0	2.98	3.15	0.22	1.85	7.22	1.01	
3	13.73	1.50	2.70	22.5	101.0	3.00	3.25	0.29	2.38	5.70	1.19	
4	13.41	3.84	2.12	18.8	90.0	2.45	2.68	0.27	1.48	4.28	0.91	
5	12.20	3.03	2.32	19.0	96.0	1.25	0.49	0.40	0.73	5.50	0.66	
6	13.83	1.65	2.60	17.2	94.0	2.45	2.99	0.22	2.29	5.60	1.24	
7	12.69	1.53	2.26	20.7	80.0	1.38	1.46	0.58	1.62	3.05	0.96	
8	14.75	1.73	2.39	11.4	91.0	3.10	3.69	0.43	2.81	5.40	1.25	
9	12.43	1.53	2.29	21.5	86.0	2.74	3.15	0.39	1.77	3.94	0.69	
10	13.05	5.80	2.13	21.5	86.0	2.62	2.65	0.30	2.01	2.60	0.73	
11	13.71	5.65	2.45	20.5	95.0	1.68	0.61	0.52	1.06	7.70	0.64	

- The `magnesium` and `proline` features you transformed earlier have now been scaled back to their original values.
- The predictions and actual values for each wine are displayed as their own columns.
- Each of the three classes has its own `ProbWine#` column, indicating what proportion of those three classes the model felt was most likely for a given example. The classification prediction is derived from the class with the highest probability.

10. Shut down the notebook and close the lab.

a) From the menu, select **Kernel→Shutdown**.
b) In the **Shutdown kernel?** dialog box, select **Shutdown**.
c) Close the lab browser tab and continue on with the course.

MODULE 4

Evaluate and Tune Classification Models

The following labs are for Module 4: Evaluate and Tune Classification Models.

LAB 3-6
Evaluating a Classification Model

Data Files

~/Classification/Classification-Titanic-Tuned.ipynb

~/Classification/titanic_data/test.csv

~/Classification/titanic_data/train.csv

Scenario

Your *Titanic* classifier was a hit with students, but it still has room for improvement. You don't just want the model to make predictions, you want it to make *good* predictions. So, to improve the model's skill, you'll evaluate its performance in a variety of ways. Later, you'll tune the model based on your findings and expectations. This will hopefully lead to an optimized classifier that you and your students can be more confident in using.

---

1. Open the lab and notebook.
   a) From the Coursera lab page, select **Open Lab** to open a Jupyter Notebook session in another web browser tab.
   b) Select **Classification/Classification-Titanic-Tuned.ipynb** to open it.

2. Import the relevant libraries and load the dataset.
   a) Select the code cell beneath the **Import software libraries and load the dataset** title, then select **Run**.
   b) Verify that **test.csv** and **train.csv** are in the project folder, and that the latter was loaded with 891 records.

3. Split the datasets.
   a) Scroll down and select the code cell beneath the **Split the datasets** title, then select **Run**.
      This splits the dataset the same way as before.

4. Perform common preparation on the training and validation sets.
   a) Scroll down and select the code cell beneath the **Perform common preparation on the training and validation sets** title, then select **Run**.
      The data is being prepared in the same way as before.

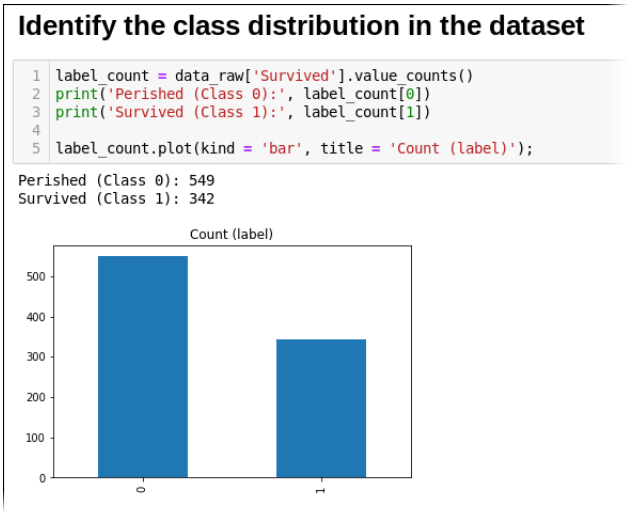
5. Drop columns that won't be used for training.
   a) Scroll down and select the code cell beneath the **Drop columns that won't be used for training** title, then select **Run**.
      Irrelevant data is being dropped in the same way as before.

6. Create a logistic regression model.
   a) Scroll down and select the code cell beneath the **Create a logistic regression model** title, then select **Run**.
   b) Verify that the score on this model is 78%.
      This is the same model that you created in the earlier lab. A score of 78% is not necessarily the best you can get out of the data, and the "score" itself is not the only way to assess the model, so you'll do some additional analysis.

7. Identify the class distribution in the dataset.

   a) Scroll down and select the code cell beneath the **Identify the class distribution in the dataset** title, then select **Run**.

   b) Examine the output.



   Since there were more passengers who perished than survived, the dataset appears to have a slight class imbalance. This can influence how useful certain metrics are in assessing the skill of the classification model. However, the imbalance is not necessarily significant.

8. Generate a confusion matrix.

   a) Scroll down and select the code cell beneath the **Generate a confusion matrix** title, then select **Run**.

   This function is defined, but hasn't been called yet. When called, the function will use Seaborn to plot a heatmap of the predicted label values and actual label values. The heatmap also doubles as a confusion matrix.

9. Compute accuracy, precision, recall, and F_1 score.

   a) Scroll down and select the code cell beneath the **Compute accuracy, precision, recall, and F_1 score** title, then select **Run**.

   This function, when called, will compute four statistical measures for the model: accuracy, precision, recall, and F_1 score. Note that the "score" you've calculated thus far for logistic regression (by calling `score()` on a model) is the same as the accuracy.

10. Generate a ROC curve and compute the AUC.

   a) Scroll down and select the code cell beneath the **Generate a ROC curve and compute the AUC** title, then select **Run**.

   This function, when called, will plot a ROC curve. It will also compute the area under that curve, which is a common method for summarizing a ROC curve.

11. Generate a precision–recall curve and compute the average precision.

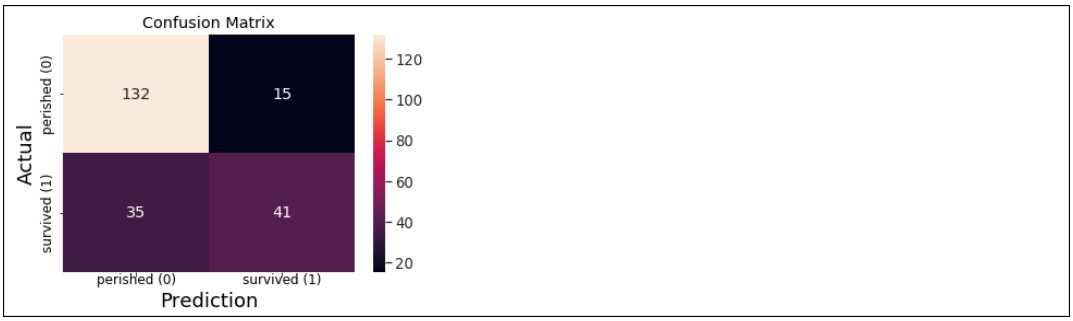
   a) Scroll down and select the code cell beneath the **Generate a precision–recall curve and compute the average precision** title, then select **Run**.

   This function, when called, will plot a precision–recall curve. It will also compute the average precision score, which is a common method for summarizing a precision–recall curve.

12. Evaluate the initial logistic regression model.

   a) Scroll down and select the code cell beneath the **Evaluate the initial logistic regression model** title, then select **Run**.

b) Examine the output.



To help orient you, each quadrant is plotted as follows:

- Top-left (**132**): *True negatives*
- Top-right (**15**): *False positives*
- Bottom-left (**35**): *False negatives*
- Bottom-right (**41**): *True positives*

13. While addressing the unique problem your classification model is trying to solve, it helps to contextualize confusion matrix results in real-world terms. When you have a fundamental understanding of the data and what you're expected to get out of it, you will make better decisions about what metrics to focus on improving.

What does each quadrant indicate in terms of predicting survivors of the *Titanic*?

14. Continue evaluating the initial logistic regression model.

a) Scroll down and select the next code listing cell.

```
In [ ]: 1 model_scores(y_val, initial_predict)
```

b) Select **Run**.

c) Examine the output.

```
Accuracy: 78%
Precision: 73%
Recall: 54%
F1: 62%
```

The familiar accuracy score is printed, as are precision, recall, and F_1 scores for this model.

15. **In what situation are precision and recall a better measure of a model's skill than accuracy?**

16. Think about the problem unique to this *Titanic* dataset. In particular, consider the problem as it pertains to the model's accuracy, precision, recall, and F_1 score.

    Is there any one of these measures you'd be more interested in optimizing than the others? Why or why not?

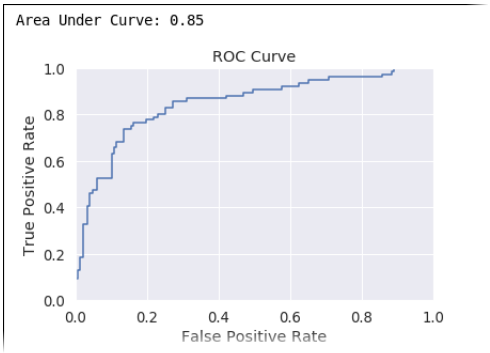
17. Continue evaluating the initial logistic regression model.

    a) Scroll down and select the next code listing cell.

```
In [ ]:
1 initial_predict_proba = log_reg.predict_proba(X_val)
2
3 roc(y_val, initial_predict_proba[:, 1])
```

    b) Select **Run**.

    c) Examine the output.



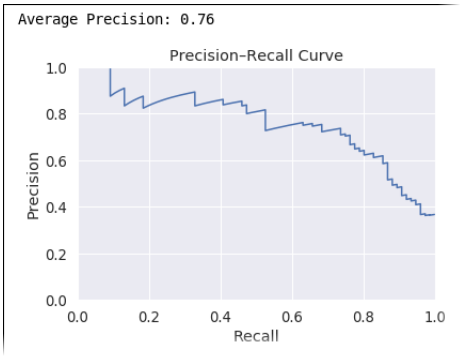
    The ROC curve is above and to the left of graph, indicating that the model is performing better than a random guess. The AUC also confirms this, as it is well above 0.5.

    d) Scroll down and select the next code listing cell.

```
In [ ]:
1 prc(y_val, initial_predict_proba[:, 1])
```

e) Select **Run**.

f) Examine the output.



The precision–recall curve is above and to the right of the no-skill line, indicating that the model is performing well with regard to the precision–recall tradeoff. The average precision confirms this.

18. **What are the advantages of a ROC curve over a precision–recall curve, and vice versa? Given your domain knowledge, are you more interested in improving one of these curves over the other? Why or why not?**

19. Save and close the lab.

a) From the menu, select **File→Save and Checkpoint**.

b) Close the lab browser tab and continue on with the course.

LAB 3-7
Tuning a Classification Model

Data Files

~/Classification/Classification-Titanic-Tuned.ipynb

~/Classification/titanic_data/test.csv

~/Classification/titanic_data/train.csv

Scenario

Now that you've assessed your *Titanic* model's skill, you'll attempt to improve that skill through hyperparameter optimization. Then, you'll evaluate the tuned model to verify whether or not it has actually improved, and how.

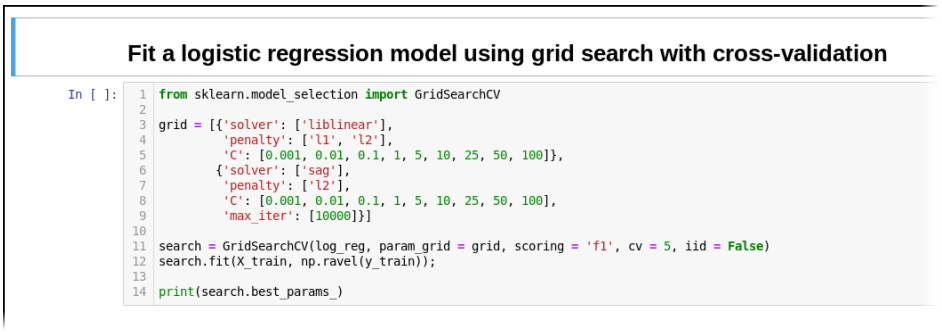
---

1. Open the lab and return to where you were in the notebook.
   a) From the Coursera lab page, select **Open Lab** to open a Jupyter Notebook session in another web browser tab.
   b) Open **Classification/Classification-Titanic-Tuned.ipynb**.
   c) Your progress from the previous lab should have been saved.

Note: If you encounter errors when running the next code blocks, select the **Fit a logistic regression model using grid search with cross-validation** heading, then select **Cell→Run All Above**.

2. Fit a logistic regression model using grid search with cross-validation.

a) Scroll down and view the cell titled **Fit a logistic regression model using grid search with cross-validation**, and examine the code listing below it.



The `grid` array holds the hyperparameters that grid search will use to train and evaluate the model. The array is actually divided into two dictionaries in order to keep valid arguments together, and to keep the grid search from trying arguments that are incompatible with one another. Each dictionary is as follows:

- Dictionary 1:
  - The `solver` is `liblinear`, which uses an iterative method called coordinate descent to solve classification problems. Compared to gradient descent, coordinate descent only updates one parameter at a time.
  - Both ℓ_1 and ℓ_2 regularization will be tried (`penalty`).
  - Various values for `C` (regularization strength) will be tried.
- Dictionary 2:
  - The `solver` uses stochastic average gradient (SAG).
  - Only ℓ_2 regularization will be tried, as SAG does not support ℓ_1 regularization.
  - The same values for `C` will be tried.
  - A maximum of 10,000 iterations will be tried in every case.

When `GridSearchCV()` is called:

- The hyperparameter array is passed in with the `param_grid` argument.
- The `scoring` argument is what the grid search is optimizing for. In this case, you're optimizing for F_1 score. You could optimize for whichever metric you'd prefer.
- By providing a `cv` value of 5, the dataset will be split using stratified k-fold cross-validation, where k is 5.

b) Select the cell that contains the code listing, then select **Run**.

This will take a minute or two.

c) Examine the output.

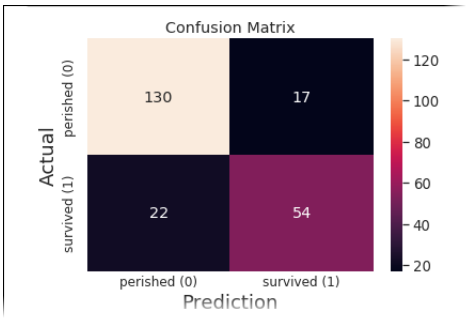
```
{'C': 10, 'penalty': 'l1', 'solver': 'liblinear'}
```

Grid search identified these hyperparameters as being the best combination for optimizing the F_1 score on the model.

3. Use a confusion matrix to evaluate the optimized model.

a) Scroll down and select the code cell beneath the **Evaluate the optimized model** title, then select **Run**.

b) Examine the output.



4. **Compared to the initial model, how has the confusion matrix changed for the optimized model?**

5. Continue evaluating the optimized model.

a) Scroll down and select the next code listing cell.

```
In [ ]: 1 model_scores(y_val, search_predict)
```

b) Select **Run**.

c) Examine the output.

```
Accuracy: 83%
Precision: 76%
Recall: 71%
F1: 73%
```

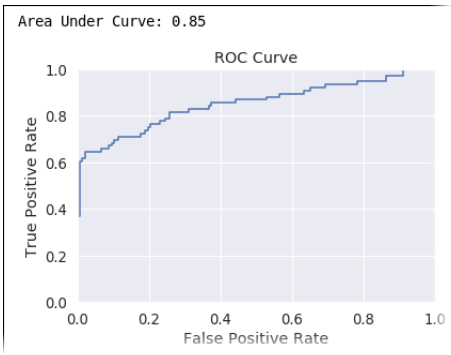
As you can see, all of the scores for this optimized model have improved. In particular, the recall and F_1 have improved the most, with the former increasing by 17%, and the latter increasing by 11%.

d) Scroll down and select the next code listing cell.

```
In [ ]: 1 search_predict_proba = search.predict_proba(X_val)
        2
        3 roc(y_val, search_predict_proba[:, 1])
```

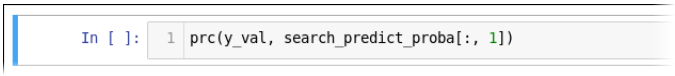
e) Select **Run**.

f) Examine the output.



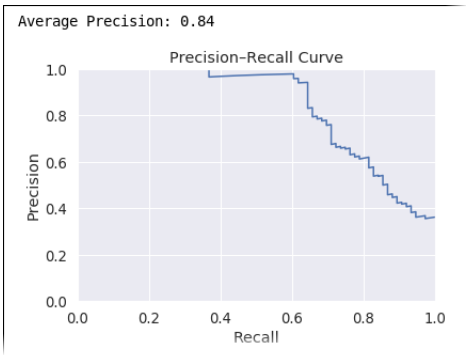
The ROC curve itself has changed, but the AUC is essentially the same as it was in the unoptimized model. You typically won't be able to improve every metric, which is why it's important to focus on the metrics you think are most relevant.

g) Scroll down and select the next code listing cell.



h) Select **Run**.

i) Examine the output.



Once again, the curve has changed. However, the average precision statistic appears to have improved by about 8%.

6. Most of the metrics have improved in the model that was optimized with grid search. However, this doesn't mean that you've automatically built the best possible classifier for this particular dataset.

   What else might you do to continue improving your classification performance for this dataset?

7. Compare the logistic regression models' predictions on the test data.

a) Scroll down and select the code cell beneath the **Compare the logistic regression models' predictions on the test data** title, then select **Run**.
b) Verify that the **test.csv** file was loaded.
c) Scroll down and select the next code listing cell.

```
In [ ]:
# Show example predictions with the test data using the initial (unoptimized) model.
results_log_reg = X_test_raw.copy()
results_log_reg['PredictedSurvival'] = log_reg.predict(X_test)
results_log_reg['ProbPerished'] = np.round(log_reg.predict_proba(X_test)[:, 0] * 100, 2)
results_log_reg['ProbSurvived'] = np.round(log_reg.predict_proba(X_test)[:, 1] * 100, 2)
results_log_reg.head(10)
```

d) Select **Run**, then verify that the initial model made predictions for the first ten test examples.
e) Scroll down and select the next code listing cell.

```
In [ ]:
# Show example predictions with the test data using the model optimized with grid search.
results_log_reg = X_test_raw.copy()
results_log_reg['PredictedSurvival'] = search.predict(X_test)
results_log_reg['ProbPerished'] = np.round(search.predict_proba(X_test)[:, 0] * 100, 2)
results_log_reg['ProbSurvived'] = np.round(search.predict_proba(X_test)[:, 1] * 100, 2)
results_log_reg.head(10)
```

f) Select **Run**, then verify that the optimized model made predictions for the first ten test examples.
g) Compare both results and verify that the optimized model made different predictions on the same test set.

8. Shut down the notebook and close the lab.

a) From the menu, select **Kernel→Shutdown**.
b) In the **Shutdown kernel?** dialog box, select **Shutdown**.
c) Close the lab browser tab and continue on with the course.

MODULE 5

Build Clustering Models

The following labs are for Module 5: Build Clustering Models.

LAB 3-8
Building a *k*-Means Clustering Model

Data Files

~/Clustering/Clustering-Housing.ipynb

~/Clustering/housing_data/kc_house_data.csv

Scenario

A real estate company that's been working with the King County dataset has prospective buyers. Each buyer identifies which houses they are most interested in. However, due to certain circumstances—like a house being pulled from the market—they may not be able to get their top choices. So, the real estate agent needs to be able to recommend to them one or more houses that are similar to their top choices. Since there are so many factors that go into a buyer's choice, what is defined as "similar" is not easy to determine.

Earlier, you were able to train a machine learning model to predict the prices of houses. This was a supervised problem, because there was a target variable/label involved (price). However, datasets like these can be applicable to many problems, not just one. So, in this case, you want to be able to group, or cluster, houses together based on the similarity of their features. Because there is no label to predict, this presents an unsupervised problem—one you'll use *k*-means clustering to address.

---

1. Open the lab and notebook.
   a) From the Coursera lab page, select **Open Lab** to open a Jupyter Notebook session in another web browser tab.
   b) Select **Clustering/Clustering-Housing.ipynb** to open it.

2. Import the relevant libraries and load the dataset.
   a) Select the code cell beneath the **Import software libraries** title, and then select **Run**.
   b) Select the code cell beneath the **Load the dataset** title, then select **Run**.
   c) Verify that **kc_house_data.csv** was loaded with 21,613 records and 21 columns.

```
Data files in this project: ['kc_house_data.csv']
Loaded 21613 records from ./housing_data/kc_house_data.csv.

Dataset Rows and Columns: (21613, 21)
```

	id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	view	...	grade	sqft_above	sqft_basement	y
0	7129300520	20141013T000000	221900.0	3	1.00	1180	5650	1.0	0	0	...	7	1180	0	
1	6414100192	20141209T000000	538000.0	3	2.25	2570	7242	2.0	0	0	...	7	2170	400	
2	5631500400	20150225T000000	180000.0	2	1.00	770	10000	1.0	0	0	...	6	770	0	
3	2487200875	20141209T000000	604000.0	4	3.00	1960	5000	1.0	0	0	...	7	1050	910	
4	1954400510	20150218T000000	510000.0	3	2.00	1680	8080	1.0	0	0	...	8	1680	0	

5 rows × 21 columns

3. Engineer features as needed.
   a) Select the code cell beneath the **Engineer features as needed** title, then select **Run**.

b) Scroll the table to the right as needed to verify that the `price_per_sqft` feature was created from `price` and `sqft_living`.

Engineer features as needed

```
# Create a price per square foot feature based on 'price' and 'sqft_living'.
housing_data['price_per_sqft'] = housing_data['price'] / housing_data['sqft_living']
housing_data.head(5)
```

sqft_lot	floors	waterfront	view	...	sqft_above	sqft_basement	yr_built	yr_renovated	zipcode	lat	long	sqft_living15	sqft_lot15	price_per_sqft
5650	1.0	0	0	...	1180	0	1955	0	98178	47.5112	-122.257	1340	5650	188.050847
7242	2.0	0	0	...	2170	400	1951	1991	98125	47.7210	-122.319	1690	7639	209.338521
10000	1.0	0	0	...	770	0	1933	0	98028	47.7379	-122.233	2720	8062	233.766234
5000	1.0	0	0	...	1050	910	1965	0	98136	47.5208	-122.393	1360	5000	308.163265
8080	1.0	0	0	...	1680	0	1987	0	98074	47.6168	-122.045	1800	7503	303.571429

4. Use a k-means model to label every row in the dataset.

a) Scroll down and view the cell titled **Use a *k*-means model to label every row in the dataset**, and examine the code listing below it.

Use a *k*-means model to label every row in the dataset

In [ ]:

```
1  from sklearn.cluster import KMeans
2
3  # Produce cluster labels.
4  def get_cluster_labels(cluster_count, X):
5
6      kmeans = KMeans(n_clusters = cluster_count,
7                      init = 'k-means++',
8                      random_state = 42)
9
10     kmeans.fit(X)
11     cluster_labels = kmeans.predict(X)
12
13     # Return the original DataFrame with the labels appended as a new column.
14     return cluster_labels
15
16 print('The function to produce cluster labels using a k-means model has been defined.')
```

- This function, when called, will create a k-means clustering model object using the number of clusters specified in the call.
- The `init` parameter of the `KMeans()` class determines how the model initializes centroids. The `k-means++` method is an improvement over the traditional k-means method, as it initializes centroids that are far away from each other, leading to better clustering.
- The function will return a `DataFrame` of the training dataset with a new column of cluster labels added.
- The print statement in line 16 shows that the function has been defined. (The function will be called later.)

b) Select the cell that contains the code listing, then select **Run**.

```
The function to produce cluster labels using a k-means model has been defined.
```

5. Generate the cluster labels and attach them to the original dataset.

a) Select the code cell beneath the **Generate the cluster labels and attach them to the original dataset** title, then select **Run**.

b) Scroll the table to the right and examine the output.

Generate the cluster labels and attach them to the original dataset

```
1  # Initially cluster by latitude and longitude, just to verify we're using k-means model correctly.
2  feature_set_1 = ['lat', 'long']
3  X = housing_data[feature_set_1]
4
5  # Generate cluster labels for the housing data, assuming 4 clusters.
6  cluster_labels = get_cluster_labels(4, X)
7
8  # Append the cluster labels to a new column in the original dataset.
9  labeled_houses = housing_data.assign(c_label = cluster_labels)
10
11 # Show a preview of rows in the dataset with cluster labels added.
12 labeled_houses.head()
```

ving	sqft_lot	floors	waterfront	view	...	sqft_basement	yr_built	yr_renovated	zipcode	lat	long	sqft_living15	sqft_lot15	price_per_sqft	c_label
180	5650	1.0	0	0	...	0	1955	0	98178	47.5112	-122.257	1340	5650	188.050847	1
570	7242	2.0	0	0	...	400	1951	1991	98125	47.7210	-122.319	1690	7639	209.338521	3
770	10000	1.0	0	0	...	0	1933	0	98028	47.7379	-122.233	2720	8062	233.766234	3
960	5000	1.0	0	0	...	910	1965	0	98136	47.5208	-122.393	1360	5000	308.163265	1
680	8080	1.0	0	0	...	0	1987	0	98074	47.6168	-122.045	1800	7503	303.571429	2

- This code just uses two features—latitude and longitude of the houses—to perform the clustering on. This is just a preliminary step to see how the clustering works.
- On line 6, the code uses an arbitrary number of clusters (4) to call the clustering function with.
- The output shows which cluster each data example is placed in: 0, 1, 2, or 3.

6. Observe how many houses were distributed to each cluster.

a) Select the code cell beneath the **Observe how many houses were distributed to each cluster** title, then select **Run**.

b) Examine the output.

Observe how many houses were distributed to each cluster

```
1  for i in range(4):
2      num_in_clust = len(labeled_houses[labeled_houses['c_label'] == i])
3      print('Number of houses in cluster {} = {}'.format(i, num_in_clust))
```

```
Number of houses in cluster 0 = 4273
Number of houses in cluster 1 = 5104
Number of houses in cluster 2 = 4810
Number of houses in cluster 3 = 7426
```

Each cluster contains several thousand houses, with cluster 3 having the most.

7. Show clusters of homes on the map by location.

a) Scroll down and view the cell titled **Show clusters of homes on the map by location**. Examine lines 1 through 22 in the code listing below it.

```
1  from folium.plugins import HeatMap
2
3  def show_on_map(labeled_house_dataset, map_title):
4
5      # To avoid overwhelming the visualization tool, we'll only plot every nth house.
6      n_homes = 20
7      mapping_set = labeled_house_dataset.sort_values(by = ['price'], ascending = False)[::n_homes]
8
9      # Descriptions of the building grades used in King County.
10     bldg_grades = ['Unknown', 'Cabin', 'Substandard', 'Poor', 'Low', 'Fair',
11                    'Low Average', 'Average', 'Good', 'Better',
12                    'Very Good', 'Excellent', 'Luxury', 'Mansion', 'Exceptional Properties']
13
14     # Generate the base map, centering on King County.
15     base_map = folium.Map(location = [47.5300, -122.2000],
16                           control_scale = True,
17                           max_zoom = 20,
18                           zoom_start = 10,
19                           zoom_control = True)
20
21     # Get price of most expensive house.
22     max_price = labeled_house_dataset.loc[labeled_house_dataset['price'].idxmax()]['price']
23
```

- The `show_on_map` function plots every 20th data point on a geographic map of the area, along with the clusters of that data.
- Lines 15 through 19 use the Folium library to display the geographic map.
- Line 22 defines the highest price of the houses in the dataset to use in scaling the rest of the points.

b) Examine lines 24 through 58.

```
24     # Plot homes by price.
25     for index, row in mapping_set.iterrows():
26
27         # Add popup text. Click each point to show details.
28         popup_text = '<br>'.join(['King County Housing Sales Data',
29                                   'Price: ${:,.0f}',
30                                   'Cluster: {:.0f}',
31                                   'Bedrooms: {:.0f}',
32                                   'Bathrooms: {:.0f}',
33                                   'Sqft Living: {:,.0f}',
34                                   'Location: [{:.3f},{:.3f}]'])
35
36         popup_text = popup_text.format(row['price'],
37                                        row['c_label'],
38                                        row['bedrooms'],
39                                        row['bathrooms'],
40                                        row['sqft_living'],
41                                        row['lat'],
42                                        row['long'])
43
44         cluster_value = int(row['c_label'])
45         scaling_value = (row['price'] / max_price)      # 1.0 for highest price.
46
47         folium.CircleMarker([row['lat'], row['long']],
48                             radius = 25 * scaling_value,
49                             weight = 3,
50                             fill = True,
51                             fill_color = '#000000',
52                             color = '#0000FF',
53                             fill_opacity = 0.8,
54                             opacity = 0.8,
55                             popup = popup_text).add_to(base_map)
56
57     # Heat map around each cluster.
58     cluster_max = labeled_house_dataset.loc[labeled_house_dataset['c_label'].idxmax()]['c_label'] + 1
```

- Line 25 begins a `for` loop that will iterate through every 20th row of the dataset, and will be used to plot the data for the houses.
- Lines 28 through 55 create the markers and popup text for the data points.

c) Examine lines 60 through 80.

```
60      for cluster_num in range(0, cluster_max):
61
62          houses_in_same_cluster = labeled_house_dataset.loc[labeled_house_dataset['c_label'] == cluster_num]
63          house_locations = houses_in_same_cluster[['lat', 'long']].copy()
64
65          mean_price = houses_in_same_cluster['price_per_sqft'].mean()
66
67          cluster_name = f'Cluster {cluster_num} (mean ${mean_price:,.0f} per sqft)'
68          feature_group = folium.FeatureGroup(name = cluster_name)
69          feature_group.add_child(HeatMap(house_locations, radius = 10))
70          base_map.add_child(feature_group)
71
72      folium.map.LayerControl('bottomright', collapsed = False).add_to(base_map)
73
74      # Add title to map.
75      map_html = f'<div style="position:fixed; top:10px; left:60px; z-index:9999"><b>{map_title}</b></div>'
76      base_map.get_root().html.add_child(folium.Element(map_html))
77
78      return base_map
79
80  print('The function to show the map has been defined.')
```

- The `for` loop that begins on line 60 iterates through each cluster.
- Lines 62 through 70 create the heat map and display it along the dimensions of latitude and longitude, using the mean price per square feet of each cluster as a summary statistic.
- The entire function returns the map when it is called.
- The print statement in line 80 shows that the function has been defined.

d) Select the cell that contains the code listing, then select **Run**.

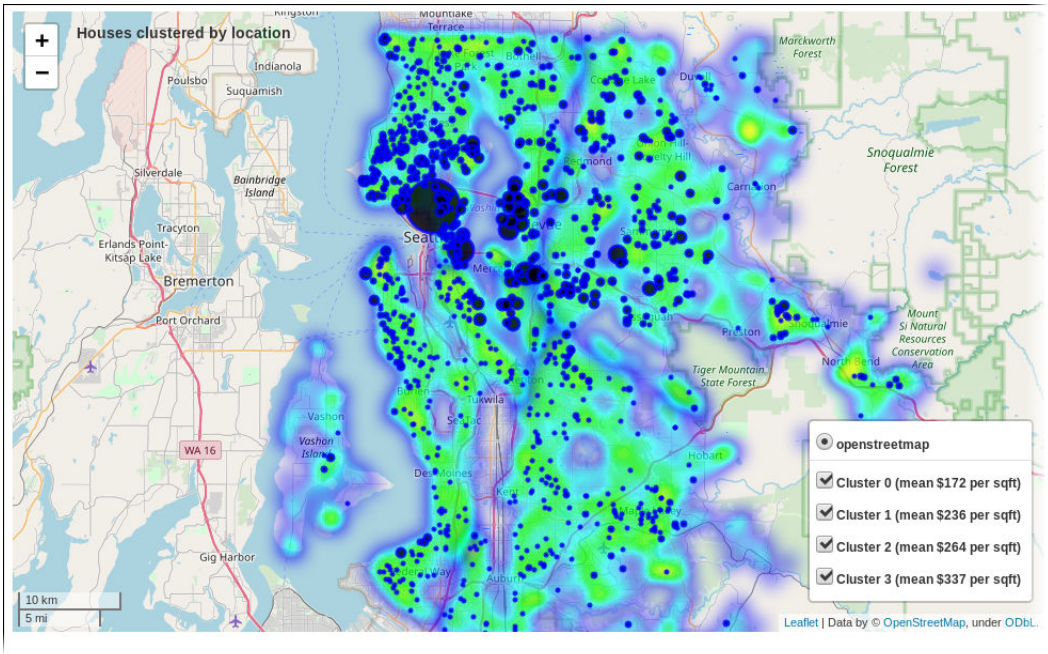
```
The function to show the map has been defined.
```

e) Scroll down and select the next code listing cell.

```
In [ ]:  1  # View the results on the map.
         2  show_on_map(labeled_houses, 'Houses clustered by location')
```

f) Select **Run**.

g) Examine the output.



There are four clusters, each grouped by house location (latitude and longitude). You can check and uncheck the different clusters to display and hide those cluster heat maps, respectively.

8. How did the clusters form with regard to location?

9. Cluster by price per square foot.

a) Scroll down and view the cell titled **Cluster by price per square foot**, and examine the code listing below it.

Cluster by price per square foot

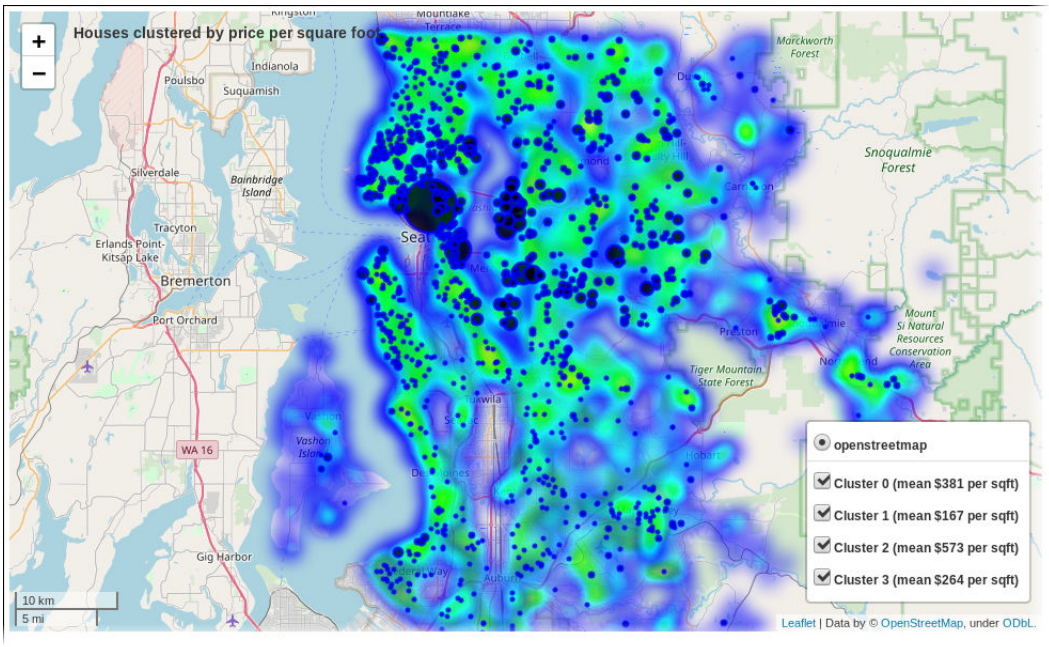
In [ ]:

```
# Cluster homes by the price per square foot.
feature_set_2 = ['price_per_sqft']
X = housing_data[feature_set_2]

# Generate cluster labels for the housing data, assuming 4 clusters.
cluster_labels = get_cluster_labels(4, X)

# Append the cluster labels to a new column in the original dataset.
labeled_houses = housing_data.assign(c_label = cluster_labels)

# Show on the map.
show_on_map(labeled_houses, 'Houses clustered by price per square foot')
```

This is essentially the same code as before, except it's using `price_per_sqft` rather than `lat` and `long` (latitude and longitude).

b) Select the cell that contains the code listing, then select **Run**.

c) Examine the output.



10. How did the clusters form with regard to price per square foot?

11. Prepare to cluster by multiple features of interest to customers.

a) Scroll down and view the cell titled **Prepare to cluster by multiple features of interest to customers**, and examine the code listing below it.

Prepare to cluster by multiple features of interest to customers

```
In [ ]:
# Specify the new dataset.
final_feature_set = ['sqft_living', 'bathrooms', 'bedrooms', 'grade', 'view', 'waterfront']
X = housing_data[final_feature_set]

print('A dataset containing features of interest to customers has been defined.')
```

Now that you're confident the clustering model is working, you'll generate the model using all of the features that are relevant to the prospective buyers' interests. These features are:

- The square footage of the house.
- The number of bathrooms.
- The number of bedrooms.
- The "grade" of the house.
- The quality of the house's view.
- Whether the house has a view to a waterfront.

b) Select the cell that contains the code listing, then select **Run**.

```
A dataset containing features of interest to customers has been defined.
```

12. Use the elbow method to determine the optimal number of clusters.

a) Scroll down and view the cell titled **Use the elbow method to determine the optimal number of clusters**, and examine the code listing below it.

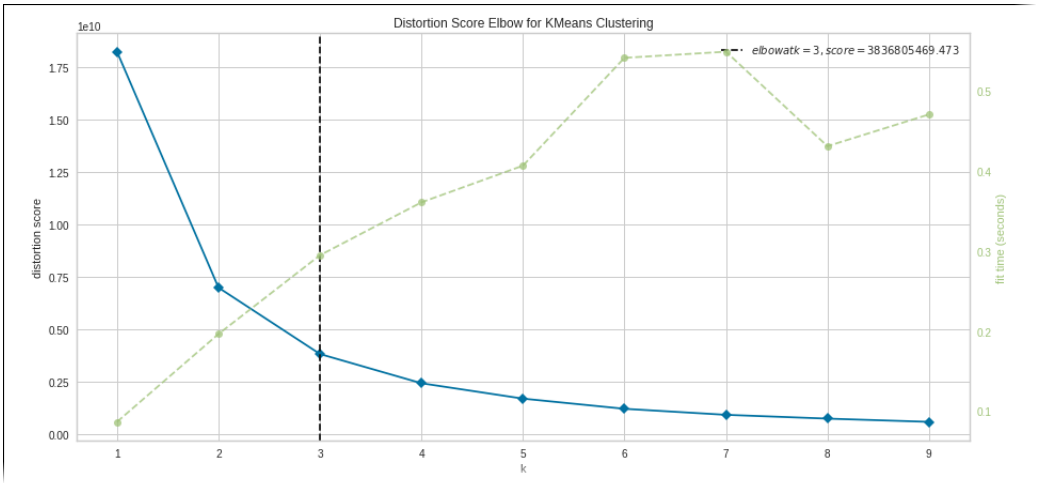
Use the elbow method to determine the optimal number of clusters

In [ ]:
```
from yellowbrick.cluster import KElbowVisualizer

# Use the elbow method to find the optimal number of clusters.
plt.rcParams["figure.figsize"] = (15, 7)

visualizer = KElbowVisualizer(KMeans(init = 'k-means++', random_state = 42), k = (1, 10))
visualizer.fit(X)
visualizer.poof();
```

Rather than supply an arbitrary value for the number of clusters, you should do some analysis to determine what the optimal number of clusters is. The elbow method, generated here by a visualization library called Yellowbrick, is one method of doing so.

b) Select the cell that contains the code listing, then select **Run**.

c) Examine the output.



This plot automatically locates the elbow for you: 3. As you can see, this is the point where adding more clusters gives diminishing performance returns. The dashed green line indicates the time taken to train the model at each cluster value.

13. Use silhouette analysis to determine the optimal number of clusters.

a) Scroll down and view the cell titled **Use silhouette analysis to determine the optimal number of clusters**, and examine lines 1 through 33 in the code listing below it.

```
1  from sklearn.metrics import silhouette_score
2  from sklearn.metrics import silhouette_samples
3
4  # The number of clusters to try out.
5  range_n_clusters = [2, 3, 4, 5]
6
7  high_score = 0
8  optimum_n_clusters = 0
9
10 for n in range_n_clusters:
11
12     # Create k-means model and generate labels from the dataset.
13     kmeans = KMeans(n_clusters = n, random_state = 10)
14     cluster_labels = kmeans.fit_predict(X)
15     silhouette_avg = silhouette_score(X, cluster_labels)
16
17     print('\nWith {} clusters:'.format(n))
18     print('   - Average silhouette score:', silhouette_avg)
19
20     # Note the high score.
21     if silhouette_avg > high_score:
22         high_score = silhouette_avg
23         optimum_n_clusters = n
24
25     # Compute the silhouette scores for each sample.
26     sample_silhouette_values = silhouette_samples(X, cluster_labels)
27
28     # Prepare to plot charts side by side.
29     fig, (ax1, ax2) = plt.subplots(1, 2)
30     fig.set_size_inches(17, 5)
31     ax1.set_xlim([-0.1, 1])
32     ax1.set_ylim([0, len(X) + (n + 1) * 10])
33     y_lower = 10
```

- Line 5 passes in the arbitrary cluster values to try.
- Line 10 begins a `for` loop that iterates through each cluster value.
- Lines 13 through 15 create the k-means clustering model and calculate the highest mean silhouette coefficient of all samples.
- Lines 21 through 23 keep track of the highest mean coefficient of all samples.
- Line 26 computes the silhouette coefficient for each sample.
- Lines 29 through 33 begin plotting two charts side by side: the silhouette plot on the left, and a scatter plot of the clusters on the right.

b) Examine lines 35 through 62.

```
35     # LEFT SIDE: SILHOUETTE PLOTS
36     for i in range(n):
37
38         # Plot silhouette for one cluster at a time.
39         ith_cluster_silhouette_values = sample_silhouette_values[cluster_labels == i]
40         ith_cluster_silhouette_values.sort()
41         size_cluster_i = ith_cluster_silhouette_values.shape[0]
42         y_upper = y_lower + size_cluster_i
43         color = cm.nipy_spectral(float(i) / n)
44         ax1.fill_betweenx(np.arange(y_lower, y_upper),
45                           0,
46                           ith_cluster_silhouette_values,
47                           facecolor = color,
48                           edgecolor = color,
49                           alpha = 1)
50
51         ax1.text(-0.05, y_lower + 0.5 * size_cluster_i, str(i))
52         y_lower = y_upper + 10  # Use 10 for the 0 samples.
53
54         # Log how many values were in this cluster.
55         print('   - Cluster {} includes {} values.'.format(i, size_cluster_i))
56
57     ax1.set_title('Silhouette Plots')
58     ax1.set_xlabel('Silhouette Coefficient Values')
59     ax1.set_ylabel('Cluster Label')
60     ax1.axvline(x = silhouette_avg, color = 'red', linestyle = '--')
61     ax1.set_yticks([])
62     ax1.set_xticks([-0.1, 0, 0.2, 0.4, 0.6, 0.8, 1])
```

- Line 36 begins a `for` loop that iterates through each cluster value to plot the silhouette graphs on the left side of the screen.
- Lines 39 through 62 plot the silhouette for each cluster.

c) Examine lines 64 through 92.

```
63
64      # RIGHT SIDE: SCATTER PLOTS
65
66      # Caption the various clusters.
67      colors = cm.nipy_spectral(cluster_labels.astype(float) / n)
68      centers = kmeans.cluster_centers_
69
70      # Plot the first two features.
71      ax2.scatter(X['sqft_living'],
72                  X['bathrooms'],
73                  marker = 'o',
74                  alpha = 0.7,
75                  s = 50,
76                  color = colors,
77                  edgecolor = 'black');
78
79      # Show a box at the center of each cluster, with the cluster number inside it.
80      ax2.scatter(centers[:, 0],centers[:, 1],marker = 's', c = 'white', alpha = 1.0, s = 200, edgecolor = 'black')
81      for i, c in enumerate(centers):
82          ax2.scatter(c[0], c[1], marker = '$%d$' % i, alpha = 1.0, s = 50, edgecolor = 'black')
83
84      # Axis labels.
85      ax2.set_title('Clustered Data')
86      ax2.set_xlabel(X.columns[0])
87      ax2.set_ylabel(X.columns[1])
88      plt.suptitle(('Number of clusters = %d' % n), fontsize = 16, fontweight = 'bold')
89
90  plt.show()
91
92  print('The highest score ({}) was obtained using {} centers.'.format(high_score, optimum_n_clusters))
```

- Lines 67 through 88 continue the `for` loop, this time plotting the cluster graph that will appear on the right side of the screen for each corresponding silhouette graph.
- Lines 71 and 72 will use the square foot living space and number of bathrooms as the axes for the scatter plots.
- Line 92 prints the highest silhouette coefficient and the number of clusters that obtained it. This can be used to determine the optimal number of clusters.

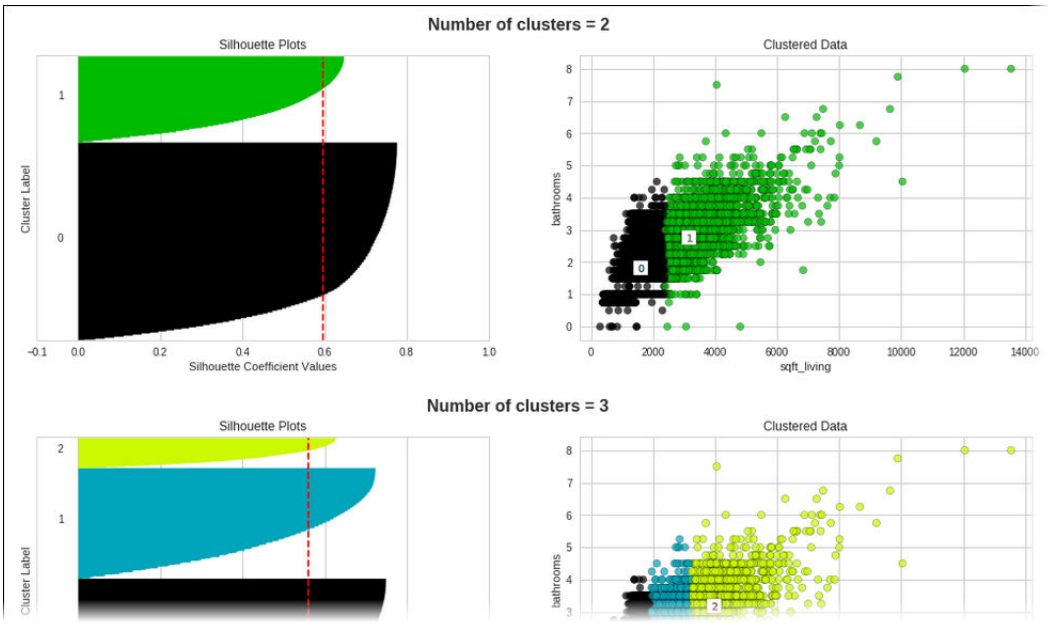
d) Select the cell that contains the code listing, then select **Run**.

e) Examine the output.

```
With 2 clusters:
   - Average silhouette score: 0.5956028486082886
   - Cluster 0 includes 15029 values.
   - Cluster 1 includes 6584 values.

With 3 clusters:
   - Average silhouette score: 0.55921974259051
   - Cluster 0 includes 10918 values.
   - Cluster 1 includes 8374 values.
   - Cluster 2 includes 2321 values.

With 4 clusters:
   - Average silhouette score: 0.5414443565247314
   - Cluster 0 includes 7882 values.
   - Cluster 1 includes 4552 values.
   - Cluster 2 includes 1067 values.
   - Cluster 3 includes 8112 values.

With 5 clusters:
   - Average silhouette score: 0.5321936319368658
   - Cluster 0 includes 7281 values.
   - Cluster 1 includes 2447 values.
   - Cluster 2 includes 388 values.
   - Cluster 3 includes 6331 values.
   - Cluster 4 includes 5166 values.
```

The highest score appears to be for the model with 2 clusters.

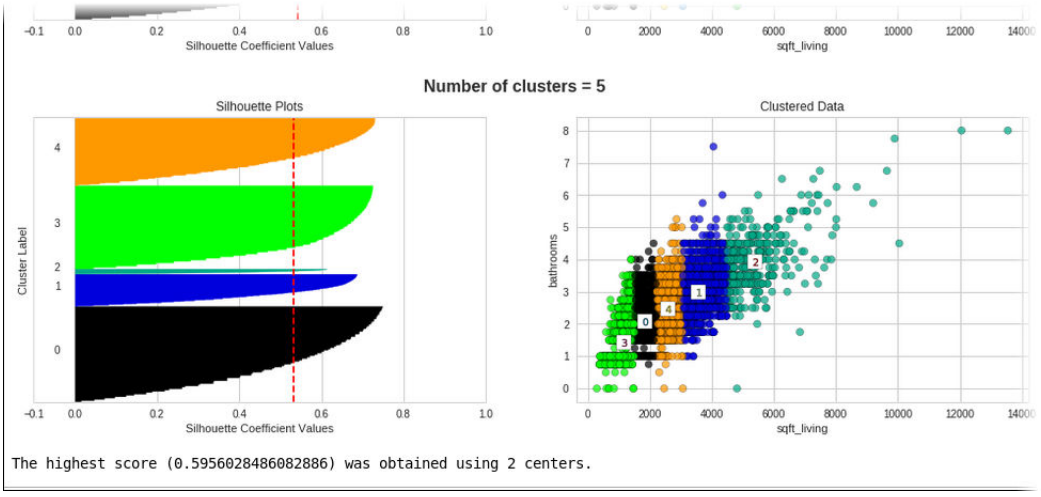
f) Examine the silhouette plots.



- The silhouette coefficient for each cluster value appears as a dashed red line on the silhouette plot.
- The scatter plot represents the corresponding clusters as they appear for square foot living space compared to number of bathrooms.

	Note: Although it might look like cluster 1 in the first scatter plot has more examples than cluster 0, cluster 0 actually includes more, but they're more densely packed. You can scroll up to confirm the number of examples in each cluster.

g) Scroll as needed to see the remaining silhouette plots.



The silhouette method indicates that 2 clusters are optimal, because that amount of clusters returned the highest silhouette coefficient. This doesn't quite align with the elbow method's conclusions, but neither method is necessarily more valid than the other for all problems. For this particular scenario, you'll be using the conclusion of the silhouette analysis as your optimal number of clusters.

14. Use the optimal number of clusters to show the house groups.

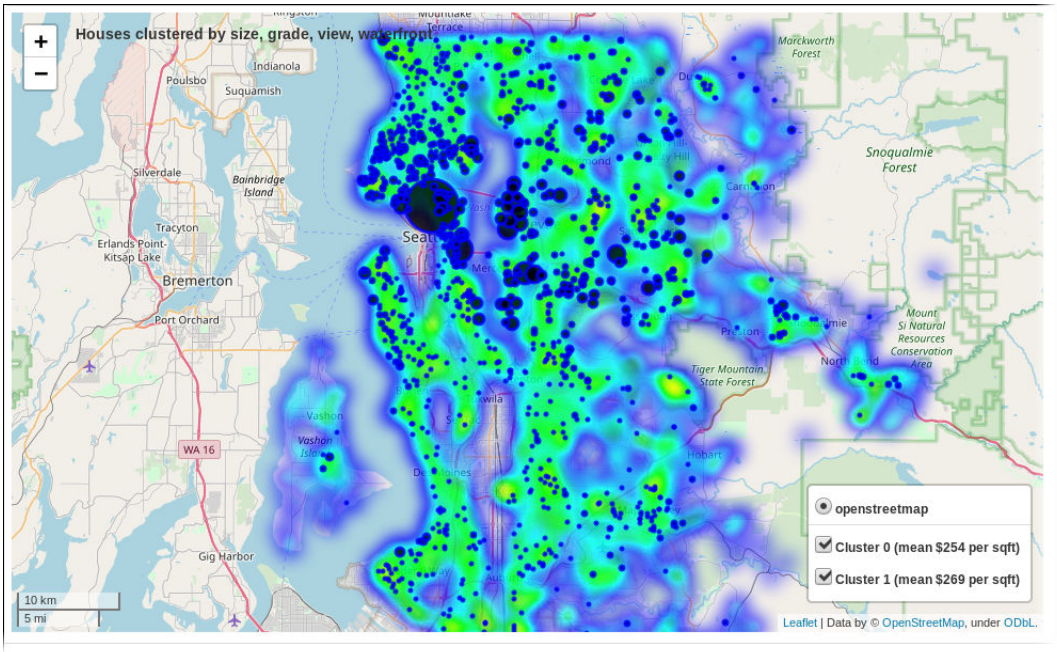
a) Scroll down and select the code cell beneath the **Use the optimal number of clusters to show the house groups** title, then select **Run**.

Use the optimal number of clusters to show the house groups

In [ ]:
```
# Generate cluster labels for the housing data based on the optimal number of clusters.
cluster_labels = get_cluster_labels(optimum_n_clusters, X)

# Append the cluster labels to a new column in the original dataset.
labeled_houses = housing_data.assign(c_label = cluster_labels)

# Show on the map
show_on_map(labeled_houses, 'Houses clustered by size, grade, view, waterfront')
```

Note: Don't get hung up on the fact that the heatmap shows a lot of overlap between the clusters. This is just because the data is being plotted in two dimensions (latitude and longitude), whereas the clusters have incorporated all of the relevant dimensions as specified by `final_feature_set`.

b) Examine the output.



Note: Don't get hung up on the fact that the heatmap shows a lot of overlap between the clusters. This is just because the data is being plotted in two dimensions (latitude and longitude), whereas the clusters have incorporated all of the relevant dimensions as specified by `final_feature_set`.

15. **Do you think this model is adequate in solving the problem of recommending similar houses to buyers who have expressed interest in a specific house? Why or why not?**

16. **What are some reasons why this model may need to be retrained over time?**

17. Shut down the notebook and close the lab.
    a) From the menu, select **Kernel→Shutdown**.
    b) In the **Shutdown kernel?** dialog box, select **Shutdown**.
    c) Close the lab browser tab and continue on with the course.

LAB 3-9
Building a Hierarchical Clustering Model

Data File

~/Clustering/Clustering-Moons.ipynb

Scenario

As part of your efforts to reveal logical groups of data through clustering, you've been presented with another unsupervised dataset. This one, however, includes data that is well separated. The typical *k*-means clustering algorithm may not be the ideal choice in this situation. So, you'll train a *k*-means model on the data and then compare it to the results of a hierarchical agglomerative clustering (HAC) model. The HAC model's different approach to clustering may be more applicable to this dataset. In addition, you'll evaluate and then try to select the optimal number of clusters for that model.

Note: For demonstration purposes, this lab uses an artificially generated dataset that conforms to an usual shape, rather than a real-world dataset. The differences between hierarchical and *k*-means clustering are more obvious with an artificial dataset than with a real-world dataset.

---

1. Open the lab and notebook.
   a) From the Coursera lab page, select **Open Lab** to open a Jupyter Notebook session in another web browser tab.
   b) Select **Clustering/Clustering-Moons.ipynb** to open it.

2. Open the notebook and load the dataset.
   a) Select the code cell beneath the **Import software libraries** title, then select **Run**.
   b) Select the code cell beneath the **Load the dataset** title, then select **Run**.
   c) Examine the output.

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000 entries, 0 to 999
Data columns (total 2 columns):
Col1    1000 non-null float64
Col2    1000 non-null float64
dtypes: float64(2)
memory usage: 15.7 KB
None
```

	Col1	Col2
0	0.201460	0.958164
1	0.050975	0.305029
2	1.573935	-0.366780
3	-0.991440	0.094687
4	0.465245	-0.326029

This dataset is being artificially generated on the fly in order to produce a shape that is conducive to hierarchical clustering. Based on the arguments provided:

- There are 1,000 total samples.
- There are two feature columns, both having float values.
- Some random noise is added to the sample spread for a bit more realism.

3. Plot the data to identify its shape.

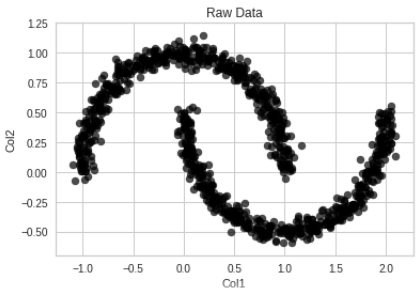
a) Scroll down and select the code cell beneath the **Plot the data to identify its shape** title, then select **Run**.

b) Examine the output.

Plot the data to identify its shape

```
fig, ax = plt.subplots()

# Plot the features
ax.scatter(X['Col1'],
           X['Col2'],
           marker = 'o',
           alpha = 0.7,
           s = 50,
           color = 'black',
           edgecolor = 'black');

# Axis labels.
ax.set_title('Raw Data')
ax.set_xlabel(X.columns[0])
ax.set_ylabel(X.columns[1])
```

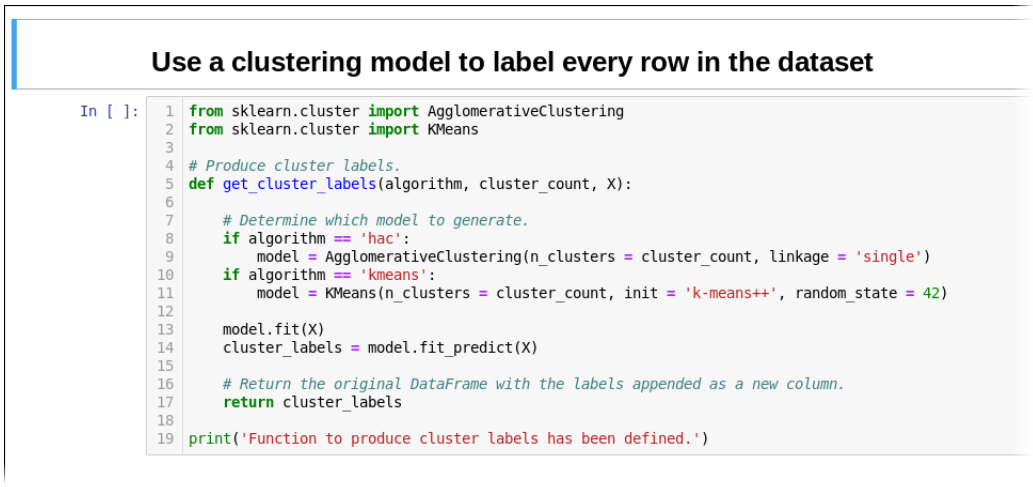
```
Text(0, 0.5, 'Col2')
```



The data takes the shape of two symmetrical crescent moons.

4. Use a clustering model to label every row in the dataset.

a) Scroll down and view the cell titled **Use a clustering model to label every row in the dataset**, and examine the code listing below it.



Use a clustering model to label every row in the dataset

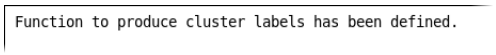
In [ ]:

```
from sklearn.cluster import AgglomerativeClustering
from sklearn.cluster import KMeans

# Produce cluster labels.
def get_cluster_labels(algorithm, cluster_count, X):

    # Determine which model to generate.
    if algorithm == 'hac':
        model = AgglomerativeClustering(n_clusters = cluster_count, linkage = 'single')
    if algorithm == 'kmeans':
        model = KMeans(n_clusters = cluster_count, init = 'k-means++', random_state = 42)

    model.fit(X)
    cluster_labels = model.fit_predict(X)

    # Return the original DataFrame with the labels appended as a new column.
    return cluster_labels

print('Function to produce cluster labels has been defined.')
```

This function is similar to the function used to generate the cluster labels for the King County dataset. The only difference is that you can now select which type of clustering model to generate—*k*-means or agglomerative. For the agglomerative model:

- `n_clusters` is the same as it is in `KMeans()`—it defines the number of clusters.
- `linkage` refers to the linkage criterion, which specifies the distance metric to use when comparing two sets of data examples. In this case, the model is using the `single` method, which uses the minimum of the distances between all data examples in the two sets. This enables the model to determine which clusters to merge.

b) Select the cell that contains the code listing, then select **Run**.



```
Function to produce cluster labels has been defined.
```

5. Plot the clusters.

a) Scroll down and view the cell titled **Plot the clusters**, and examine the code listing below it.

Plot the clusters

In [ ]:

```
1  def plot_clusters():
2
3      fig, ax = plt.subplots()
4
5      # Caption the various clusters.
6      colors = cm.nipy_spectral(cluster_labels.astype(float) / 2)
7
8      # Plot the features.
9      ax.scatter(X['Col1'],
10                X['Col2'],
11                marker = 'o',
12                alpha = 0.7,
13                s = 50,
14                color = colors,
15                edgecolor = 'black');
16
17     # Axis labels.
18     ax.set_title('Clustered Data')
19     ax.set_xlabel(X.columns[0])
20     ax.set_ylabel(X.columns[1])
21
22 print('Function to plot the clusters has been defined.')
```

This function, when called, plots the data as before. The difference is on line 6, in which each of the model's cluster labels is being assigned to a different color. Each data point will be plotted in the color of whatever its cluster label is.

b) Select the cell that contains the code listing, then select **Run**.

```
Function to plot the clusters has been defined.
```

6. Generate a k-means clustering model and plot the results.

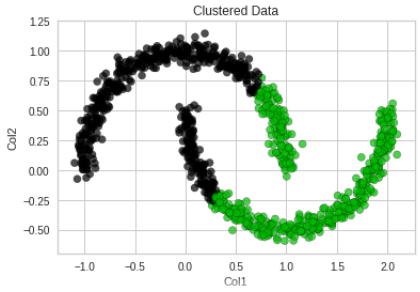
a) Scroll down and select the code cell beneath **Generate a k-means clustering model and plot the results** title, then select **Run**.

b) Examine the output.

Generate a *k*-means clustering model and plot the results

```
# Generate cluster labels for the moons data, assuming 2 clusters.
cluster_labels = get_cluster_labels('kmeans', 2, X)

# Append the cluster labels to a new column in the original dataset.
labeled_data = X.assign(c_label = cluster_labels)

# Show a preview of rows in the dataset with cluster labels added.
display(labeled_data.head())

plot_clusters()
```

	Col1	Col2	c_label
0	0.201460	0.958164	0
1	0.050975	0.305029	0
2	1.573935	-0.366780	1
3	-0.991440	0.094687	0
4	0.465245	-0.326029	1



- Two clusters were generated using a k-means clustering model.
- The clusters don't seem to do a good job of supporting the separation between the crescents. Instead, the clusters seem to be defined by dividing the entire feature space in half.

7. Generate a hierarchical agglomerative clustering model and plot the results.

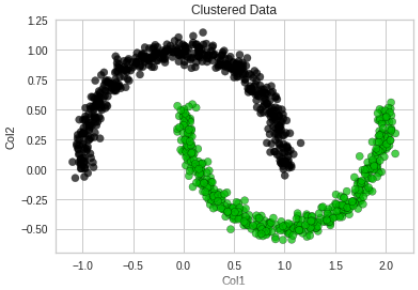
a) Scroll down and select the code cell beneath the **Generate a hierarchical agglomerative clustering model and plot the results** title, then select **Run**.

b) Examine the output.

Generate a hierarchical agglomerative clustering model and plot the results

```
# Generate cluster labels for the moons data, assuming 2 clusters.
cluster_labels = get_cluster_labels('hac', 2, X)

# Append the cluster labels to a new column in the original dataset.
labeled_data = X.assign(c_label = cluster_labels)

# Show a preview of rows in the dataset with cluster labels added.
display(labeled_data.head())

plot_clusters()
```

	Col1	Col2	c_label
0	0.201460	0.958164	0
1	0.050975	0.305029	1
2	1.573935	-0.366780	1
3	-0.991440	0.094687	0
4	0.465245	-0.326029	1



- Two clusters were generated using a hierarchical agglomerative clustering (HAC) model.
- The clusters seem to do a better job of supporting the separation between the crescents.

8. Use the elbow method to determine the optimal number of clusters.

a) Scroll down and view the code cell beneath the **Use the elbow method to determine the optimal number of clusters** title.

Use the elbow method to determine the optimal number of clusters

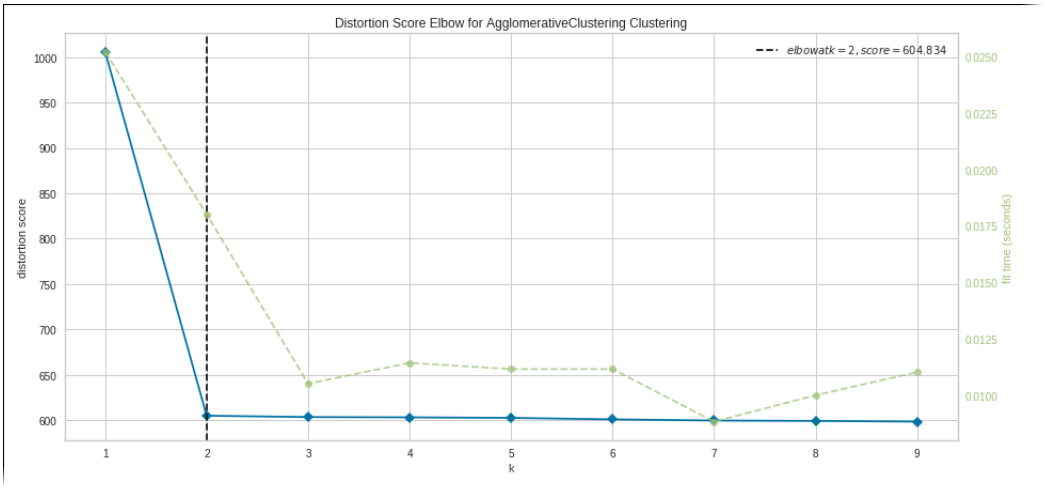
In [ ]:

```
from yellowbrick.cluster import KElbowVisualizer

# Use the elbow method to find the optimal number of clusters.
plt.rcParams["figure.figsize"] = (15, 7)

visualizer = KElbowVisualizer(AgglomerativeClustering(linkage = 'single'), k = (1, 10))
visualizer.fit(X)
visualizer.poof();
```

When using `single` as the linkage mode, silhouette analysis is not particularly useful. The silhouette for each cluster will likely show many negative values. This is because there are many instances where a point in cluster A is actually closer to centroid B, and vice versa. You can see this in the previous cluster plot—the green points at the inner tip of the green crescent are actually closer to the center of the black cluster than they are to their own (green) center.

b) Select the cell that contains the code listing, then select **Run**.

c) Examine the output.



The elbow method clearly indicates that the optimal number of clusters is 2. However, as with k-means clustering, there are other methods to determine k that are just as valid.

9. Plot a dendrogram for additional cluster analysis.

a) Scroll down and view the cell titled **Plot a dendrogram for additional cluster analysis**, and examine the code listing below it.

Plot a dendrogram for additional cluster analysis

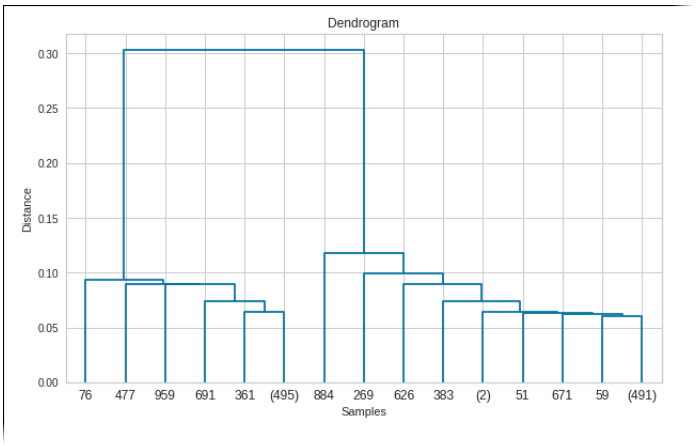
In [ ]:

```
1  import scipy.cluster.hierarchy as sch
2
3  # Specify same linkage used in training model.
4  linkage = sch.linkage(X, method = 'single')
5
6  def plot_dendro(cutoff = 0):
7      plt.figure(figsize = (10, 6))
8      plt.title('Dendrogram')
9      plt.xlabel('Samples')
10     plt.ylabel('Distance')
11
12     # Truncate dendrogram x-axis so it's readable.
13     dendrogram = sch.dendrogram(Z = linkage, truncate_mode = 'lastp', p = 15, color_threshold = cutoff)
14
15 plot_dendro()
```

- Line 4 defines a `linkage` object that will be used in the plot. This is the same `single` linkage criterion used before.
- Line 13 creates the dendrogram with the defined linkage as the `Z` argument.
- The `truncate_mode` will truncate the dendrogram based on the value provided by `p` (15), which determines how many examples are plotted.

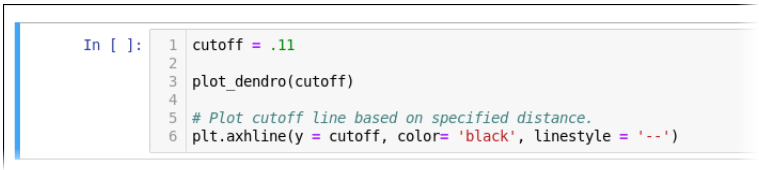
b) Select the cell that contains the code listing, then select **Run**.

c) Examine the output.



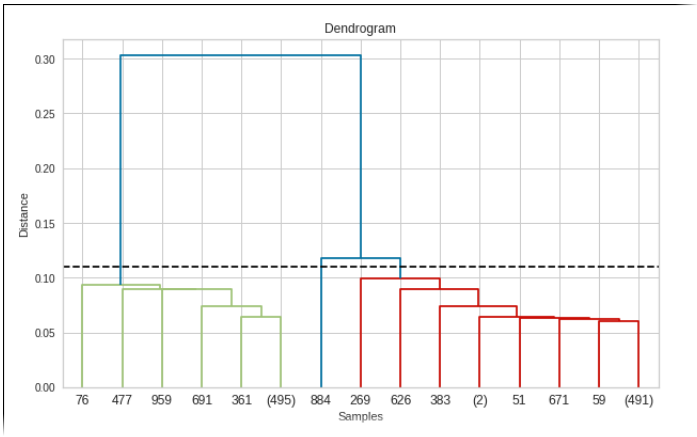
The x-axis plots the 15 examples, whereas the y-axis plots the distance values between examples and clusters.

d) Scroll down and select the next code listing cell.



e) Select **Run**.

f) Examine the output.



A cutoff line is plotted as a way of determining an optimal number of clusters. In this case, the distance of the longest branch before it begins to merge is around a distance of 0.11, so that is being used as the cutoff's y-axis value. Since the cutoff passes through 3 branches, the number of clusters is 3. This aligns with the elbow analysis, but, like with k-means clustering, there is not necessarily one "best" cluster number for a given agglomerative clustering model.

10. **If this were a real-world problem rather than an artificial dataset, how might domain knowledge of that dataset help you determine the optimal number of clusters?**

11. Shut down the notebook and close the lab.
    a) From the menu, select **Kernel→Shutdown**.
    b) In the **Shutdown kernel?** dialog box, select **Shutdown**.
    c) Close the lab browser tab and continue on with the course.

MODULE 6

Project

The following labs are for the Course 3 Project.

PROJECT 3-1

Building a Linear Regression Model to Predict Diabetes Progression

Data File

~/Projects/LinearRegression.ipynb

Scenario

You're doing consultant work for Greene City Physicians Group (GCPG), a medical practice that provides treatment in several different fields. One of those fields is endocrinology. The endocrinologists treat hundreds of different diabetic patients, helping them manage the disease. Preventing the disease from reaching the more severe stages is of primary importance, but knowing when a patient is at risk of progressing to the later stages is not always easy.

The endocrinologists have supplied you with an anonymized dataset of past diabetic patients, including various medical attributes that might be indicators of disease progression. You've been asked to help the doctors predict the progression of the disease in patients. Since this is an ordinal numeric value, you'll create a linear regression model.

The following are the attributes of the dataset:

Attribute	*Description*
`age`	The patient's age in years.
`sex`	The patient's sex.
`bmi`	The patient's body mass index (BMI).
`bp`	The patient's average blood pressure.
`s1–s6`	Six different blood serum measurements taken from the patient.
`target`	A measurement of the disease's progression one year after a baseline.

Note: There is no "correct" answer for every line of code, as there are many ways to write code to do the same thing. You can write the code however you feel most comfortable, as long as it accomplishes the tasks set out for you.

Note: If you're stuck and need help, you can "borrow" code from existing notebooks, and/or look up documentation on how to use the various Python libraries.

Note: Sometimes, when you run code containing logic errors or bugs, you may corrupt the data contained in variables you created in previous code cells. To clear out such problems, you can select **Kernel→Restart & Clear Output**, then run each code cell again.

1. Open the lab and notebook.
   a) From the Coursera lab page, select **Open Lab** to open a Jupyter Notebook session in another web browser tab.
   b) Select **Projects/LinearRegression.ipynb** to open it.

c) Observe the notebook.

Placeholder code cells have been provided in which you can add your own code. Comments provide hints on tasks you might perform in each code cell. The first code cell has already been completely programmed for you.

2. Import software libraries and load the dataset.

a) Select the code listing under **Import software libraries and load the dataset**.

b) Run the code and examine the results.

The dataset is loaded for you. There are 442 records.

3. Get acquainted with the dataset.

a) In the code block under **Get acquainted with the dataset**, write code to convert the loaded dataset to a `DataFrame`, view the data types for each feature, and then view the first 10 records.

Note: Except for the label, the features have already been standardized.

b) Run the code and verify that the shape of the training data is shown.

4. Examine the distribution of various features.

a) In the code block under **Examine the distribution of various features**, write code to plot distribution histograms for all features.

b) Run the code and verify that the distributions are shown.

5. Examine a general summary of statistics.

a) In the code block under **Examine a general summary of statistics**, write code to view summary statistics (mean, standard deviation, min, max, etc.) for each feature.

b) Run the code and verify that the summary statistics are printed.

6. Look for columns that correlate with `target` (disease progression).

a) In the code block under **Look for columns that correlate with `target` (disease progression)**, write code to view the correlation values for each feature compared to the label.

b) Run the code and examine the feature correlations.

7. Split the label from the dataset.

a) In the code block under **Split the label from the dataset**, write code to split the data into train and test sets, as well as split the label from the features.

b) Run the code and observe the shape of the split datasets.

8. Drop columns that won't be used for training.

a) In the code block under **Drop columns that won't be used for training**, write code to drop the three features that have the least correlation with the label.

One of these features is being dropped because it is categorical. You could technically encode that feature instead, but for simplicity's sake, you'll drop it.

b) Run the code and verify that the columns were dropped.

9. Create a linear regression model.

a) In the code block under **Create a linear regression model**, write code to construct a basic linear regression class object, then fit the training data to that object.

b) Run the code.

10. Compare the first ten predictions to actual values.

a) In the code block under **Compare the first ten predictions to actual values**, write code to make predictions on the test set. Then, view the first 10 records with two new columns: the actual label value for that record (disease progression), and the value that your model predicted for that record.
b) Run the code and observe the predictions.

11. Calculate the error between predicted and actual values.

a) In the code block under **Calculate the error between predicted and actual values**, write code to print the mean squared error (MSE) for the model's predictions on the test set.
b) Run the code and observe the error value.

12. Plot lines of best fit for four features.

a) In the code block under **Plot lines of best fit for four features**, write code to generate scatter plots for the four features that exhibited the strongest correlation with the label. Also plot a line of best fit for each feature on top of its scatter plot.
b) Run the code and examine the plots.

13. Save and download the notebook.

a) From the menu, select **File→Save and Checkpoint**.
b) Select **File→Download as→Notebook (.ipynb)**.
c) Save the file to your local drive if it wasn't automatically.
d) Find the saved file and rename it using the following convention: **FirstnameLastname-LinearRegression-Project.ipynb**.

For example: **JohnSmith-LinearRegression-Project.ipynb**.

Note: Make sure not to lose this file. This is the project you'll be uploading and sharing to your peers for grading.

14. Shut down the notebook and close the lab.

a) Back in Jupyter Notebook, select **Kernel→Shutdown**.
b) In the **Shutdown kernel?** dialog box, select **Shutdown**.
c) Close the lab browser tab and continue on to the peer review.

PROJECT 3-2

Creating a Logistic Regression Model to Predict Breast Cancer Recurrence

Data Files

~/Projects/LogisticRegression.ipynb

~/Projects/breast_cancer_data/breast-cancer.csv

Scenario

Breast cancer is a leading cause of death, affecting millions of women across the globe each year. After a first incidence of breast cancer and subsequent remission of a year or more, cancer may reappear in the same location or elsewhere in the body, a situation known as recurrence. Early detection of a recurrence improves a woman's chance of survival, but follow-up care can be time-consuming and expensive. Based on the circumstances, some women are at greater risk for recurrence than others, so it would be very helpful to know which women are at greater risk, and should therefore have much more frequent follow-up care.

Given the extensive data available regarding breast cancer patients, machine learning can provide a valuable tool for helping to classify which patients are at greatest risk for recurrence. As you continue your work with the Greene City Physicians Group (GCPG), you will create a simple logistic regression model to demonstrate how this might be done.

A dataset has been provided to you. The dataset has already been cleaned and encoded, so it shouldn't require any additional preparation to use it for a simple logistic regression model. The dataset has the following attributes:

Attribute	*Description*
`recurrence`	Whether the patient had a recurrence event. • `0` - Patient had no recurrence. • `1` - Patient had a recurrence.
`age_decade`	Age of the patient at the time of diagnosis. Ages have been divided into bins for each decade, so a 33-year-old patient would have a value of 30.

`meno_pre`
`meno_lt_40`
`meno_ge_40`
	These three attributes identify the patient's age at menopause.
	• `meno_pre = 1` - The patient has not yet reached menopause.
	• `meno_lt_40 = 1` - The patient was less than 40 when reaching menopause.
	• `meno_ge40 = 1` - The patient was at least 40 when reaching menopause.

`tumor_size`	The largest diameter (in millimeters) of the excised tumor.
`inv_nodes`	The number (ranging from 0 to 39) of axillary lymph nodes that contain metastatic breast cancer in a histological examination.

Attribute	Description
`node_caps`	The outermost layer of a lymph node is a thin capsule made of connective tissue. Cancer may spread outside the lymph node but remain inside of the capsule. Or it may have penetrated the capsule and further spread into the surrounding tissue. • `node_caps = 0` - The cancer remains contained within the capsule. • `node_caps = 1` - The cancer has spread outside the capsule.
`deg_malig`	The histological grade of the tumor. Tumors that are grade 1 primarily contain cells that retain many of their normal characteristics. Grade 3 tumors contain cells that are largely abnormal, in which disease has spread considerably. • `1` - Still largely normal. • `2` - Somewhat abnormal. • `3` - Largely abnormal.
`breast_left` `breast_right`	These two attributes identify the presence of cancer in each breast. • `breast_left = 1` - The patient had cancer in the left breast. • `breast_right = 1` - The patient had cancer in the right breast.
`irradiat`	Whether the cancer has been irradiated. Cancer cells may be exposed to high-energy x-rays to destroy them, in the hope that the normal cells will regenerate, but cancerous cells will not. • `0` - The cancer has not been irradiated. • `1` - The cancer has been irradiated.

Note: There is no "correct" answer for every line of code, as there are many ways to write code to do the same thing. You can write the code however you feel most comfortable, as long as it accomplishes the tasks set out for you.

Note: If you're stuck and need help, you can "borrow" code from existing notebooks, and/or look up documentation on how to use the various Python libraries.

Note: Sometimes, when you run code containing logic errors or bugs, you may corrupt the data contained in variables you created in previous code cells. To clear out such problems, you can select **Kernel→Restart & Clear Output**, then run each code cell again.

1. Open the lab and examine the data file.

a) From the Coursera lab page, select **Open Lab** to open a Jupyter Notebook session in another web browser tab.

b) Navigate to the **~/Projects/breast_cancer_data/breast-cancer.csv** file directly in Jupyter Notebook. Examine the column headings and refer to the various attributes described in the table above.

c) Close the **breast-cancer.csv** tab to return to the list of files in Jupyter Notebook.

2. Open the logistic regression notebook.

a) From Jupyter Notebook, select **~/Projects/LogisticRegression.ipynb** to open it.

b) Observe what has been provided for you in the notebook.

Placeholder code cells have been provided in which you can add your own code. Comments provide hints on tasks you might perform in each code cell. The first code cell has already been completely programmed for you.

3. Import software libraries and load the dataset.

a) Select the code listing under **Import software libraries and load the dataset**.

b) Run the code and examine the results.

The dataset is loaded for you. There are 286 records.

4. Get acquainted with the data structure and preview the records.

a) In the code block under **Get acquainted with the data structure and preview the records**, write statements to show the various features and their data types and to view the first five records.

b) Run the code cell and verify that a summary of the various features and their data types is shown, along with a sample of data in the first five records.

5. Examine descriptive statistics.

a) In the code block under **Examine descriptive statistics**, write code to show descriptive statistics (count, mean, std, etc.) for the dataset.

b) Run the code cell and verify that descriptive statistics are shown for features in the dataset.

6. Use histograms to visualize the distribution of various features.

a) In the code block under **Use histograms to visualize the distribution of various features**, write code to show a histogram for each attribute in the dataset.

b) Run the code cell and verify that the histograms are shown for each feature.

7. Split the data into training and validation sets and labels

a) In the code block under **Split the data into training and validation sets and labels**, write code to perform the following tasks:

- Import a function to split the dataset
- Specify the column to be included in the label set (`recurrence`)
- Specify columns to be included in the training and validation sets (all other columns)
- Split the training set, validation set, and labels for both
- Compare the number of rows and columns in the original data to the training and validation sets

b) Run the code cell and verify that the data has been split into training, validation, and label sets.

c) In the following two code cells, write and run code to preview a sample of the training data and labels.

8. Build the model

a) In the code block under **Build the model**, write code to create a logistic regression model, and use the validation data and labels to score it.

b) Run the code cell and observe the score.

9. Test the model

a) In the code block under **Test the model**, write code to perform the following tasks:

- Add columns to a copy of the test data to compare predictions against actual values.
- View examples of the predictions compared to actual recurrence.

b) Run the code cell.

c) Scroll if necessary, and compare the predicted values against the actual values.

10. Save and download the notebook.

a) From the menu, select **File→Save and Checkpoint**.

b) Select **File→Download as→Notebook (.ipynb)**.

c) Save the file to your local drive if it wasn't automatically.

d) Find the saved file and rename it using the following convention: **FirstnameLastname-LogisticRegression-Project.ipynb**.

For example: **JohnSmith-LogisticRegression-Project.ipynb**.

Note: Make sure not to lose this file. This is the project you'll be uploading and sharing to your peers for grading.

11. Shut down the notebook and close the lab.

a) Back in Jupyter Notebook, select **Kernel→Shutdown**.
b) In the **Shutdown kernel?** dialog box, select **Shutdown**.
c) Close the lab browser tab and continue on to the peer review.

PROJECT 3-3

Building a Clustering Model for Customer Segmentation

Data Files

~/Projects/Clustering.ipynb

~/Projects/wholesale_customers_data/wholesale_customer_data.csv

Scenario

You work for Mixed Messages Media, a marketing firm. One of the firm's clients is a large wholesale distributor. The distributor sells many different kinds of products to various retail stores, but specializes in selling food products. As part of a marketing push, you've been hired to help the distributor with its customer segmentation approach. The distributor wants to be able to target their advertisements to specific retailers in order to maximize sales. You've been given historical data that includes annual spending figures for each of the distributor's retail clients for several product categories (paper products, frozen products, milk products, etc.). There is no label associated with this data, so your mission will be to try to assign the retailers to meaningful groups based on how much they spend for each type of product. So, you'll use a clustering approach to this data.

The following are the attributes of the dataset:

Attribute	*Annual Spending On*
`Fresh`	Fresh products.
`Milk`	Milk products.
`Grocery`	Grocery products.
`Frozen`	Frozen products.
`Detergents_Paper`	Detergent products and paper products.
`Deli`	Deli products.

Note: There is no "correct" answer for every line of code, as there are many ways to write code to do the same thing. You can write the code however you feel most comfortable, as long as it accomplishes the tasks set out for you.

Note: If you're stuck and need help, you can "borrow" code from existing notebooks, and/or look up documentation on how to use the various Python libraries.

Note: Sometimes, when you run code containing logic errors or bugs, you may corrupt the data contained in variables you created in previous code cells. To clear out such problems, you can select **Kernel→Restart & Clear Output**, then run each code cell again.

1. Open the lab and notebook.
   a) From the Coursera lab page, select **Open Lab** to open a Jupyter Notebook session in another web browser tab.
   b) Select **Projects/Clustering.ipynb** to open it.
   c) Observe the notebook.

Placeholder code cells have been provided in which you can add your own code. Comments provide hints on tasks you might perform in each code cell. The first code cell has already been completely programmed for you.

2. Import software libraries and load the dataset.
   a) Select the code listing under **Import software libraries and load the dataset**.
   b) Run the code and examine the results.
   The dataset is loaded for you. There are 440 records.

3. Get acquainted with the dataset.
   a) In the code block under **Get acquainted with the dataset**, write code to view the data types for each feature, and then view the first 10 records.
   b) Run the code and verify that the shape of the training data is shown.

4. Examine the distribution of various features.
   a) In the code block under **Examine the distribution of various features**, write code to plot distribution histograms for all features.
   b) Run the code and verify that the distributions are shown.

5. Examine a general summary of statistics.
   a) In the code block under **Examine a general summary of statistics**, write code to view summary statistics (mean, standard deviation, min, max, etc.) for each feature.
   b) Run the code and verify that the summary statistics are printed.

6. Use a k-means model to label every row in the dataset.
   a) In the code block under **Use a *k*-means model to label every row in the dataset**, write code to create a k-means clustering object with 3 as the initial number of clusters.
   b) Write code to fit the training data to the clustering object, using only the fresh products and milk products features. Then, predict the cluster labels using this truncated dataset.
   c) Run the code.

7. Attach cluster labels to the original dataset.
   a) In the code block under **Attach cluster labels to the original dataset**, write code to append the cluster labels to the original dataset, then preview the first five rows.
   b) Run the code and observe the label assignments.

8. Show clusters of customers based on fresh products and milk products sale.
   a) In the code block under **Show clusters of customers based on fresh products and milk products sale**, write code to show a scatter plot of customer data, where fresh products is the x-axis and milk products is the y-axis. Ensure each cluster is distinguished by color.
   b) Run the code and observe the plotted clusters.

9. Use the elbow method to determine the optimal number of clusters.
   a) In the code block under **Use the elbow method to determine the optimal number of clusters**, write code to generate an elbow plot for 1 to 10 clusters. Fit the full training data to the model this time.
   b) Run the code and examine the suggested number of clusters based on the elbow point analysis.

10. Use silhouette analysis to determine the optimal number of clusters.
   a) In the code block under **Use silhouette analysis to determine the optimal number of clusters**, write code to print the silhouette scores of several k-means clustering models. The number of clusters for each model should range from 2 to 5.

Note: You can write code to visually plot the silhouettes if you want to, but just returning the scores is sufficient.

b) Write code to print the number of clusters that received the highest silhouette score.
c) Run the code and examine the suggested number of clusters based on the silhouette analysis.

11. Generate and preview cluster labels using the full dataset.

a) In the code block under **Generate and preview cluster labels using the full dataset**, write code to generate a k-means clustering model and then fit the full training dataset to it. Use your own judgment to determine the desired number of clusters.
b) Write code to predict the cluster labels, then append the labels to the dataset.
c) Write code to show the first 20 rows in the dataset.
d) Run the code and examine the cluster assignments.

12. Save and download the notebook.

a) From the menu, select **File→Save and Checkpoint**.
b) Select **File→Download as→Notebook (.ipynb)**.
c) Save the file to your local drive if it wasn't automatically.
d) Find the saved file and rename it using the following convention: **FirstnameLastname-Clustering-Project.ipynb**.

   For example: **JohnSmith-Clustering-Project.ipynb**.

Note: Make sure not to lose this file. This is the project you'll be uploading and sharing to your peers for grading.

13. Shut down the notebook and close the lab.

a) Back in Jupyter Notebook, select **Kernel→Shutdown**.
b) In the **Shutdown kernel?** dialog box, select **Shutdown**.
c) Close the lab browser tab and continue on to the peer review.

Course 4: Build Decision Trees, SVMs, and Artificial Neural Networks

Course Introduction

The following labs are for Course 4: Build Decision Trees, SVMs, and Artificial Neural Networks.

Modules

The labs in this course pertain to the following modules:

- Module 1: Build Decision Trees and Random Forests
- Module 2: Build Support-Vector Machines (SVM)
- Module 3: Build Multi-Layer Perceptrons (MLP)
- Module 4: Build Convolutional and Recurrent Neural Networks (CNN/RNN)
- Apply What You've Learned

MODULE 1

Build Decision Trees and Random Forests

The following labs are for Module 1: Build Decision Trees and Random Forests.

LAB 4-1
Building a Decision Tree Model

Data Files

~/Decision Trees and Random Forests/DecisionTrees-Titanic.ipynb

~/Decision Trees and Random Forests/titanic_data/test.csv

~/Decision Trees and Random Forests/titanic_data/train.csv

Scenario

The logistic regression model you created and optimized for the *Titanic* dataset has done well in solving the problem assigned to it. However, you're concerned that the model may not be the best possible classifier for the job. Logistic regression isn't the only classification algorithm out there, and in order to truly be sure that you've built the best model, you need to apply your data to other algorithms. So, you'll try to build a competing model using a decision tree algorithm, then compare the results to your earlier logistic regression model. There's no guarantee that the decision tree model will be any better, but you won't know until you actually build and evaluate it.

---

1. Open the lab and notebook.
   a) From the Coursera lab page, select **Open Lab** to open a Jupyter Notebook session in another web browser tab.
   b) Select **Decision Trees and Random Forests/DecisionTrees-Titanic.ipynb** to open it.

2. Import the relevant libraries and load the dataset.
   a) View the cell titled **Import software libraries and load the dataset**, and examine the code listing below it.
   b) Select the cell that contains the code listing, then select **Run**.
   c) Verify that **test.csv** and **train.csv** are in the project folder, and that the latter was loaded with 891 records.

3. Split the datasets.
   a) Scroll down and view the cell titled **Split the datasets**, and examine the code listing below it.
   b) Select the cell that contains the code listing, then select **Run**.
      You'll be using cross-validation later during hyperparameter optimization; for now, you'll just split the datasets.

4. Look for categorical features that need to be one-hot encoded.
   a) Scroll down and view the cell titled **Look for categorical features that need to be one-hot encoded**, and examine the code listing below it.
   b) Select the cell that contains the code listing, then select **Run**.
   c) Examine the output.

5. The scikit-learn implementation of decision trees does not work well with non-ordinal categorical values. Such values must be one-hot encoded before being trained. For the following question, keep in mind that `PassengerId`, `Cabin`, `Ticket`, and `Name` will be dropped from the dataset since they are largely irrelevant to the learning process.

   Which of these remaining features do you think need to be one-hot encoded, and why?

6. Perform common preparation on the training and validation sets.
   a) Scroll down and view the cell titled **Perform common preparation on the training and validation sets**, and examine the code listing below it.

```
1  # Perform common cleaning and feature engineering tasks on datasets.
2  def prep_dataset(dataset):
3
4      # PROVIDE MISSING VALUES
5
6      # Fill missing Age values with the median age.
7      dataset['Age'].fillna(dataset['Age'].median(), inplace = True)
8
9      # Fill missing Fare values with the median fare.
10     dataset['Fare'].fillna(dataset['Fare'].median(), inplace = True)
11
12     # Fill missing Embarked values with the mode.
13     dataset['Embarked'].fillna(dataset['Embarked'].mode()[0], inplace = True)
14
15     # ONE-HOT ENCODING
16
17     cols = ['Pclass', 'Sex', 'Embarked']
18
19     for i in cols:
20         dummies = pd.get_dummies(dataset[i], prefix = i, drop_first = False)
21         dataset = pd.concat([dataset, dummies], axis = 1)
22
23     return dataset
24
25 X_train = prep_dataset(X_train.copy())
26
27 X_val = prep_dataset(X_val.copy())
```

   This is somewhat different than the data preparation you performed earlier as part of building a logistic regression model.

   - On lines 4 to 13, the function is still imputing missing values for `Age`, `Fare`, and `Embarked`. The scikit-learn implementation of decision trees does not support missing values, so this is necessary.
   - On lines 15 to 21, the function is performing one-hot encoding on the categorical features that appear to need it. The `get_dummies()` function from the pandas library automatically performs one-hot encoding for the given features.
   - You're not combining `SibSp` and `Parch` into its own feature as before, as a decision tree may be able to generate useful splits from each feature individually.

   b) Select the cell that contains the code listing, then select **Run**.

7. Drop columns that won't be used for training.
   a) Scroll down and view the cell titled **Drop columns that won't be used for training**, and examine the code listing below it.
   b) Select the cell that contains the code listing, then select **Run**.

8. Preview current training data.

a) Scroll down and view the cell titled **Preview current training data**, and examine the code listing below it.

b) Select the cell that contains the code listing, then select **Run**.

c) Examine the output.

	Age	SibSp	Parch	Fare	Pclass_1	Pclass_2	Pclass_3	Sex_female	Sex_male	Embarked_C	Embarked_Q	Embarked_S
439	31.0	0	0	10.5000	0	1	0	0	1	0	0	1
617	26.0	1	0	16.1000	0	0	1	1	0	0	0	1
242	29.0	0	0	10.5000	0	1	0	0	1	0	0	1
82	27.5	0	0	7.7875	0	0	1	1	0	0	1	0
398	23.0	0	0	10.5000	0	1	0	0	1	0	0	1

The unnecessary features have been dropped, and the one-hot encoded features have replaced the originals.

9. Create a basic decision tree model.

a) Scroll down and view the cell titled **Create a basic decision tree model**, and examine the code listing below it.

```
1  from sklearn.tree import DecisionTreeClassifier
2
3  tree = DecisionTreeClassifier(random_state = 1912)
4  start = time()
5  tree.fit(X_train, np.ravel(y_train))
6  end = time()
7  train_time = (end - start) * 1000
8
9  prediction = tree.predict(X_val)
10
11 # Score using the validation data.
12 score = tree.score(X_val, y_val)
13
14 print('Decision tree model took {:.2f} milliseconds to fit.'.format(train_time))
15 print('Accuracy: {:.0f}%'.format(score * 100))
```

The `DecisionTreeClassifier()` class is scikit-learn's implementation of the CART algorithm. Other than the random seed state, the object being created on line 3 uses all of the default values:

- `criterion` defaults to `gini`, which is the Gini index used by CART. The only other option is `entropy`, or information gain. There is no C4.5 implementation (information gain ratio) in the standard scikit-learn library.
- `max_depth` defaults to `None`; the tree will keep splitting until the Gini index is 0, or until all leaves contain less than the number of samples specified by `min_samples_split`.
- `min_samples_split` defaults to 2.
- `min_samples_leaf` defaults to 1.

b) Select the cell that contains the code listing, then select **Run**.

c) Examine the output.

```
Decision tree model took 9.44 milliseconds to fit.
Accuracy: 80%
```

Like with other classification methods, you can evaluate the model using the familiar metrics. The default model has an accuracy of 80%.

10. Visualize the decision tree structure.

a) Scroll down and view the cell titled **Visualize the decision tree structure**, and examine the code listing below it.

```
from sklearn.tree import export_graphviz
from sklearn.externals.six import StringIO
from IPython.display import Image, display
import pydotplus as pdotp

def plot_tree(model, image):
    dot_data = StringIO()
    export_graphviz(model, out_file = dot_data,
                    filled = True,
                    rounded = True,
                    special_characters = True,
                    feature_names = X_train.columns.values.tolist(),
                    class_names = ['0', '1'])

    graph = pdotp.graph_from_dot_data(dot_data.getvalue())
    graph.write_png(image)
    Image(graph.create_png())
```

This function, when called, will use the `export_graphviz` library to visually generate the structure of the decision tree. The `pydotplus` library writes the visual structure to an image file on the computer's file system.

b) Select the cell that contains the code listing, then select **Run**.

11. Compute accuracy, precision, recall, and F_1 score.

a) Scroll down and view the cell titled **Compute accuracy, precision, recall, and F_1 score**, and examine the code listing below it.

b) Select the cell that contains the code listing, then select **Run**.

This function, when called, computes the four named statistical measures.

12. Generate a ROC curve and compute the AUC.

a) Scroll down and view the cell titled **Generate a ROC curve and compute the AUC**, and examine the code listing below it.

b) Select the cell that contains the code listing, then select **Run**.

This function, when called, generates a ROC curve.

13. Generate a precision–recall curve and compute the average precision.

a) Scroll down and view the cell titled **Generate a precision–recall curve and compute the average precision**, and examine the code listing below it.

b) Select the cell that contains the code listing, then select **Run**.

This function, when called, generates a precision–recall curve.

14. Evaluate the initial decision tree model.

a) Scroll down and view the cell titled **Evaluate the initial decision tree model**, and examine the code listing below it.

b) Select the cell that contains the code listing, then select **Run**.

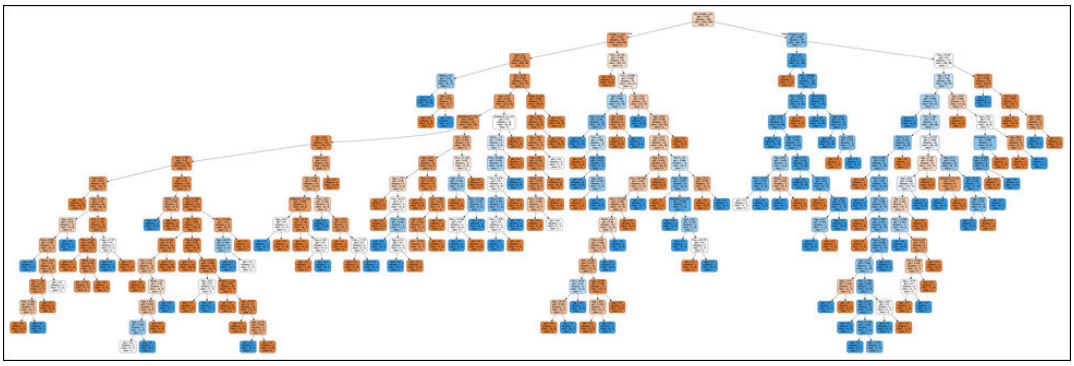
c) Wait for the `graphviz` library to download and install.

Note: The conda installer might indicate that it failed; however, it will keep trying and should finish installing the library within a minute or two.

d) Scroll down and select the next code listing cell.

e) Select **Run**.

f) Examine the output.



By default, the decision tree is quite large and difficult to interpret. You'll address this soon; for now, you can just ignore the specifics.

g) Scroll down and select the next code listing cell.

```
initial_predict = tree.predict(X_val)

model_scores(y_val, initial_predict)
```

h) Select **Run**.

i) Examine the output.

```
Accuracy: 80%
Precision: 74%
Recall: 63%
F1: 68%
```

There's room to improve these scores, which you'll do shortly.

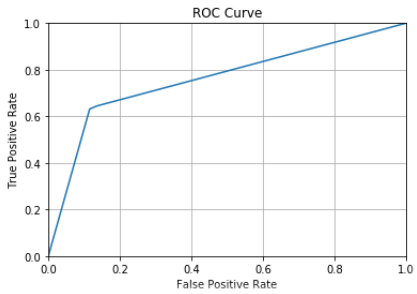
j) Scroll down and select the next code listing cell.

```
initial_predict_proba = tree.predict_proba(X_val)

roc(y_val, initial_predict_proba[:, 1])
```

k) Select **Run**.

l) Examine the output.

```
Area Under Curve: 0.76
```

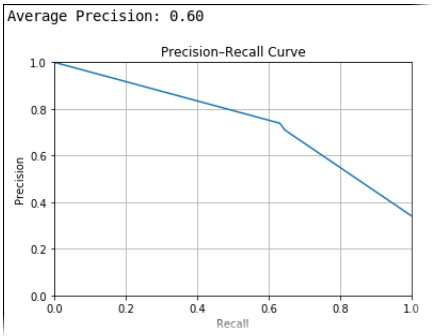


m) Scroll down and select the next code listing cell.

```
prc(y_val, initial_predict_proba[:, 1])
```

n) Select **Run**.

o) Examine the output.



Both the AUC and the average precision are relatively low at this point.

15. Perform pre-pruning on the decision tree.

a) Scroll down and view the cell titled **Perform pre-pruning on the decision tree**, and examine the code listing below it.

```
from sklearn.tree import DecisionTreeClassifier

pruned_tree = DecisionTreeClassifier(max_depth = 4, random_state = 1912)
start = time()
pruned_tree.fit(X_train, np.ravel(y_train))
end = time()
train_time = (end - start) * 1000

prediction = pruned_tree.predict(X_val)

# Score using the validation data.
score = pruned_tree.score(X_val, y_val)

print('Decision tree model took {:.2f} milliseconds to fit.'.format(train_time))
print('Accuracy: {:.0f}%'.format(score * 100))
```

The only difference here is that the decision tree is being constrained to a max depth of 4. This is a pre-pruning technique that can help improve the skill of the model, as too large of trees can lead to overfitting.

b) Select the cell that contains the code listing, then select **Run**.

c) Examine the output.

```
Decision tree model took 5.74 milliseconds to fit.
Accuracy: 82%
```

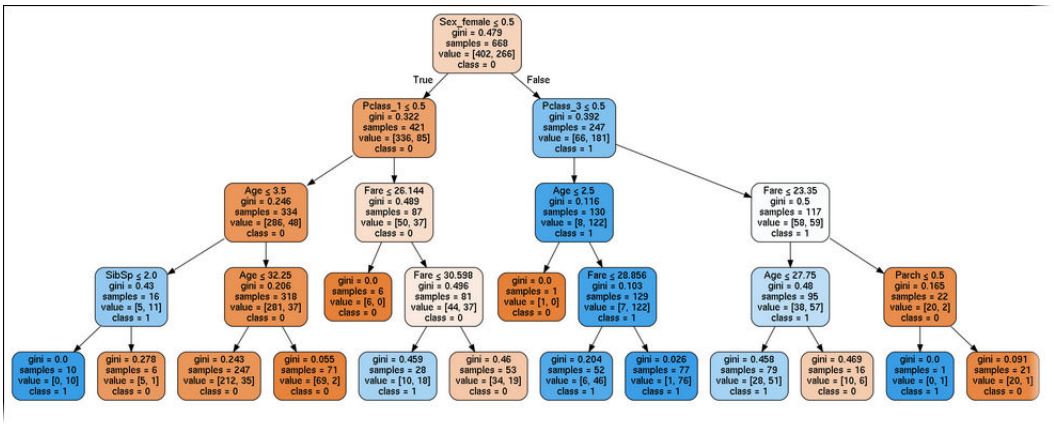
The accuracy has increased slightly.

16. Evaluate the pruned decision tree model.

a) Scroll down and view the cell titled **Evaluate the pruned decision tree model**, and examine the code listing below it.

b) Select the cell that contains the code listing, then select **Run**.

c) Examine the output.



This pruned tree is much more manageable and easier to interpret.

> **Note:** You can double-click the image to zoom in.

Each node includes splitting/decision information at that node. For the root decision node at the top:

- The tree starts by evaluating if the passenger was *not* female (in other words, male).
- The Gini index at this point is 0.479. An index of 0 represents purity, so the number should be decreasing as you descend the branches.
- The number of samples at this node is 668 (the entire training set).
- The `value` shows that 402 examples are true for this decision, and 266 are false. In other words, 402 passengers were male, and 266 passengers were female.
- The class prediction at this point is 0, meaning perished. This prediction will get much more granular as you descend the tree.

For the first "True" branch on the left:

- The decision node here is evaluating if the passenger was *not* in first class.
- The Gini index is 0.322, which, as expected, is becoming more pure.
- The number of samples is 421.
- 336 samples were not in first class; 85 were.
- The class prediction here is still 0.

For the first "False" branch on the right:

- The decision node here is evaluating if the passenger was *not* in third class.
- The Gini index is 0.392.
- The number of samples is 247.
- 66 samples were not in third class; 181 were.
- The class prediction is 1 (survived).

You can continue to descend the tree until you get to the bottom, where there are multiple leaves, each with their own class predictions based on the branching logic of their parent nodes. Note that the Gini indices for these leaves are low, but few of them are totally pure (0). This is because you specified a maximum depth as an early stopping criterion.

d) Scroll down and select the next code listing cell.

```
pruned_predict = pruned_tree.predict(X_val)

model_scores(y_val, pruned_predict)
```

e) Select **Run**.

f) Examine the output.

```
Accuracy: 82%
Precision: 82%
Recall: 59%
F1: 69%
```

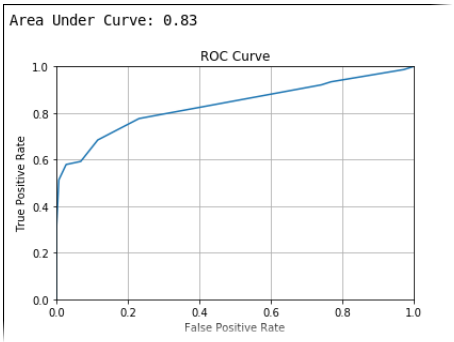
The recall seems to have decreased by a few percent, whereas the other metrics have increased by a few.

g) Scroll down and select the next code listing cell.

```
pruned_predict_proba = pruned_tree.predict_proba(X_val)

roc(y_val, pruned_predict_proba[:, 1])
```

h) Select **Run**.

i) Examine the output.

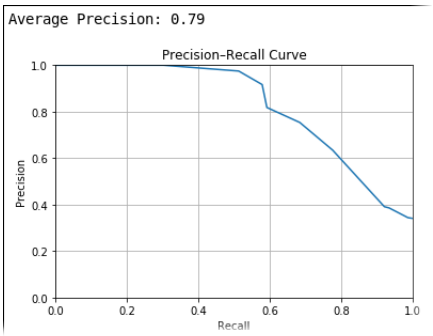


j) Scroll down and select the next code listing cell.

```
prc(y_val, pruned_predict_proba[:, 1])
```

k) Select **Run**.

l) Examine the output.



Both the AUC and the average precision have improved; the latter has experienced a more significant improvement.

17. Fit a decision tree model using randomized search with cross-validation.

a) Scroll down and view the cell titled **Fit a decision tree model using randomized search with cross-validation**, and examine the code listing below it.

```
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import randint as sp_randint

dist = {'criterion': ['gini', 'entropy'],
        'splitter': ['best', 'random'],
        'max_depth': sp_randint(2, 10),
        'min_samples_split': sp_randint(5, 100),
        'min_samples_leaf': sp_randint(5, 100)}

search = RandomizedSearchCV(tree, param_distributions = dist, n_iter = 500,
                            scoring = 'f1', cv = 5, iid = False, random_state = 1912)
search.fit(X_train, np.ravel(y_train));
optimized_tree = search.best_estimator_

print(search.best_params_)
```

The `dist` dictionary holds the hyperparameters and the hyperparameter distributions that the randomized search will use to train and evaluate the model. Because there are so many potential combinations, grid search will take too long, so randomized search is preferable. The dictionary is as follows:

- The `criterion` is either the Gini index or information gain.
- The `splitter` is either `"best"` (the default Gini index approach) or `"random"`.
- The `max_depth` is a distribution of random integers from 2 to 10.
- The `min_samples_split` is a distribution of random integers from 5 to 100.
- The `min_samples_leaf` is a distribution of random integers from 5 to 100.

When `RandomizedSearchCV()` is called:

- The hyperparameter dictionary is passed in with the `param_distributions` argument.
- The number of times a random combination of hyperparameters will be sampled (`n_iter`) is 500.
- The `scoring` argument is what the randomized search is optimizing for. In this case, you're optimizing for F_1 score. Like with grid search, you could optimize for whichever metric you'd prefer.
- By providing a `cv` value of 5, the dataset will be split using stratified k-fold cross-validation, where k is 5.

b) Select the cell that contains the code listing, then select **Run**.

This may take a few moments.

c) Examine the output.

```
{'criterion': 'gini', 'max depth': 6, 'min samples leaf': 10, 'min samples split': 78, 'splitter': 'best'}
```

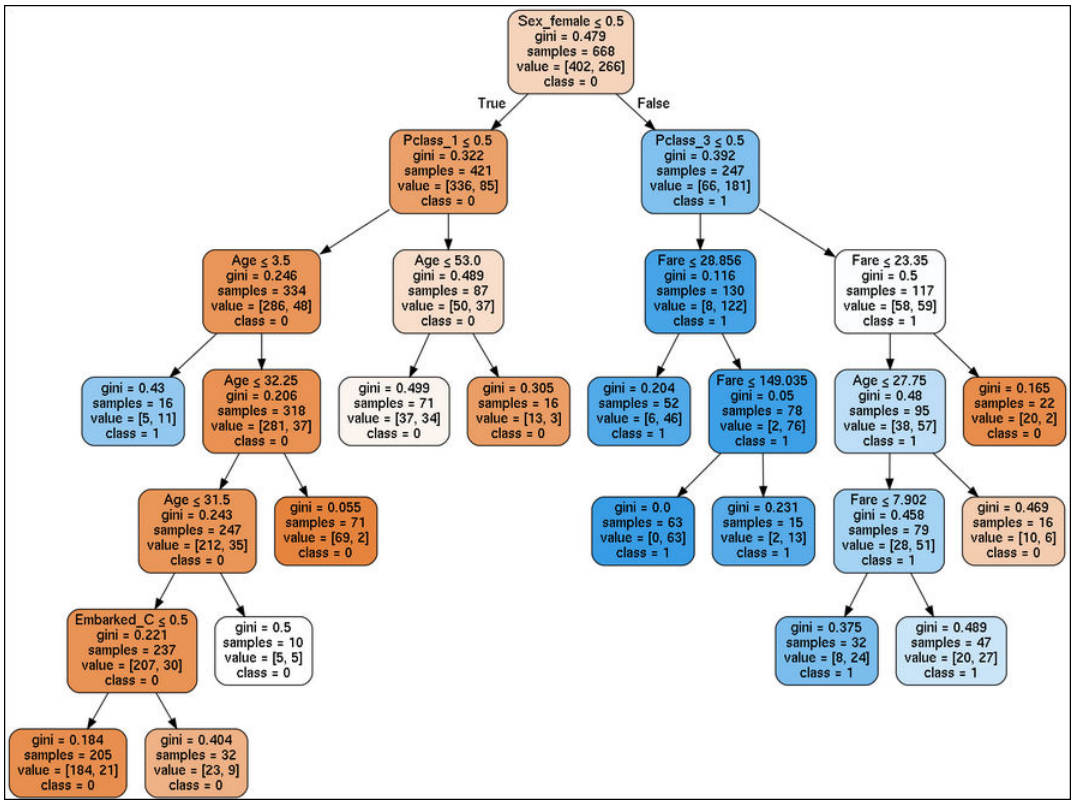
Randomized search identified these hyperparameters as being the best combination for optimizing the F_1 score on the model.

18. Evaluate the optimized model.

a) Scroll down and view the cell titled **Evaluate the optimized model**, and examine the code listing below it.

b) Select the cell that contains the code listing, then select **Run**.

c) Examine the output.



The decision tree has changed based on the selected hyperparameters. The max depth is larger, as are both the minimum number of splits and the minimum number of samples at a leaf.

d) Scroll down and select the next code listing cell.

```
optimized_predict = optimized_tree.predict(X_val)

model_scores(y_val, optimized_predict)
```

e) Select **Run**.

f) Examine the output.

```
Accuracy: 84%
Precision: 92%
Recall: 59%
F1: 72%
```

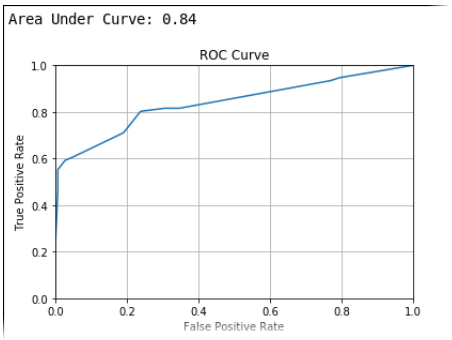
Accuracy and F_1 have improved slightly, precision has improved by a lot, and recall hasn't changed.

g) Scroll down and select the next code listing cell.

```
optimized_predict_proba = optimized_tree.predict_proba(X_val)

roc(y_val, optimized_predict_proba[:, 1])
```

h) Select **Run**.

i) Examine the output.

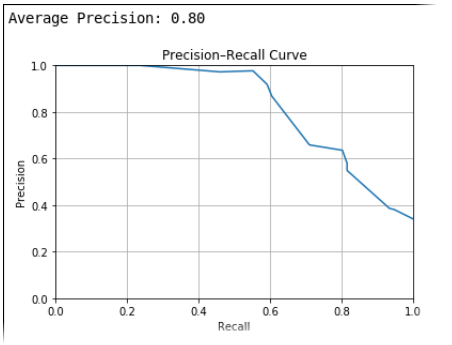


j) Scroll down and select the next code listing cell.

```
1 prc(y_val, optimized_predict_proba[:, 1])
```

k) Select **Run**.

l) Examine the output.



Both the AUC and the average precision have experienced a very slight improvement.

19. Compare the results to a model that used a different algorithm.

a) Compare the optimized decision tree model with the results of the optimized logistic regression model you created earlier.

For reference, here are the results:

Model	*Accuracy*	*Precision*	*Recall*	F_1	*ROC AUC*	*Average Precision*
Optimized logistic regression	83%	76%	71%	73%	85%	84%
Optimized decision tree	84%	92%	59%	72%	84%	80%

b) Use the results table to answer the following question.

20. **Are you satisfied with the results of the decision tree as compared to the logistic regression model? Do you think one model is more skillful than the other? If so, which one, and why?**

21. Shut down the notebook and close the lab.
    a) From the menu, select **Kernel→Shutdown**.
    b) In the **Shutdown kernel?** dialog box, select **Shutdown**.
    c) Close the lab browser tab and continue on with the course.

LAB 4-2
Building a Random Forest Model

Data Files

~/Decision Trees and Random Forests/RandomForests-Titanic.ipynb

~/Decision Trees and Random Forests/titanic_data/test.csv

~/Decision Trees and Random Forests/titanic_data/train.csv

Scenario

As part of your quest to optimize the *Titanic* survivors model, you want to keep pushing further and trying out different classification algorithms. The single decision tree you built earlier produced some promising results, but training multiple trees in an ensemble is usually an even better approach. So, you'll build a random forest that incorporates many decision trees in order to maximize the predictive skill of the model.

---

1. Open the lab and notebook.
   a) From the Coursera lab page, select **Open Lab** to open a Jupyter Notebook session in another web browser tab.
   b) Select **Decision Trees and Random Forests/RandomForests-Titanic.ipynb** to open it.

2. Prepare the dataset.
   a) Select the code cell beneath the **Import software libraries and load the dataset** title, then select **Run**.
   b) Verify that **test.csv** and **train.csv** are in the project folder, and that the latter was loaded with 891 records.

```
Libraries used in this project:
- Python 3.7.3 (default, Mar 27 2019, 22:11:17)
[GCC 7.3.0]
- NumPy 1.16.2
- pandas 0.24.2
- Matplotlib 3.0.3
- Seaborn 0.9.0
- scikit-learn 0.20.3

Data files in this project: ['test.csv', 'train.csv']
Loaded 891 records from ./titanic_data/train.csv.
```

   c) Scroll down and select the code cell beneath the **Split the datasets** title, then select **Run**.

```
The datasets and labels have been split.
```

3. **Why is cross-validation typically not necessary when training a random forest model?**

4. Perform common preparation on the training and validation sets.

a) Scroll down and select the code cell beneath the **Perform common preparation on the training and validation sets** title, then select **Run**.

Perform common preparation on the training and validation sets

```
# Perform common cleaning and feature engineering tasks on datasets.
def prep_dataset(dataset):

    # PROVIDE MISSING VALUES

    # Fill missing Age values with the median age.
    dataset['Age'].fillna(dataset['Age'].median(), inplace = True)

    # Fill missing Fare values with the median fare.
    dataset['Fare'].fillna(dataset['Fare'].median(), inplace = True)

    # Fill missing Embarked values with the mode.
    dataset['Embarked'].fillna(dataset['Embarked'].mode()[0], inplace = True)

    # ONE-HOT ENCODING

    cols = ['Pclass', 'Sex', 'Embarked']

    for i in cols:
        dummies = pd.get_dummies(dataset[i], prefix = i, drop_first = False)
        dataset = pd.concat([dataset, dummies], axis = 1)

    return dataset

X_train = prep_dataset(X_train.copy())

X_val = prep_dataset(X_val.copy())

print('The datasets have been cleaned and prepared.')
```

```
The datasets have been cleaned and prepared.
```

Since a random forest is just a collection of decision trees, you'll need to prepare the data in the same way (i.e., impute missing values and one-hot encode non-ordinal categorical values). This function, when called, does this same data preparation.

5. Drop columns that won't be used for training.

a) Scroll down and select the code cell beneath the **Drop columns that won't be used for training** title, then select **Run**.

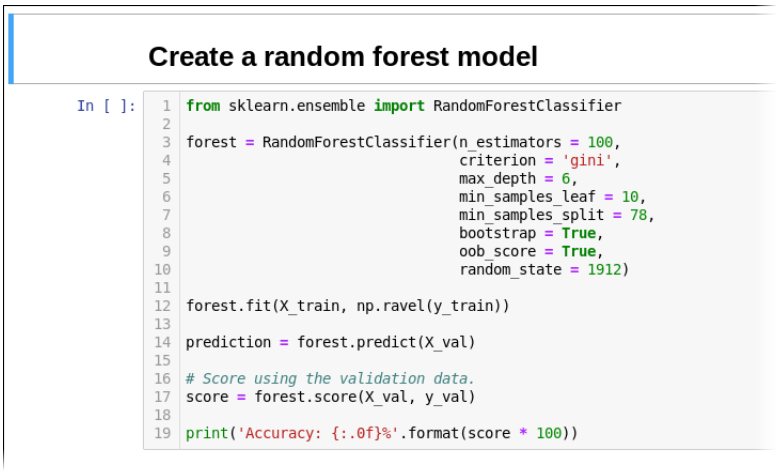
Drop columns that won't be used for training

```
# Drop unused columns from datasets.
def drop_unused(dataset):

    dataset = dataset.drop(['PassengerId'], axis = 1)
    dataset = dataset.drop(['Cabin'], axis = 1)
    dataset = dataset.drop(['Ticket'], axis = 1)
    dataset = dataset.drop(['Name'], axis = 1)

    # These have been replaced with one-hot encoding.
    dataset = dataset.drop(['Pclass'], axis = 1)
    dataset = dataset.drop(['Sex'], axis = 1)
    dataset = dataset.drop(['Embarked'], axis = 1)

    return dataset

X_train = drop_unused(X_train.copy())

X_val = drop_unused(X_val.copy())

print('Unused columns have been dropped.')
```

```
Unused columns have been dropped.
```

6. Create a random forest model.

a) Scroll down and view the cell titled **Create a random forest model**, and examine the code listing below it.



Create a random forest model

In [ ]:

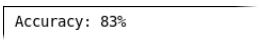
```
from sklearn.ensemble import RandomForestClassifier

forest = RandomForestClassifier(n_estimators = 100,
                                criterion = 'gini',
                                max_depth = 6,
                                min_samples_leaf = 10,
                                min_samples_split = 78,
                                bootstrap = True,
                                oob_score = True,
                                random_state = 1912)

forest.fit(X_train, np.ravel(y_train))

prediction = forest.predict(X_val)

# Score using the validation data.
score = forest.score(X_val, y_val)

print('Accuracy: {:.0f}%'.format(score * 100))
```

The `RandomForestClassifier()` class is scikit-learn's implementation of a decision tree ensemble. The class uses much of the same hyperparameters as a lone decision tree. In this case, the hyperparameters are set to the same optimized values that were identified earlier. However, there are three new hyperparameters/arguments:

- `n_estimators` specifies how many decision trees will be grown in the forest. Having more than a few hundred trees will usually just slow the training process down without providing any significant gains, so 100 should be adequate.
- `bootstrap` specifies whether or not bagging will be performed. `True`, the default value, means that it will be. If `False`, the entire training set is used for every tree.
- `oob_score` set to `True` tells the algorithm to compute accuracy scores using out-of-bag samples.

b) Select the cell that contains the code listing, then select **Run**.

c) Examine the output.

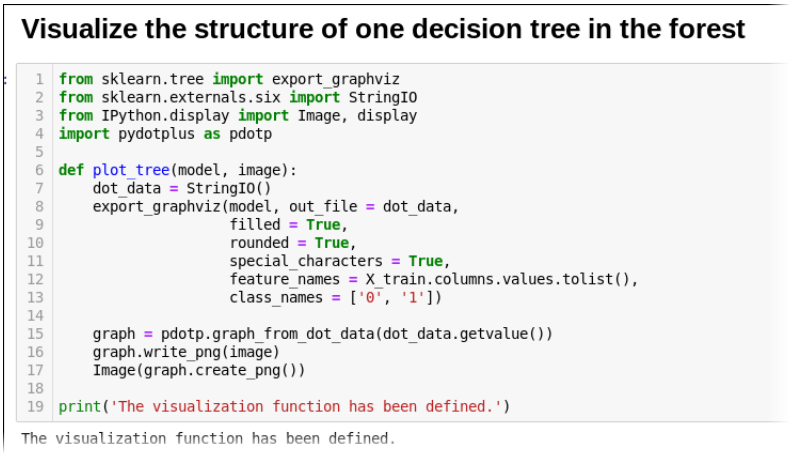


```
Accuracy: 83%
```

This initial random forest model has an accuracy of 83%. You'll evaluate the model using other metrics shortly.

7. Visualize the structure of one decision tree in the forest.

a) Scroll down and select the code cell beneath the **Visualize the structure of one decision tree in the forest** title, then select **Run**.



Visualize the structure of one decision tree in the forest

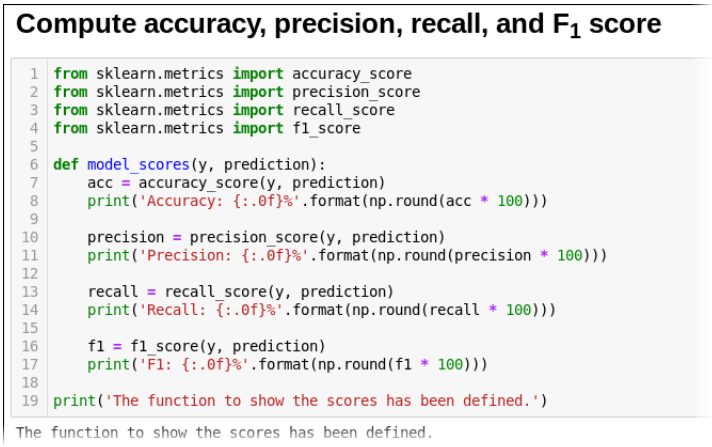
```
from sklearn.tree import export_graphviz
from sklearn.externals.six import StringIO
from IPython.display import Image, display
import pydotplus as pdotp

def plot_tree(model, image):
    dot_data = StringIO()
    export_graphviz(model, out_file = dot_data,
                    filled = True,
                    rounded = True,
                    special_characters = True,
                    feature_names = X_train.columns.values.tolist(),
                    class_names = ['0', '1'])

    graph = pdotp.graph_from_dot_data(dot_data.getvalue())
    graph.write_png(image)
    Image(graph.create_png())

print('The visualization function has been defined.')
```

```
The visualization function has been defined.
```

As before, this function, when called, will generate an image of a decision tree. However, since a random forest is made of many trees, you'll need to specify which trees to visualize.

8. Compute accuracy, precision, recall, and F_1 score.

a) Scroll down and select the code cell beneath the **Compute accuracy, precision, recall, and F_1 score** title, then select **Run**.



Compute accuracy, precision, recall, and F_1 score

```
from sklearn.metrics import accuracy_score
from sklearn.metrics import precision_score
from sklearn.metrics import recall_score
from sklearn.metrics import f1_score

def model_scores(y, prediction):
    acc = accuracy_score(y, prediction)
    print('Accuracy: {:.0f}%'.format(np.round(acc * 100)))

    precision = precision_score(y, prediction)
    print('Precision: {:.0f}%'.format(np.round(precision * 100)))

    recall = recall_score(y, prediction)
    print('Recall: {:.0f}%'.format(np.round(recall * 100)))

    f1 = f1_score(y, prediction)
    print('F1: {:.0f}%'.format(np.round(f1 * 100)))

print('The function to show the scores has been defined.')
```

```
The function to show the scores has been defined.
```

9. Generate a ROC curve and compute the AUC.

a) Scroll down and select the code cell beneath the **Generate a ROC curve and compute the AUC** title, then select **Run**.

Generate a ROC curve and compute the AUC

```
from sklearn.metrics import roc_curve
from sklearn.metrics import roc_auc_score

def roc(y, prediction_proba):
    fpr, tpr, thresholds = roc_curve(y, prediction_proba)

    plt.plot(fpr, tpr);
    plt.xlim([0.0, 1.0]);
    plt.ylim([0.0, 1.0]);
    plt.title('ROC Curve');
    plt.xlabel('False Positive Rate');
    plt.ylabel('True Positive Rate');
    plt.grid(True);

    auc = roc_auc_score(y, prediction_proba)
    print('Area Under Curve: {:.2f}'.format(auc))

print('The function to generate the ROC curve and compute AUC has been defined.')
```

```
The function to generate the ROC curve and compute AUC has been defined.
```

10. Generate a precision–recall curve and compute the average precision.

a) Scroll down and select the code cell beneath the **Generate a precision–recall curve and compute the average precision** title, then select **Run**.

Generate a precision–recall curve and compute the average precision

```
from sklearn.metrics import precision_recall_curve
from sklearn.metrics import average_precision_score

def prc(y, prediction_proba):
    precision, recall, thresholds = precision_recall_curve(y, prediction_proba)

    plt.plot(recall, precision);
    plt.xlim([0.0, 1.0]);
    plt.ylim([0.0, 1.0]);
    plt.title('Precision-Recall Curve');
    plt.xlabel('Recall');
    plt.ylabel('Precision');
    plt.grid(True);

    ap = average_precision_score(y, prediction_proba)
    print('Average Precision: {:.2f}'.format(ap))

print('The function to generate a PRC and compute average precision has been defined.')
```

```
The function to generate a PRC and compute average precision has been defined.
```

11. Examine a couple of trees in the forest.

a) Scroll down and select the code cell beneath the **Examine a couple of trees in the forest** title, then select **Run**.

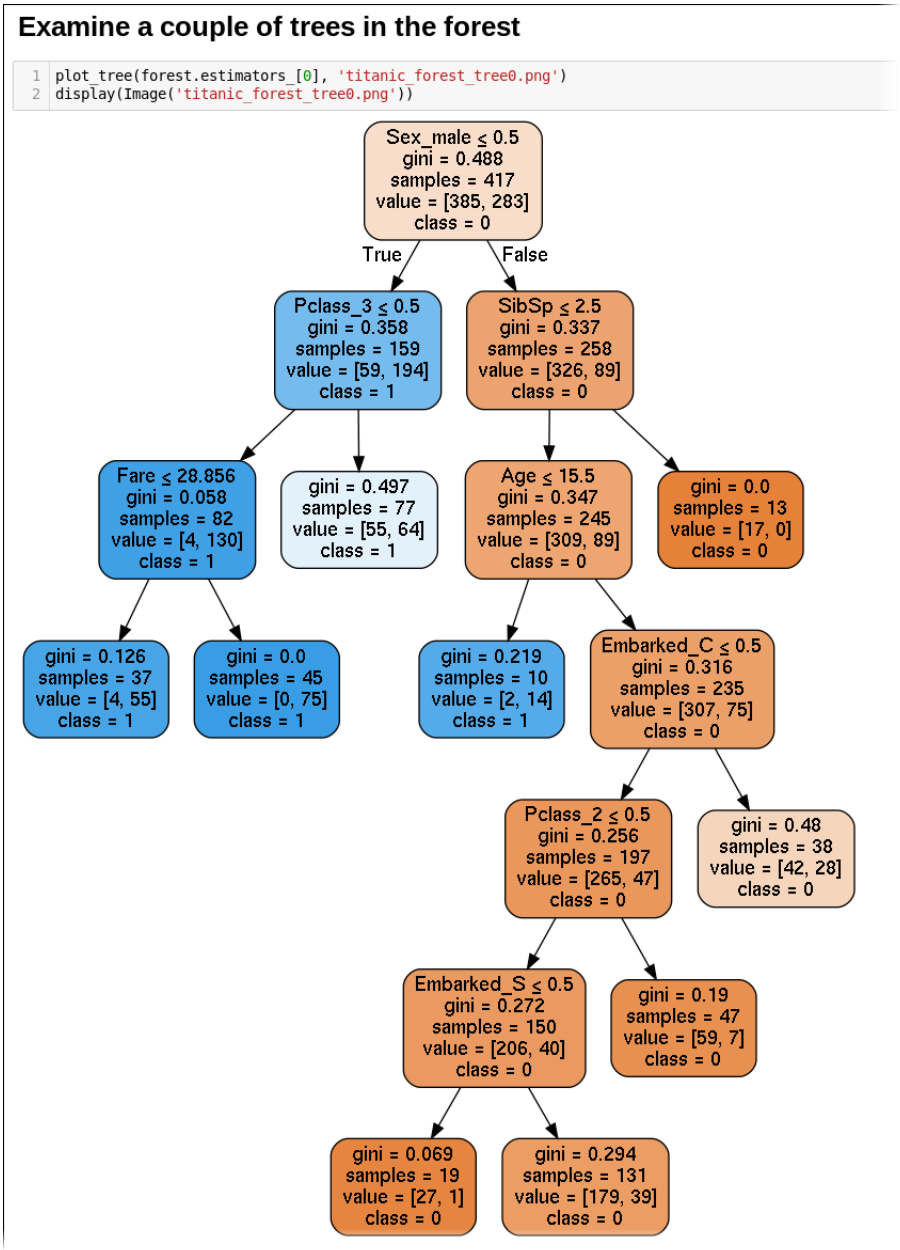
b) Wait for the `graphviz` library to download and install.

Note: The library may still be installed from when you ran this code in the previous lab.

c) Scroll down and select the next code listing cell.

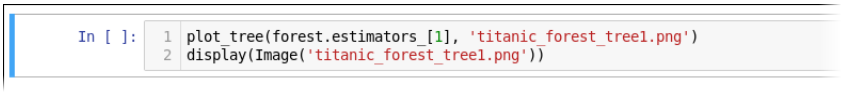
d) Select **Run**.

e) Examine the output.



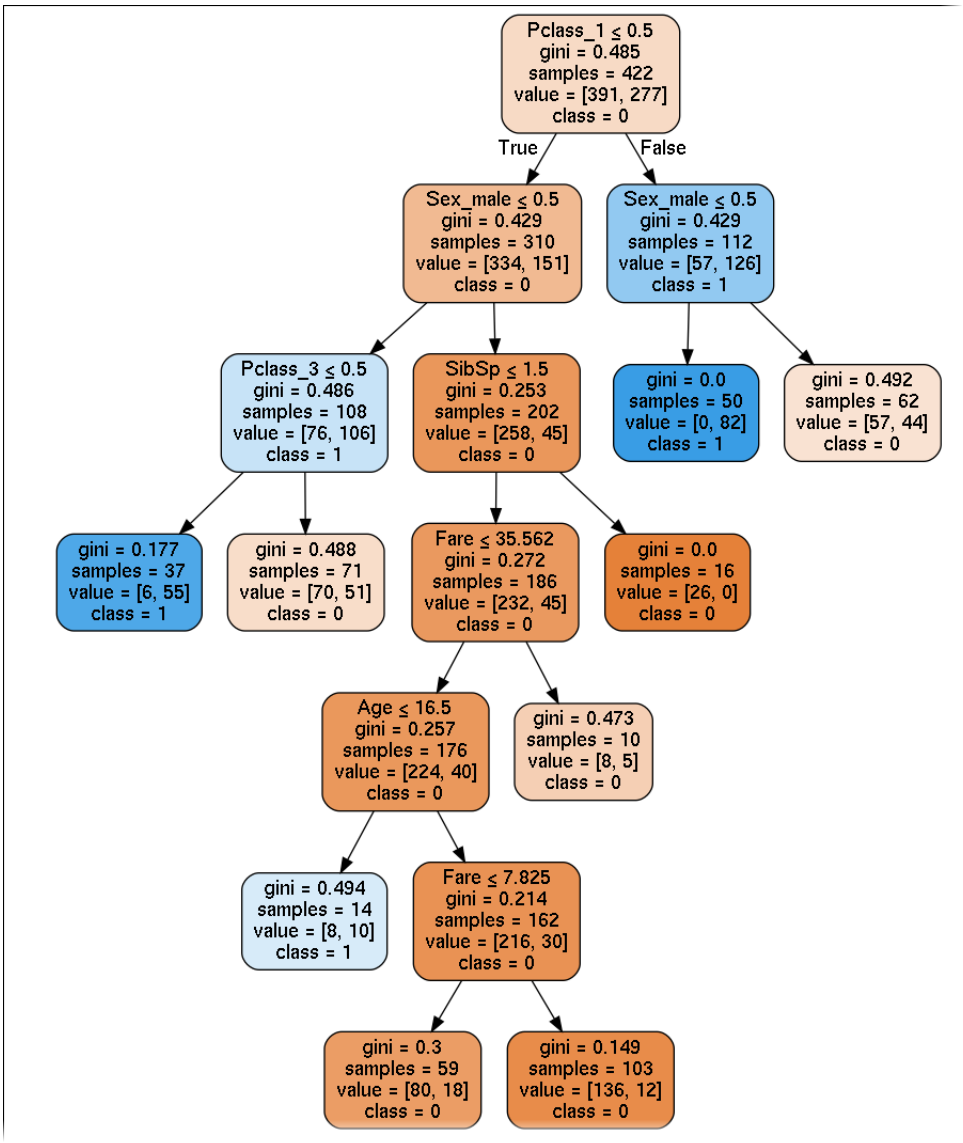
The `estimators_` attribute refers to each tree in the forest, so the tree at index 0 is being shown here.

f) Scroll down and select the next code listing cell.



g) Select **Run**.

h) Examine the output.



The tree at index 1 is being shown here.

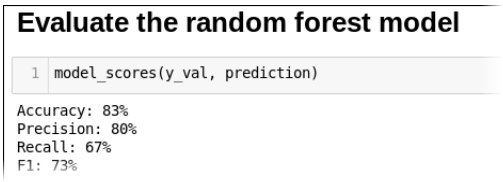
12. Compare the two trees.

How do the first two branches of each tree differ in terms of their splitting criteria?

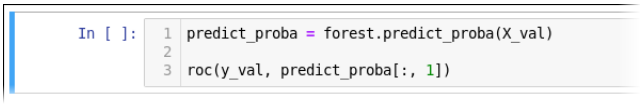
13. Evaluate the random forest model.

a) Scroll down and select the code cell beneath the **Evaluate the random forest model** title, then select **Run**.

b) Examine the output.

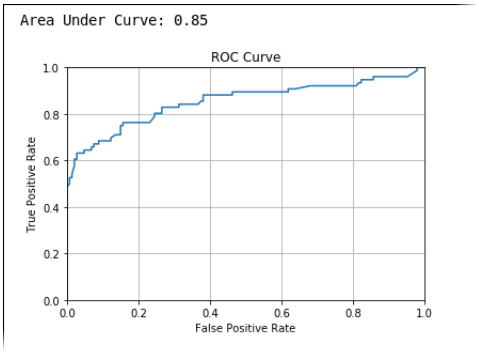


c) Scroll down and select the next code listing cell.

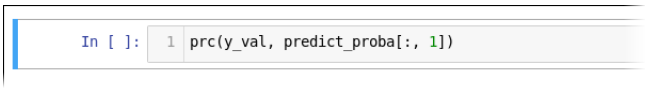


d) Select **Run**.

e) Examine the output.

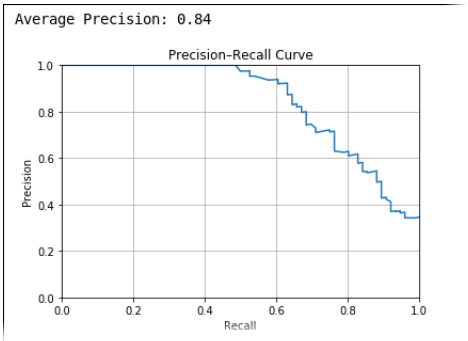


f) Scroll down and select the next code listing cell.



g) Select **Run**.

h) Examine the output.



i) Compare the random forest model with the results of the single optimized decision tree model you created earlier.

For reference, here are the results:

Model	*Accuracy*	*Precision*	*Recall*	F_1	*ROC AUC*	*Average Precision*
Single optimized decision tree	84%	92%	59%	72%	84%	80%
Random forest	83%	80%	67%	73%	85%	84%

Some of these metrics changed slightly (e.g., accuracy and F_1), whereas others change more dramatically (e.g., precision and recall). While the hyperparameters worked well for the one decision tree, they may not be optimal for the entire forest. You could conceivably run a randomized search on the forest to find better hyperparameters, but this will take a while depending on how many trees (`n_estimators`) are being grown in the forest.

Note: Growing even more trees in the forest may have led to better model skill, but would have been taxing on computing performance.

14. What advantage does a random forest have over a single decision tree?

15. Generate the out-of-bag error and decision function.

a) Scroll down and select the code cell beneath the **Generate the out-of-bag error and decision function** title, then select **Run**.

b) Examine the output.

```
Out-of-bag error: 0.22
```

In this case, 22% of the predictions were incorrect.

c) Scroll down and select the next code listing cell.

```
In [ ]:  1  forest.oob_decision_function_
```

d) Select **Run**.

e) Examine the output.

```
array([[0.83990492, 0.16009508],
       [0.44309796, 0.55690204],
       [0.88436021, 0.11563979],
       ...,
       [0.72455935, 0.27544065],
       [0.43065855, 0.56934145],
       [0.18766646, 0.81233354]])
```

This is an array of probability predictions for each out-of-bag example. The first column indicates the probability for class 0 (perished) and the second column indicates the probability for class 1 (survived). Like with any other binary classifier, the class with the highest probability is used as the prediction.

16. Verify that the random forest took the majority vote of all trees in the forest.

a) Scroll down and select the code cell beneath the **Verify that the random forest took the majority vote of all trees in the forest** title, then select **Run**.

b) Examine the output.

Verify that the random forest took the majority vote of all trees in the forest

```
from statistics import mode

class_list = []
class_mode = []
i = 0

for i in range(10):  # Get predictions for the first 10 data examples.
    for tree in forest.estimators_:  # Get predictions from all trees in forest.
        predict = tree.predict(X_val)
        predict_examples = predict[i]
        class_list.append(predict_examples)

    # Get mode of predictions to determine majority vote for each example.
    class_mode.append(int(mode(class_list)))
    class_list = []

print('Majority vote classification:    {}'.format(np.array(class_mode)))
print('Random forest classification:    {}'.format(forest.predict(X_val)[0:10]))
```

```
Majority vote classification:    [0 0 0 1 0 1 1 0 0 0]
Random forest classification:    [0 0 0 1 0 1 1 0 0 0]
```

The purpose of this code block is to demonstrate that the random forest, for each data example, returns the majority vote across all of the trees in the forest. As you can see, manually taking this vote gives you the same results as automatically calling the forest's `predict()` method.

17. Identify important features.

a) Scroll down and select the code cell beneath the **Identify important features** title, then select **Run**.

b) Examine the output.

Identify important features

```
# Get feature importances.
feature_importances = list(forest.feature_importances_)

# Get column names.
feature_list = list(X_train)

importances_list = []

# Map feature/importance indices and put into a single list.
for feature, importance in zip(feature_list, feature_importances):
    importances_list.append((feature, round(importance, 2)))

# Sort list by importance index.
importances_list = sorted(importances_list, key = lambda i: i[1], reverse = True)

for feature, importance in importances_list:
    print('Feature: {:10} | Importance: {}'.format(feature, importance))
```

```
Feature: Sex_female | Importance: 0.34
Feature: Sex_male   | Importance: 0.25
Feature: Pclass_3   | Importance: 0.11
Feature: Fare       | Importance: 0.1
Feature: Pclass_1   | Importance: 0.06
Feature: Age        | Importance: 0.04
Feature: SibSp      | Importance: 0.02
Feature: Parch      | Importance: 0.02
Feature: Pclass_2   | Importance: 0.02
Feature: Embarked_C | Importance: 0.02
Feature: Embarked_S | Importance: 0.01
Feature: Embarked_Q | Importance: 0.0
```

The `feature_importances_` attribute returns a ranking of each feature's contribution to the class prediction in the random forest. All of the features combined add up to 1, and higher values indicate more importance.

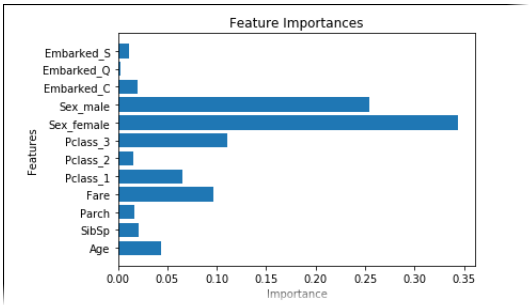
- It looks like the sex of the passenger, particularly if they were female, contributed the most to the predictions.
- Whether or not a passenger was in third class, as well as the passenger's fare, also seemed to have some importance.
- Analyzing this data visually might be more helpful.

c) Scroll down and select the next code listing cell.

```
In [ ]:
y_values = list(range(len(feature_importances)))
plt.barh(y_values, feature_importances)
plt.yticks(y_values, feature_list, rotation='horizontal')
plt.xlabel('Importance')
plt.ylabel('Features')
plt.title('Feature Importances');
```

d) Select **Run**.

e) Examine the output.



This visual plot helps to reinforce which features are truly important. Likewise, it can help you decide which features to keep and which to drop when you retrain the forest.

- `Sex_male`, `Sex_female`, `Pclass_3`, and `Fare` all seem relatively important, so you'll keep them.
- `Pclass_1` and maybe even `Age` still have some importance, though perhaps not to any degree that it would impact the model all that much. You'll drop these, along with the rest of the features not mentioned.

18. Select the most important features to use for a new round of training.

a) Scroll down and select the code cell beneath the **Select the most important features to use for a new round of training** title, then select **Run**.

b) Examine the output.

Select the most important features to use for a new round of training

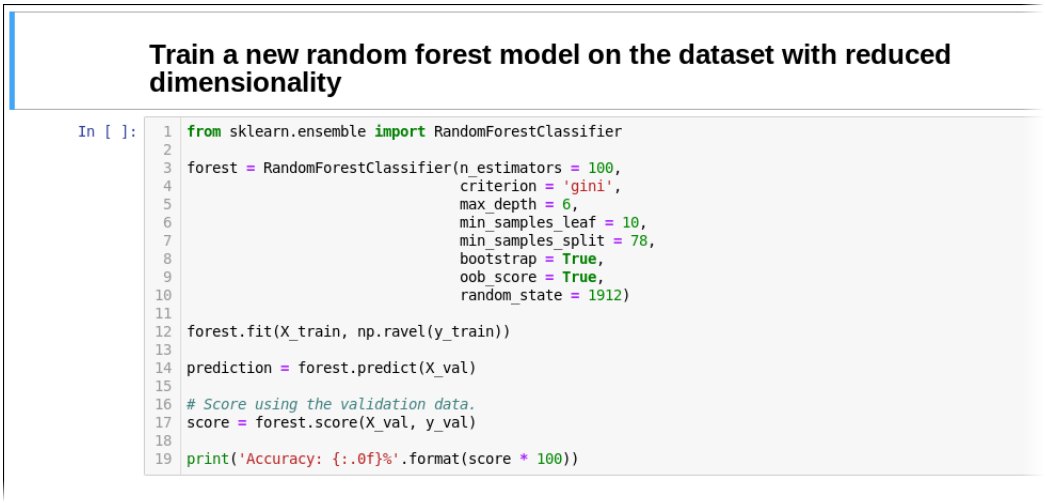
```
X_train = X_train[['Sex_female', 'Sex_male', 'Pclass_3', 'Fare']].copy()
X_val = X_val[['Sex_female', 'Sex_male', 'Pclass_3', 'Fare']].copy()

X_train.head()
```

	Sex_female	Sex_male	Pclass_3	Fare
439	0	1	0	10.5000
617	1	0	1	16.1000
242	0	1	0	10.5000
82	1	0	1	7.7875
398	0	1	0	10.5000

The training and validation sets now only include the features that were identified as having high importance, reducing the datasets' dimensionality.

19. Train a new random forest model on the dataset with reduced dimensionality.

a) Scroll down and view the cell titled **Train a new random forest model on the dataset with reduced dimensionality**, and examine the code listing below it.



Train a new random forest model on the dataset with reduced dimensionality

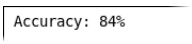
In [ ]:

```
from sklearn.ensemble import RandomForestClassifier

forest = RandomForestClassifier(n_estimators = 100,
                                criterion = 'gini',
                                max_depth = 6,
                                min_samples_leaf = 10,
                                min_samples_split = 78,
                                bootstrap = True,
                                oob_score = True,
                                random_state = 1912)

forest.fit(X_train, np.ravel(y_train))

prediction = forest.predict(X_val)

# Score using the validation data.
score = forest.score(X_val, y_val)

print('Accuracy: {:.0f}%'.format(score * 100))
```

This is the same training code as before, but now it's using the newly reduced datasets.

b) Select the cell that contains the code listing, then select **Run**.

c) Examine the output.



```
Accuracy: 84%
```

The model's accuracy has increased slightly.

20. Evaluate the new random forest model.

a) Scroll down and select the code cell beneath the **Evaluate the new random forest model** title, then select **Run**.

b) Examine the output.

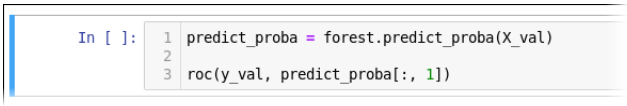
Evaluate the new random forest model

```
model_scores(y_val, prediction)
```

```
Accuracy: 84%
Precision: 82%
Recall: 67%
F1: 74%
```

Compared to the previous model, the new model's accuracy, precision, and F_1 score have improved slightly.

c) Scroll down and select the next code listing cell.

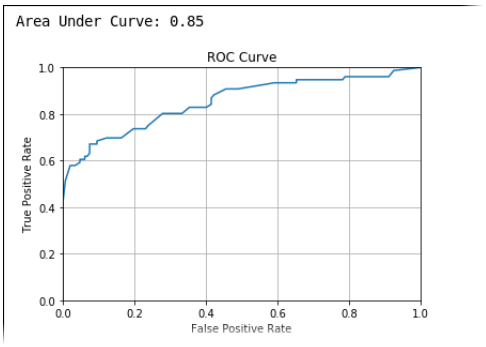


In [ ]:

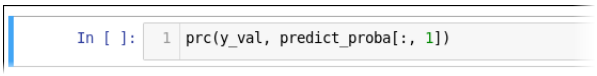
```
predict_proba = forest.predict_proba(X_val)

roc(y_val, predict_proba[:, 1])
```

d) Select **Run**.

e) Examine the output.

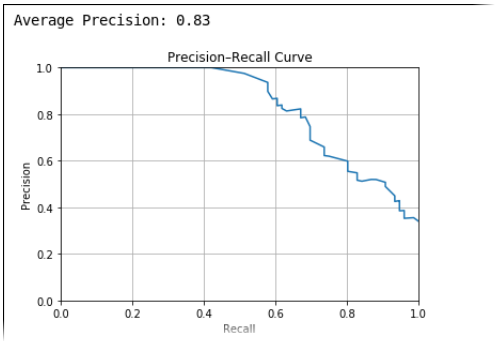


f) Scroll down and select the next code listing cell.



g) Select **Run**.

h) Examine the output.

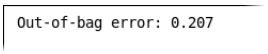


The ROC AUC seems to have stayed the same, whereas the average precision decreased slightly.

i) Scroll down and select the next code listing cell.

j) Select **Run**.

k) Examine the output.



The out-of-bag error improved on this reduced model.

21. The changes to these scores were largely the result of reducing the datasets' dimensionality.

    How might you retrain the model to improve these scores even further?

22. Shut down the notebook and close the lab.
    a) From the menu, select **Kernel→Shutdown**.
    b) In the **Shutdown kernel?** dialog box, select **Shutdown**.
    c) Close the lab browser tab and continue on with the course.

MODULE 2

Build Support-Vector Machines (SVM)

The following labs are for Module 2: Build Support-Vector Machines (SVM).

LAB 4-3
Building an SVM Model for Classification

Data File

~/SVMs/SVMs-Iris.ipynb

Scenario

Another department at the college has expressed interest in your machine learning classification models. A Botany professor wants to demonstrate how the biological components that make up plant life can determine a plant's taxonomic ranks (family, genus, species, etc.). The professor provides you with a dataset of plants and some of their physical attributes. In speaking with this professor, you learn that the dataset may contain several outliers. Rather than create a logistic regression or decision tree-based model, you decide to create a classifier using SVMs. This will hopefully lead to a model with more skill when it comes to classifying data with outliers.

---

1. Open the lab and notebook.
   a) From the Coursera lab page, select **Open Lab** to open a Jupyter Notebook session in another web browser tab.
   b) Select **SVMs/SVMs-Iris.ipynb** to open it.

2. Load the dataset.
   a) Select the code cell beneath the **Import software libraries and load the dataset** title, then select **Run**.
   b) Verify that 150 records were loaded.

      This dataset, which comes pre-packaged with scikit-learn, is one of the most well-known datasets in machine learning. It includes some physical attributes of three types of flowers within the *Iris* genus.

3. Get acquainted with the dataset.
   a) Scroll down and select the code cell beneath the **Get acquainted with the dataset**, then select **Run**.

b) Examine the output.

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 150 entries, 0 to 149
Data columns (total 5 columns):
sepal length (cm)    150 non-null float64
sepal width (cm)     150 non-null float64
petal length (cm)    150 non-null float64
petal width (cm)     150 non-null float64
target               150 non-null int64
dtypes: float64(4), int64(1)
memory usage: 5.9 KB
None
```

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	target
0	5.1	3.5	1.4	0.2	0
1	4.9	3.0	1.4	0.2	0
2	4.7	3.2	1.3	0.2	0
3	4.6	3.1	1.5	0.2	0
4	5.0	3.6	1.4	0.2	0
5	5.4	3.9	1.7	0.4	0
6	4.6	3.4	1.4	0.3	0
7	5.0	3.4	1.5	0.2	0
8	4.4	2.9	1.4	0.2	0
9	4.9	3.1	1.5	0.1	0

- The training set includes 150 rows and 5 columns.
- All of the columns contain float values, except for the `target` column, which contains integer values. The `target` column is the label of *Iris* species the model must predict, classified as 0, 1, or 2. Each species' label is as follows:
  - *Iris setosa* (0)
  - *Iris versicolor* (1)
  - *Iris virginica* (2)
- There is no missing data; all rows have values for every column.
- The columns describe the length and width of the flower's sepal (a part of the flower that supports the petals) and the length and width of the flower's petals.

4. Examine a general summary of statistics.

a) Scroll down and select the code cell beneath the **Examine a general summary of statistics** title, then select **Run**.

b) Examine the output.

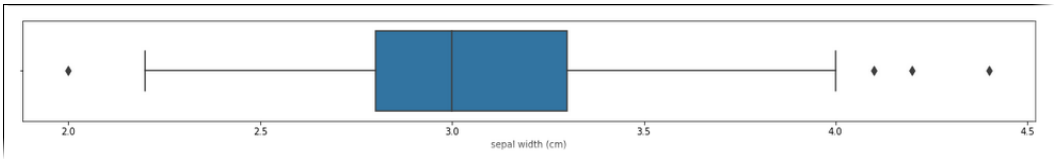
```
       sepal length (cm)  sepal width (cm)  petal length (cm)  \
count             150.00            150.00             150.00
mean                5.84              3.06               3.76
std                 0.83              0.44               1.77
min                 4.30              2.00               1.00
25%                 5.10              2.80               1.60
50%                 5.80              3.00               4.35
75%                 6.40              3.30               5.10
max                 7.90              4.40               6.90

       petal width (cm)  target
count            150.00  150.00
mean               1.20    1.00
std                0.76    0.82
min                0.10    0.00
25%                0.30    0.00
50%                1.30    1.00
75%                1.80    2.00
max                2.50    2.00
```

- This is a very simple and relatively clean dataset. Not much feature engineering needs to be done.
- However, there may be a few outliers that could impact a classification model's skill.

5. Identify outliers.

a) Scroll down and select the code cell beneath the **Identify outliers** title, then select **Run**.

b) Examine the output.

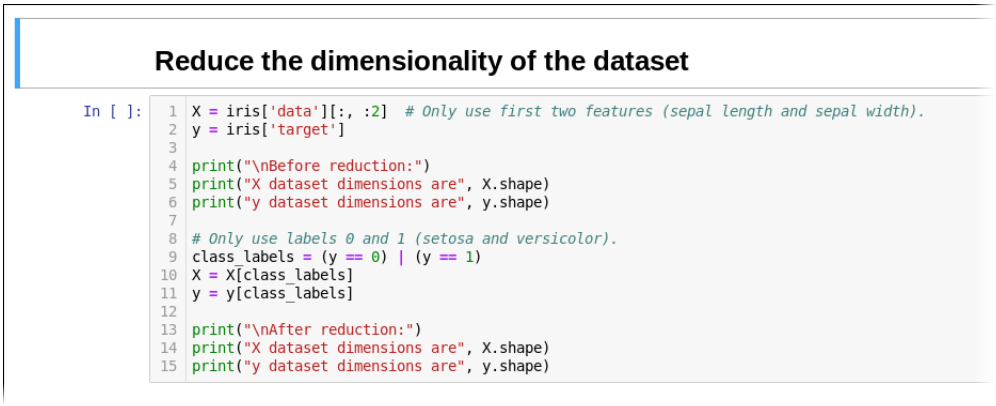


There appear to be a few outliers for sepal width.

6. Why are SVMs often better than other algorithms at handling datasets with outliers?

7. Reduce the dimensionality of the dataset.

a) Scroll down and view the cell titled **Reduce the dimensionality of the dataset**, and examine the code listing below it.



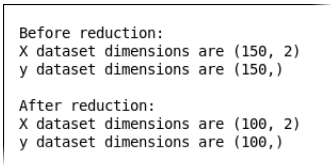
Reduce the dimensionality of the dataset

In [ ]:

```
X = iris['data'][:, :2]  # Only use first two features (sepal length and sepal width).
y = iris['target']

print("\nBefore reduction:")
print("X dataset dimensions are", X.shape)
print("y dataset dimensions are", y.shape)

# Only use labels 0 and 1 (setosa and versicolor).
class_labels = (y == 0) | (y == 1)
X = X[class_labels]
y = y[class_labels]

print("\nAfter reduction:")
print("X dataset dimensions are", X.shape)
print("y dataset dimensions are", y.shape)
```

For demonstration purposes, you'll start by looking at only two of the four features (sepal length and sepal width), and considering only two of the three *Iris* species (*setosa* and *versicolor*). Later, you'll train the model on the full dataset.

b) Select the cell that contains the code listing, then select **Run**.



```
Before reduction:
X dataset dimensions are (150, 2)
y dataset dimensions are (150,)

After reduction:
X dataset dimensions are (100, 2)
y dataset dimensions are (100,)
```

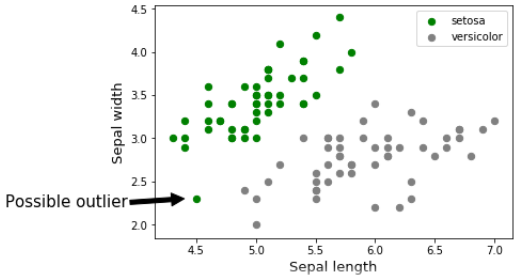
8. Examine the separation between classes using a scatter plot.

a) Scroll down and select the code cell beneath the **Examine the separation between classes using a scatter plot** title, then select **Run**.

b) Examine the output.

Examine the separation between classes using a scatter plot

```
# Sepal length along x-axis, sepal width along y-axis.
scatter_x = X[:, 0]
scatter_y = X[:, 1]

cdict = {0: 'green', 1: 'grey'}

# Generate scatter plot with legend.
for c_label in np.unique(y):
    if c_label == 0:
        iris = 'setosa'
    if c_label == 1:
        iris = 'versicolor'

    ix = np.where(y == c_label)
    plt.scatter(scatter_x[ix], scatter_y[ix], c = cdict[c_label], label = iris, s = 40)

plt.legend()
plt.xlabel("Sepal length", fontsize = 13)
plt.ylabel("Sepal width", fontsize = 13)
plt.annotate('Possible outlier', xy = (4.4, 2.3), xytext = (2.9, 2.2),
             arrowprops = dict(color= 'black'), fontsize = 15);
```



This is a basic scatter plot of the two features extracted earlier (sepal length and sepal width).

- The green dots in the top left are the data points for *setosa* flowers.
- The grey dots in the bottom right are the data points for *versicolor* flowers.
- For the most part, these two features seem reasonably separable by a straight decision boundary.
- At least one potential outlier is being called out; this might get misclassified.

9. Plot a decision boundary for a given model.

a) Scroll down and view the cell titled **Plot a decision boundary for a given model**, and examine the code listing below it.

```
1  def plot_decision_boundary(X, y, model, is_svm):
2      scatter_x = X[:, 0]
3      scatter_y = X[:, 1]
4
5      cdict = {0: 'green', 1: 'grey'}
6
7      for c_label in np.unique(y):
8          if c_label == 0:
9              iris = 'setosa'
10         if c_label == 1:
11             iris = 'versicolor'
12
13         ix = np.where(y == c_label)
14         plt.scatter(scatter_x[ix], scatter_y[ix], c = cdict[c_label], label = iris, s = 40)
15
16     plt.legend()
17     plt.xlabel("Sepal length", fontsize = 13)
18     plt.ylabel("Sepal width", fontsize = 13)
19
20     ax = plt.gca()
21     xlim = ax.get_xlim()
22     ylim = ax.get_ylim()
23
24     # Create grid.
25     xx = np.linspace(xlim[0], xlim[1], 40)
26     yy = np.linspace(ylim[0], ylim[1], 40)
27     YY, XX = np.meshgrid(yy, xx)
28     xy = np.vstack([XX.ravel(), YY.ravel()]).T
29     Z = model.decision_function(xy).reshape(XX.shape)  # Use model decision function to plot boundary.
30
```

This function, when called, will plot a decision boundary on the scatter plot shown previously.

- The function requires the training data, label data, and model object as input.
- The `is_svm` argument will determine whether or not to plot the margins (for SVM models).
- Lines 2 through 18 create the scatter plot, same as before.
- Lines 20 through 29 begin creating a grid on which to plot the model's decision function.

```
31     if is_svm == True:
32         # Plot decision boundary and margins.
33         ax.contour(XX, YY, Z, colors = 'r', levels = [-1, 0, 1],
34                    linestyles=['--', '-', '--'])
35
36         # Plot support vectors.
37         ax.scatter(model.support_vectors_[:, 0], model.support_vectors_[:, 1],
38                    s = 100, linewidth = 1, facecolors = 'none', edgecolors = 'k')
39     else:
40         ax.contour(XX, YY, Z, colors = 'r', levels = [0],
41                    linestyles=['-'])
42
43     plt.show()
44
45 print('Function to plot the decision boundary has been defined.')
```

- Lines 31 through 38 plot the decision boundary line, then plot the support-vector margins for SVM models.
- Lines 39 through 41 just plot the decision boundary for non-SVM models.
- Line 45 shows a message when the function has been defined.

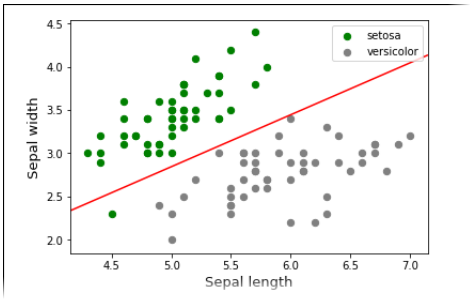
b) Select the cell that contains the code listing, then select **Run**.

```
Function to plot the decision boundary has been defined.
```

10. Train a basic logistic regression model and plot its decision boundary.

a) Scroll down and select the code cell beneath the **Train a basic logistic regression model and plot its decision boundary** title, then select **Run**.

b) Examine the output.

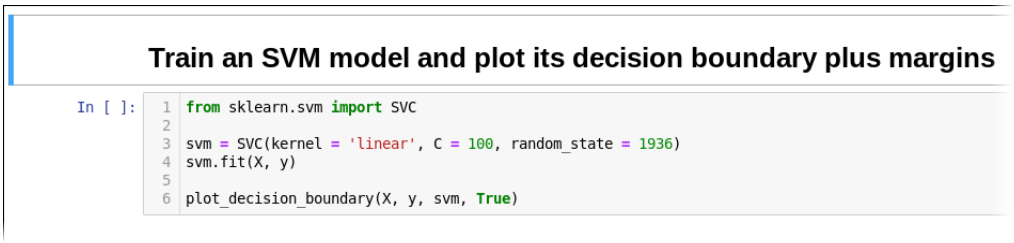


The decision boundary, for the most part, seems to cleanly divide the classes. However:

- The boundary comes pretty close to the edge data examples, potentially leading to overfitting.
- The outlier mentioned earlier has been misclassified.

11. Train an SVM model and plot its decision boundary plus margins.

a) Scroll down and view the cell titled **Train an SVM model and plot its decision boundary plus margins**, and examine the code listing below it.



Train an SVM model and plot its decision boundary plus margins

In [ ]:

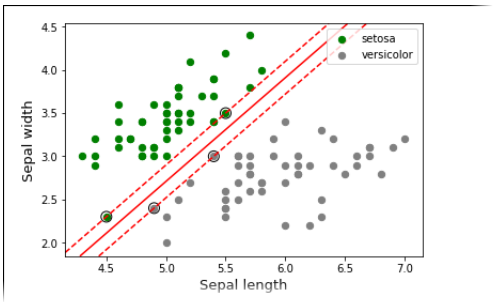
```
from sklearn.svm import SVC

svm = SVC(kernel = 'linear', C = 100, random_state = 1936)
svm.fit(X, y)

plot_decision_boundary(X, y, svm, True)
```

The `SVC()` class implements support-vector classification. In this case, there are two hyperparameters being set:

- `kernel` specifies the kernel method to use; in this case, you'll start with a linear kernel.
- `C` is the regularization penalty that determines the "wideness" of the road (i.e., its margins). A higher penalty leads to narrower margins. You're starting rather high with a value of 100.

b) Select the cell that contains the code listing, then select **Run**.

c) Examine the output.

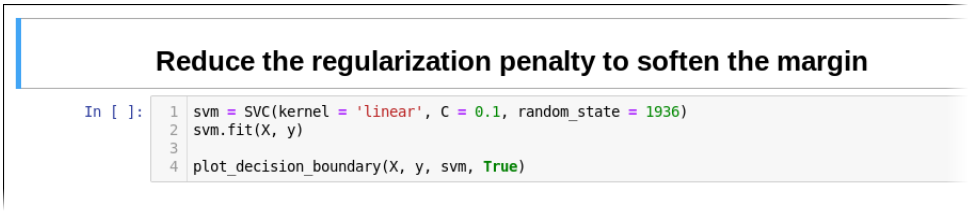


- The solid red line is the decision boundary.
- The dashed red lines are the support-vector margins.
- The circled data points are the support vectors.

12. **How does this SVM boundary fit to the data as compared to the logistic regression boundary?**

13. Reduce the regularization penalty to soften the margin.

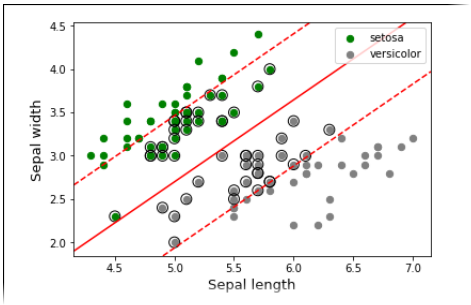
a) Scroll down and view the cell titled **Reduce the regularization penalty to soften the margin**, and examine the code listing below it.



While the previous SVM model seemed to do well as compared to logistic regression, the fact that the margins were rather narrow means that one or more outliers may potentially lead to overfitting. So, you'll plot another model where the regularization penalty is much lower, leading to softer margins.

b) Select the cell that contains the code listing, then select **Run**.

c) Examine the output.



The margins are much wider now, and more data examples are placed within those margins. This is not necessarily better for this particular case, and may actually lead to underfitting; however, it serves to demonstrate how softening the margins of an SVM model can change its classification decisions. You'll search for the optimal `C` value shortly, when you train on the full dataset.

14. Split the datasets.

a) Scroll down and view the cell titled **Split the datasets**, and examine the code listing below it.

Split the datasets

In [ ]:
```
from sklearn.model_selection import train_test_split

label_columns = ['target']

training_columns = ['sepal length (cm)', 'sepal width (cm)' , 'petal length (cm)', 'petal width (cm)']

# Split the training and test datasets and their labels.
X_train, X_test, y_train, y_test = train_test_split(data_raw[training_columns],
                                                    data_raw[label_columns],
                                                    random_state = 1936)

print('The training and test datasets and their labels have been split.')
```

This splits the datasets in the usual way, incorporating the entire dataset this time (i.e., all four features and all three labels).

b) Select the cell that contains the code listing, then select **Run**.

```
The training and test datasets and their labels have been split.
```

15. Evaluate an SVM model using a holdout test set.

a) Scroll down and view the cell titled **Evaluate an SVM model using a holdout test set**, and examine the code listing below it.

Evaluate an SVM model using a holdout test set

In [ ]:
```
svm = SVC(kernel = 'linear', C = 100, random_state = 1936)
svm.fit(X_train, np.ravel(y_train))

# Score using the test data.
score = svm.score(X_test, y_test)

print('Accuracy: {:.0f}%'.format(score * 100))
```

To start with, you'll just use the arbitrary value of 100 for the regularization penalty.

b) Select the cell that contains the code listing, then select **Run**.

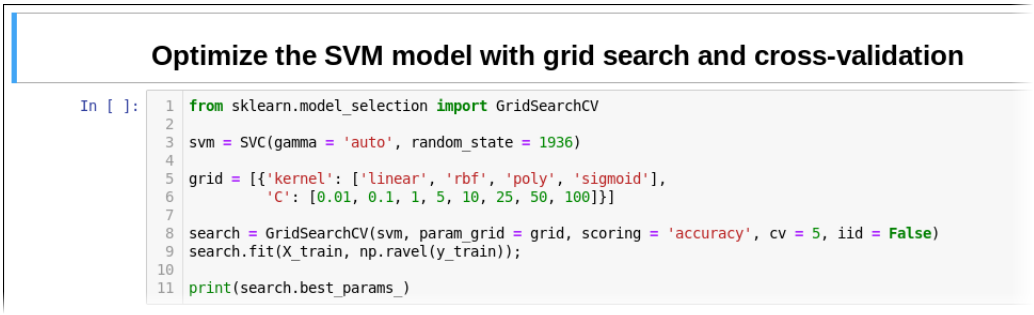
c) Examine the output.

```
Accuracy: 92%
```

- The accuracy for this initial model is 92%.
- You could evaluate the model using all of the usual metrics—precision, recall, F_1 score, etc.—but for this simple dataset, accuracy should suffice.

16. Optimize the SVM model with grid search and cross-validation.

a) Scroll down and view the cell titled **Optimize the SVM model with grid search and cross-validation**, and examine the code listing below it.



Optimize the SVM model with grid search and cross-validation

In [ ]:

```
1  from sklearn.model_selection import GridSearchCV
2
3  svm = SVC(gamma = 'auto', random_state = 1936)
4
5  grid = [{'kernel': ['linear', 'rbf', 'poly', 'sigmoid'],
6           'C': [0.01, 0.1, 1, 5, 10, 25, 50, 100]}]
7
8  search = GridSearchCV(svm, param_grid = grid, scoring = 'accuracy', cv = 5, iid = False)
9  search.fit(X_train, np.ravel(y_train));
10
11 print(search.best_params_)
```

This code performs a grid search to determine the best hyperparameters for an SVM model.

- On line 3, the model will default to using a `gamma` of `auto`. Gamma determines the coefficient to use with non-linear kernels (in this case, 1 divided by the number of features).
- On line 5, the grid search will identify the best kernel to use among the following four:
  - `linear` is the linear kernel.
  - `rbf` is the Gaussian radial basis function (RBF) kernel.
  - `poly` is the polynomial kernel.
  - `sigmoid` is the sigmoid kernel.
- On line 6, the grid search will try these kernels against a list of several `C` values, ranging from 0.01 (very soft margins) to 100 (very hard margins).
- On line 8, the grid search will be optimizing for accuracy and will perform five-fold cross-validation on the training data.

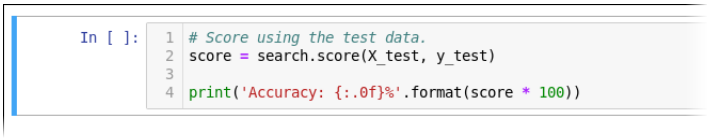
b) Select the cell that contains the code listing, then select **Run**.

c) Examine the output.

```
{'C': 0.01, 'kernel': 'poly'}
```

The grid search identified a low `C` value with the polynomial kernel as the optimal hyperparameter combination.

d) Scroll down and select the next code listing cell.



In [ ]:

```
1  # Score using the test data.
2  score = search.score(X_test, y_test)
3
4  print('Accuracy: {:.0f}%'.format(score * 100))
```

e) Select **Run**.

f) Examine the output and confirm that the model's accuracy increased.

```
Accuracy: 95%
```

17. Examine the optimized SVM model's predictions.

a) Scroll down and select the code cell beneath the **Examine the optimized SVM model's predictions** title, then select **Run**.

b) Examine the output.

Examine the optimized SVM model's predictions

```
# Use test set to evaluate.
results_comparison = X_test.copy()
results_comparison['Predicted Iris'] = search.predict(X_test)
results_comparison['Actual Iris'] = y_test.copy()

# Map labels to actual Iris names.
iris_encode = {0: 'setosa', 1: 'versicolor', 2: 'virginica'}

results_comparison['Predicted Iris'] = results_comparison['Predicted Iris'].map(iris_encode)
results_comparison['Actual Iris'] = results_comparison['Actual Iris'].map(iris_encode)

# View examples of the predictions compared to actual Iris.
results_comparison.head(20)
```

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	Predicted Iris	Actual Iris
121	5.6	2.8	4.9	2.0	virginica	virginica
67	5.8	2.7	4.1	1.0	versicolor	versicolor
148	6.2	3.4	5.4	2.3	virginica	virginica
77	6.7	3.0	5.0	1.7	versicolor	versicolor
31	5.4	3.4	1.5	0.4	setosa	setosa
7	5.0	3.4	1.5	0.2	setosa	setosa
5	5.4	3.9	1.7	0.4	setosa	setosa
127	6.1	3.0	4.9	1.8	virginica	virginica
146	6.3	2.5	5.0	1.9	virginica	virginica
35	5.0	3.2	1.2	0.2	setosa	setosa
110	6.5	3.2	5.1	2.0	virginica	virginica
29	4.7	3.2	1.6	0.2	setosa	setosa
126	6.2	2.8	4.8	1.8	virginica	virginica
143	6.8	3.2	5.9	2.3	virginica	virginica
58	6.6	2.9	4.6	1.3	versicolor	versicolor
47	4.6	3.2	1.4	0.2	setosa	setosa
90	5.5	2.6	4.4	1.2	versicolor	versicolor
34	4.9	3.1	1.5	0.2	setosa	setosa
92	5.8	2.6	4.0	1.2	versicolor	versicolor
83	6.0	2.7	5.1	1.6	virginica	versicolor

This seems to confirm the high accuracy that was just reported. The first misclassification appears on the last line that was output (example index **83**).

18. Shut down the notebook and close the lab.

a) From the menu, select **Kernel→Shutdown**.

b) In the **Shutdown kernel?** dialog box, select **Shutdown**.

c) Close the lab browser tab and continue on with the course.

LAB 4-4
Building an SVM Model for Regression

Data File

~/SVMs/SVMs-Boston.ipynb

Scenario

Thus far, you've trained the Boston housing dataset on various linear regression models. These models have helped you predict the median value of a house in a particular area, given a number of factors. However, you've noticed that there are some outliers in the dataset that may be negatively affecting the model's predictive skill. There are various ways of addressing outliers, but one of the most effective is using SVMs. So, you'll retrain the dataset, this time on SVM regression models, to see if you can improve your ability to predict housing prices.

---

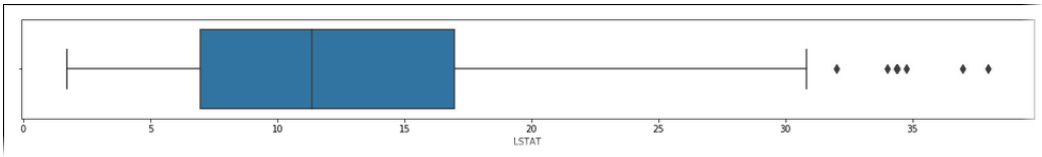
1. Open the lab and notebook.
   a) From the Coursera lab page, select **Open Lab** to open a Jupyter Notebook session in another web browser tab.
   b) Select **SVMs/SVMs-Boston.ipynb** to open it.

2. Load the dataset.
   a) Select the code cell beneath the **Import software libraries and load the dataset** title, then select **Run**.
   b) Verify that 506 records were loaded.

3. Drop columns that won't be used for training.
   a) Scroll down and select the code cell beneath the **Drop columns that won't be used for training** title, then select **Run**.
   b) Examine the output.

      Recall that, when you trained a linear regression model on this dataset, you dropped the `CHAS` feature ("Charles River dummy variable") because of its weak correlation with the target and because it's a categorical variable. You're doing the same here.

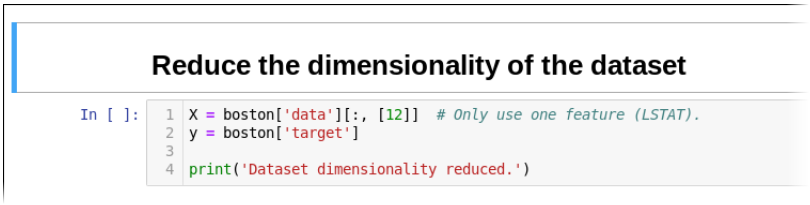
4. Identify outliers.
   a) Scroll down and select the code cell beneath the **Identify outliers** title, then select **Run**.
   b) Examine the output.



   - `LSTAT` is the percentage of the population in an area that qualifies as "low status."
   - There appear to be a few outliers toward the higher end of the range of values.
   - The percentage of low-status households is usually below 30% for a given area.

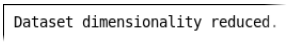
5. Reduce the dimensionality of the dataset.

a) Scroll down and view the cell titled **Reduce the dimensionality of the dataset**, and examine the code listing below it.



For demonstration purposes, you'll start by looking at only one feature (`LSTAT`) as compared to the target label (median house value, or `MEDV`). Later, you'll train the model on the full dataset.

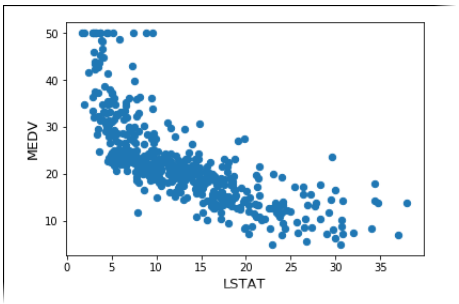
b) Select the cell that contains the code listing, then select **Run**.



6. Examine a scatter plot of `LSTAT` and `MEDV`.

a) Scroll down and select the code cell beneath the **Examine a scatter plot of `LSTAT` and `MEDV`** title, then select **Run**.

b) Examine the output.



- As you might expect, a house's value tends to decrease as the percentage of low-status households in the area increases.
- You can also see some of the outliers at the right end of the x-axis.

7. Plot a regression line for a given model.

a) Scroll down and view the cell titled **Plot a regression line for a given model**, and examine the code listing below it.

```
1  def plot_regression(X, y, model, is_svm):
2      plt.scatter(X, y, s = 40)
3      plt.xlabel("LSTAT", fontsize = 13)
4      plt.ylabel("MEDV", fontsize = 13)
5
6      predict = model.predict(X)
7
8      if is_svm == True:
9          # Calculate margins based on epsilon value.
10         high_margin = predict + model.epsilon
11         low_margin = predict - model.epsilon
12
13         # Sort arrays for plotting.
14         X_sort, predict_sort, high_margin_sort, low_margin_sort = \
15         zip(*sorted(zip(X, predict, high_margin, low_margin)))
16
17         plt.plot(X_sort, predict_sort, c = 'r', linestyle = '-')  # Plot regression line.
18         plt.plot(X_sort, high_margin_sort, c = 'r', linestyle = '--')  # Plot upper margin.
19         plt.plot(X_sort, low_margin_sort, c = 'r', linestyle = '--')  # Plot lower margin.
20
21         # Plot support vectors.
22         plt.scatter(X[model.support_], y[model.support_],
23                     s = 100, linewidth = 1, facecolors = 'none', edgecolors = 'k')
24
25     else:
26         X_sort, predict_sort = \
27         zip(*sorted(zip(X, predict)))
28
29         plt.plot(X_sort, predict_sort, c = 'r', linestyle = '-')
30
31     plt.show()
32
33 print('Function to plot the regression line has been defined.')
```

This function, when called, will plot a regression line on the scatter plot shown previously.

- The function requires the training data, label data, and model object as input.
- The `is_svm` argument will determine whether or not to plot the margins (for SVM models).
- Lines 2 through 4 create the scatter plot, same as before.
- Lines 8 through 23 plot the regression line, then plot the support-vector margins for SVM models.
- Lines 25 through 29 just plot the regression line for non-SVM models.
- Line 33 shows when the function has been defined.

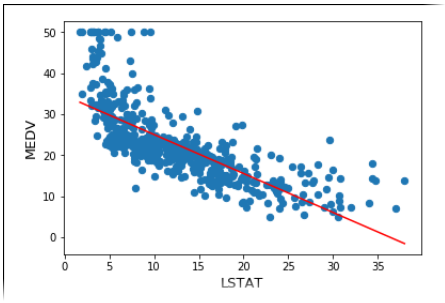
b) Select the cell that contains the code listing, then select **Run**.

```
Function to plot the regression line has been defined.
```

8. Train a basic linear regression model and plot its line of best fit.

a) Scroll down and select the code cell beneath the **Train a basic linear regression model and plot its line of best fit** title, then select **Run**.

b) Examine the output.



The regression line doesn't appear to account for the outliers all that well, which can lead to underfitting.

9. Train an SVM model and plot its regression line plus margins.

a) Scroll down and view the cell titled **Train an SVM model and plot its regression line plus margins**, and examine the code listing below it.

Train an SVM model and plot its regression line plus margins

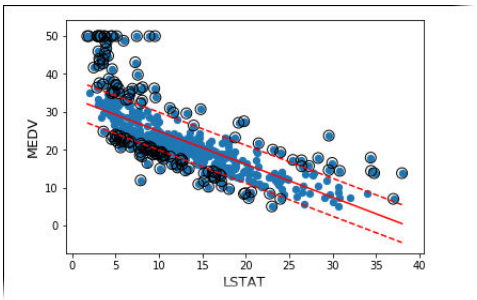
In [ ]:
```
from sklearn.svm import SVR

svm = SVR(kernel = 'linear', epsilon = 5)
svm.fit(X, y)

plot_regression(X, y, svm, True)
```

The `SVR()` class implements support-vector regression. In this case, there are two hyperparameters being set:

- `kernel` specifies the kernel method to use; in this case, you'll start with a linear kernel.
- `epsilon` (ε) determines the space between the margins. A higher value leads to wider margins. You're starting with a value of 5.

b) Select the cell that contains the code listing, then select **Run**.

c) Examine the output.



- The solid red line is the regression line.
- The dashed red lines are the support-vector margins.
- The circled data points are the support vectors.
- The regression line with SVMs appears to have shifted upward slightly, centering more on the overall spread of the dataset. This may help the model generalize better with the presence of outliers in the training.

10. How does SVM regression differ from SVM classification in terms of how the data examples are included or not included within the margins?

11. Adjust the margins using a different `epsilon` value.

a) Scroll down and view the cell titled **Adjust the margins using a different `epsilon` value**, and examine the code listing below it.

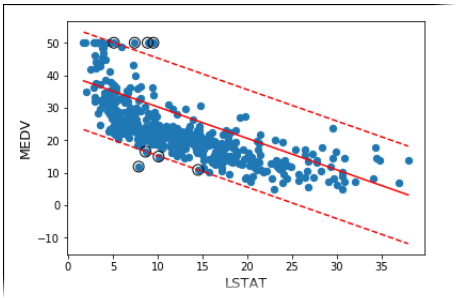
Adjust the margins using a different `epsilon` value

In [ ]:
```
svm = SVR(kernel = 'linear', epsilon = 15)
svm.fit(X, y)

plot_regression(X, y, svm, True)
```

While the previous SVM model seemed to do well in incorporating outliers as compared to linear regression, the margins may still be too narrow, leading to underfitting. So, you'll plot another model where the `epsilon` value is larger, leading to wider margins.

b) Select the cell that contains the code listing, then select **Run**.

c) Examine the output.



The margins are much wider now, and more data examples are placed within those margins. This is not necessarily better for this particular case, and may actually lead to overfitting. You'll search for the optimal `epsilon` value shortly, when you train on the full dataset.

12. Split the datasets.

a) Scroll down and view the cell titled **Split the datasets**, and examine the code listing below it.

Split the datasets

In [ ]:
```
from sklearn.model_selection import train_test_split

label_columns = ['target']

# Split the training and test datasets and their labels.
X_train, X_test, y_train, y_test = train_test_split(data_raw.loc[:, 'CRIM': 'LSTAT'],
                                                    data_raw[label_columns],
                                                    random_state = 2)


print('Training and test datasets and their labels have been split.')
```

This splits the datasets in the usual way, incorporating the entire dataset this time.

b) Select the cell that contains the code listing, then select **Run**.

```
Training and test datasets and their labels have been split.
```

13. Standardize the features.

a) Scroll down and view the cell titled **Standardize the features**, and examine the code listing below it.

Standardize the features

In [ ]:
```
def standardize(X):
    result = X.copy()

    for feature in X.columns:
        result[feature] = (X[feature] - X[feature].mean()) / X[feature].std()  # z-score formula.

    return result

X_train = standardize(X_train)
X_test = standardize(X_test)

print('The features have been standardized.')
```

Recall that you're scaling the features on this dataset so that the regularization penalty is applied equally.

b) Select the cell that contains the code listing, then select **Run**.

```
The features have been standardized.
```

14. Evaluate an SVM model using a holdout test set.

a) Scroll down and view the cell titled **Evaluate an SVM model using a holdout test set**, and examine the code listing below it.

Evaluate an SVM model using a holdout test set

In [ ]:
```
from sklearn.metrics import mean_squared_error as mse

svm = SVR(kernel = 'linear', epsilon = 5)
svm.fit(X_train, np.ravel(y_train))

predict = svm.predict(X_test)

# Calculate cost using the test data.
cost = mse(y_test, predict)

print('Cost (mean squared error): {:.2f}'.format(cost))
```

To start with, you'll just use the arbitrary value of 5 for `epsilon`.

b) Select the cell that contains the code listing, then select **Run**.

c) Examine the output.

```
Cost (mean squared error): 21.17
```

The mean squared error (MSE) for this model is 21.17. This is somewhat improved over your previous linear regression models, but there's still more opportunity to improve it further.

15. Optimize the SVM model with grid search and cross-validation.

a) Scroll down and view the cell titled **Optimize the SVM model with grid search and cross-validation**, and examine the code listing below it.

Optimize the SVM model with grid search and cross-validation

In [ ]:

```
1  from sklearn.model_selection import GridSearchCV
2
3  svm = SVR(gamma = 'auto')
4
5  grid = [{'kernel': ['linear', 'rbf', 'poly', 'sigmoid'],
6           'C': [0.01, 0.1, 1, 5, 10, 25, 50, 100],
7           'epsilon': [0.01, 0.1, 1, 5, 10, 25, 50, 100]}]
8
9  search = GridSearchCV(svm, param_grid = grid, scoring = 'neg_mean_squared_error', cv = 5, iid = False)
10 search.fit(X_train, np.ravel(y_train));
11
12 print(search.best_params_)
```

This code performs a grid search to determine the best hyperparameters for an SVM model.

- On line 3, the model will default to using a `gamma` of `auto`. Like with the `SVC()` class, gamma determines the coefficient to use with non-linear kernels (in this case, 1 divided by the number of features).
- On line 5, the grid search will identify the best kernel to use among the same four kernels as before.
- On line 6, the grid search will try these kernels against a list of several `C` values, ranging from 0.01 to 100. Unlike with classification, the `C` in SVM regression does not determine the margin width, but rather just applies a regularization penalty.
- On line 7, the grid search will try the same range of values for `epsilon`, which determines the width of the margins.
- On line 9, the grid search will be optimizing for MSE and will perform five-fold cross-validation on the training data.

Note: Grid search is being used for the sake of simplicity. You could also use randomized search to avoid hard-coding hyperparameter values.

b) Select the cell that contains the code listing, then select **Run**.

This will take a few moments to complete.

c) Examine the output.

```
{'C': 100, 'epsilon': 1, 'kernel': 'rbf'}
```

The grid search identified a `C` value of 100 and an `epsilon` value of 1, along with the RBF kernel, as the optimal hyperparameter combination.

d) Scroll down and select the next code listing cell.

In [ ]:

```
1  predict = search.predict(X_test)
2
3  cost = mse(y_test, predict)
4
5  print('Cost (mean squared error): {:.2f}'.format(cost))
```

e) Select **Run**.

f) Examine the output and confirm that the model's error decreased.

```
Cost (mean squared error): 8.15
```

16. Shut down the notebook and close the lab.

a) From the menu, select **Kernel→Shutdown**.

b) In the **Shutdown kernel?** dialog box, select **Shutdown**.

c) Close the lab browser tab and continue on with the course.

MODULE 3

Build Multi-Layer Perceptrons (MLP)

The following lab is for Module 3: Build Multi-Layer Perceptrons (MLP).

LAB 4-5
Building an MLP

Data Files

~/Neural Networks/NeuralNetworks-Occupancy.ipynb

~/Neural Networks/VisualizeNN.py

~/Neural Networks/occupancy_data/test.csv

~/Neural Networks/occupancy_data/train.csv

Scenario

You work for IOT Company, which sells building automation systems. Sensors are placed all throughout a building that measure various physical attributes of the immediate room. In order to make the system smarter, you want it to be able to detect the presence of people in a room, and react accordingly. For example, if there is someone in the room, the system might turn up the heat to more comfortable levels; and when no one is in the room, the system turns down the heat to conserve energy. Or, the system might have a virtual assistant that will offer its services when the room is occupied, and then turn off when it is not.

You've been given a labeled dataset that has sensor measurements for an office room, taken every minute for several weeks. The room is classified as either occupied or not occupied. Since there are thousands of data examples—one for each minute of observation—you feel you have a large enough dataset to make use of a neural network. So, you'll create a classification model using a multi-layer perceptron (MLP).

---

1. Open the lab and notebook.
   a) From the Coursera lab page, select **Open Lab** to open a Jupyter Notebook session in another web browser tab.
   b) Select **Neural Networks/NeuralNetworks-Occupancy.ipynb** to open it.

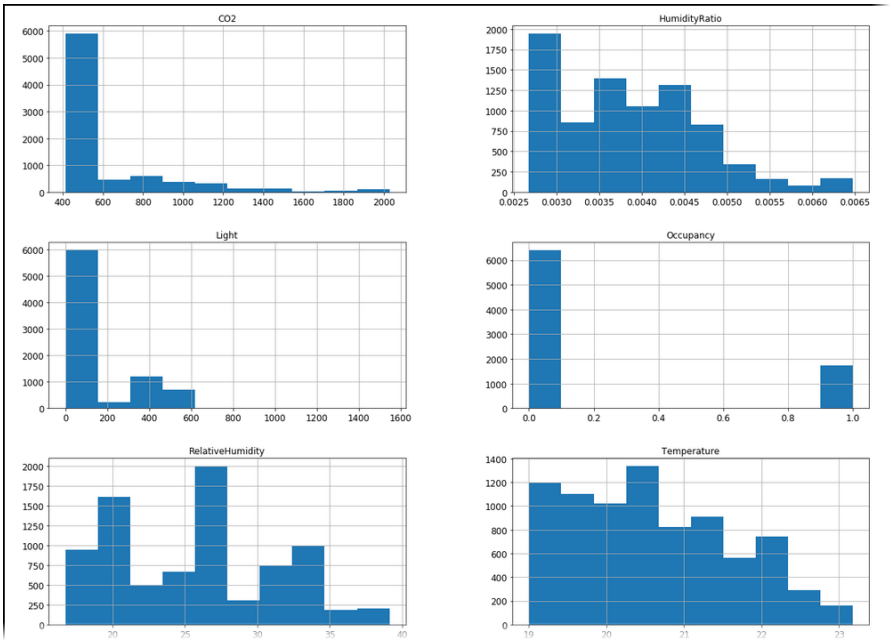
2. Load the dataset.
   a) Select the code cell beneath the **Import software libraries and load the dataset** title, then select **Run**.
   b) Verify that **train.csv** and **test.csv** are in the project folder, and that they were loaded with 8,143 and 2,665 records, respectively. This dataset already comes pre-split for training and testing.

3. Get acquainted with the dataset.
   a) Scroll down and select the cell beneath the **Get acquainted with the dataset** title, then select **Run**.
   b) Examine the output.
      - The training set includes 8,143 rows and 7 columns.
      - Most of the columns contain float values and are measurements of some physical aspect of the room.
      - The columns that don't contain floats are `Date` (string objects) and `Occupancy` (integers). The `Occupancy` column is the label that specifies if a room is not occupied (0) or if it is occupied (1).
      - Each row is a point in time specified by the `Date` column. The room's measurements were taken in intervals of one minute. Because this is a string object, you'll need to find some way to handle these time values. You could drop the `Date` column entirely, but it's highly likely that time has an effect on whether or not a room is occupied.
      - There is no missing data; all rows have values for every column.

4. Examine the distribution of various features.

a) Scroll down and select the code cell beneath the **Examine the distribution of various features** title, then select **Run**.

b) Examine the output.



- Several features are right-skewed, especially `CO2` and `Light`.
- The distribution for `RelativeHumidity` seems to have alternating peaks.

5. Examine a general summary of statistics.

a) Scroll down and select the code cell beneath the **Examine a general summary of statistics** title, then select **Run**.

b) Examine the output.

```
       Temperature  RelativeHumidity    Light      CO2  HumidityRatio  \
count     8143.000          8143.000 8143.000 8143.000       8143.000
mean        20.619            25.732  119.519  606.546          0.004
std          1.017             5.531  194.756  314.321          0.001
min         19.000            16.745    0.000  412.750          0.003
25%         19.700            20.200    0.000  439.000          0.003
50%         20.390            26.223    0.000  453.500          0.004
75%         21.390            30.533  256.375  638.833          0.004
max         23.180            39.117 1546.333 2028.500          0.006

       Occupancy
count   8143.000
mean       0.212
std        0.409
min        0.000
25%        0.000
50%        0.000
75%        0.000
max        1.000
```

- This dataset has uneven scaling.
- `Light` and `CO2` have comparatively high values.
- `Temperature` and `RelativeHumidity` have comparatively low values.
- `HumidityRatio` has an especially tiny scale; its max value is ~0.006.
- MLPs are highly sensitive to scaling, so you'll need to transform these features before training the model.

6. Split the label from the datasets.

a) Scroll down and select the code cell beneath the **Split the label from the datasets**, then select **Run**.
b) Verify that the labels for both training and test sets were split from the rest of the data.

7. Convert the `Date` column to datetime format for processing.

a) Scroll down and view the cell titled **Convert the `Date` column to datetime format for processing**, and examine the code listing below it.

Convert the `Date` column to datetime format for processing

```
In [ ]:
1 X_train['Date'] = pd.to_datetime(X_train['Date'])
2 X_test['Date'] = pd.to_datetime(X_test['Date'])
3
4 X_train.head()
```

The pandas `to_datetime()` function converts a loosely defined object with date and time values into a rigidly defined datetime format. It's quite smart at parsing the many different ways to represent date and time.

b) Select the cell that contains the code listing, then select **Run**.
c) Examine the output.

	Date	Temperature	RelativeHumidity	Light	CO2	HumidityRatio
0	2015-02-05 19:37:00	21.200	19.840	0.0	525.333333	0.003082
1	2015-02-04 22:11:00	21.390	25.700	0.0	475.000000	0.004046
2	2015-02-09 13:51:00	21.245	32.925	474.5	1126.500000	0.005146
3	2015-02-05 20:36:00	21.200	19.390	0.0	472.500000	0.003012
4	2015-02-10 02:39:00	20.290	32.900	0.0	460.000000	0.004846

- The datetime format makes it much easier to work with date and time values.
- Your next step is to extract each relevant date and time component (year, month, day, hour, etc.) and place it in its own feature column. That way, each component will have some significance on the output.
- However, some date and time components may stay the same for all examples. If every measurement was taken in the same year, there's not much point in making it a feature.

8. Determine which datetime components have unique values.

a) Scroll down and select the code cell beneath the **Determine which datetime components have unique values** title, then select **Run**.
b) Examine the output.

```
Unique years:   [2015]
Unique months:  [2]
Unique days:    [ 5  4  9 10  6  8  7]
Unique hours:   [19 22 13 20  2 21  7  6  9 18 16 15  4  0 23 17  3  8  5 12 11 10  1 14]
Unique minutes: [37 11 51 36 39 20 17 44  6 54 49 52 14 56 46 34  8 10 38 45 30 58  7 13
 12 42 31 23 28 16 33 43 53 47  9  1 21  4  5  0 59 22 50  3 26 24 29  2
 19 25 48 35 57 32 15 27 18 40 55 41]
Unique seconds: [0]
```

- Since there is only one unique value for year and month, that means all of the measurements were taken in the same month of the same year. You won't use these as features.
- Days, hours, and minutes have multiple unique values. You'll use these as features.
- The only unique value for seconds is 0. Either each measurement was taken precisely on the minute, or the time measurement wasn't precise enough to capture seconds. In either case, there's no need to use this as a feature.

9. Perform common preparation on the training and test sets.

a) Scroll down and view the cell titled **Perform common preparation on the training and test sets**, and examine the code listing below it.

Perform common preparation on the training and test sets

In [ ]:

```
1  # Perform common cleaning and feature engineering tasks on datasets.
2  def prep_dataset(X):
3
4      # FEATURE ENGINEERING
5
6      # Extract days, hours, and minutes from timestamp.
7      day = X['Date'].dt.day
8      X['Day'] = day.astype('float64')
9
10     hour = X['Date'].dt.hour
11     X['Hour'] = hour.astype('float64')
12
13     minute = X['Date'].dt.minute
14     X['Minute'] = minute.astype('float64')
15
16     return X
17
18 X_train = prep_dataset(X_train.copy())
19
20 X_test = prep_dataset(X_test.copy())
21
22 X_train.head()
```

- On lines 7 through 14, a new column is being created for days, hours, and minutes.
- Each time component has a method that enables this extraction, like `dt.day()` to extract the day from a full datetime object.

b) Select the cell that contains the code listing, then select **Run**.

c) Examine the output.

	Date	Temperature	RelativeHumidity	Light	CO2	HumidityRatio	Day	Hour	Minute
0	2015-02-05 19:37:00	21.200	19.840	0.0	525.333333	0.003082	5.0	19.0	37.0
1	2015-02-04 22:11:00	21.390	25.700	0.0	475.000000	0.004046	4.0	22.0	11.0
2	2015-02-09 13:51:00	21.245	32.925	474.5	1126.500000	0.005146	9.0	13.0	51.0
3	2015-02-05 20:36:00	21.200	19.390	0.0	472.500000	0.003012	5.0	20.0	36.0
4	2015-02-10 02:39:00	20.290	32.900	0.0	460.000000	0.004846	10.0	2.0	39.0

Now that you've extracted the relevant date and time components, you can drop the `Date` column.

10. Drop columns that won't be used for training.

a) Scroll down and select the code cell beneath the **Drop columns that won't be used for training** title, then select **Run**.

b) Examine the output and verify that the `Date` column was dropped.

11. Standardize the features.

a) Scroll down and view the cell titled **Standardize the features**, and examine the code listing below it.

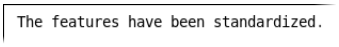
Standardize the features

In [ ]:

```
def standardize(X):
    result = X.copy()

    for feature in X.columns:
        result[feature] = (X[feature] - X[feature].mean()) / X[feature].std()  # z-score formula.

    return result

X_train = standardize(X_train)

X_test = standardize(X_test)

print('The features have been standardized.')
```

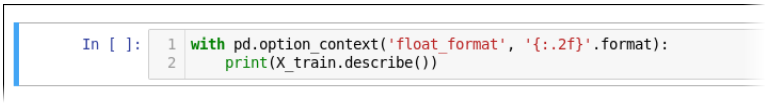
As you've seen before, this function applies the z-score formula to the features for scaling purposes.

b) Select the cell that contains the code listing, then select **Run**.



```
The features have been standardized.
```

c) Scroll down and select the next code listing cell.



In [ ]:

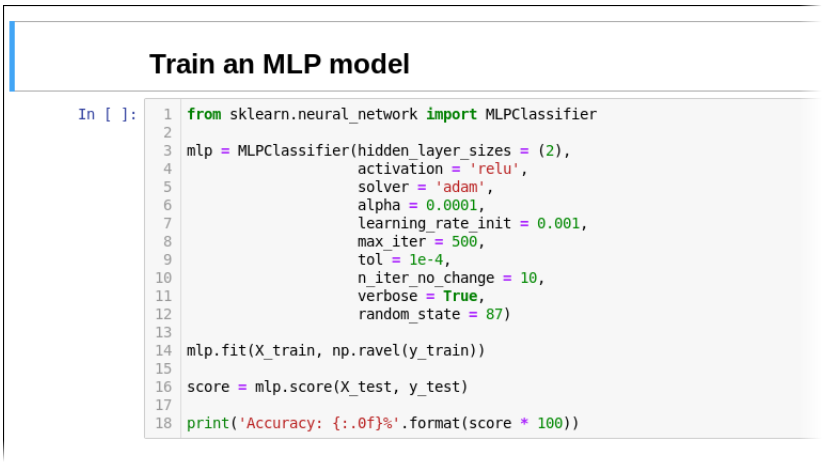
```
with pd.option_context('float_format', '{:.2f}'.format):
    print(X_train.describe())
```

d) Select **Run**.

e) Examine the output and verify that the features have been scaled.

12. Train an MLP model.

a) Scroll down and view the cell titled **Train an MLP model**, and examine the code listing below it.



Train an MLP model

In [ ]:

```
from sklearn.neural_network import MLPClassifier

mlp = MLPClassifier(hidden_layer_sizes = (2),
                    activation = 'relu',
                    solver = 'adam',
                    alpha = 0.0001,
                    learning_rate_init = 0.001,
                    max_iter = 500,
                    tol = 1e-4,
                    n_iter_no_change = 10,
                    verbose = True,
                    random_state = 87)

mlp.fit(X_train, np.ravel(y_train))

score = mlp.score(X_test, y_test)

print('Accuracy: {:.0f}%'.format(score * 100))
```

As its name implies, the `MLPClassifier()` class in scikit-learn trains an MLP neural network for classification purposes. It has many hyperparameters/arguments. The ones being specified here include:

- `hidden_layer_sizes` specifies the number of hidden layers to include in the network, as well as the number of neurons in each hidden layer. The argument takes a tuple, where the first value is the number of neurons in the first hidden layer, the second value is the number of neurons in the second hidden layer, and so on. So, if you wanted three hidden layers where each layer has ten neurons, you'd input `(10, 10, 10)` as the tuple. Here, because there is only one value, there will only be one hidden layer, and it will include two neurons.
- `activation` specifies the activation function to use. ReLU is being used here.
- `solver` specifies the method used to minimize cost and optimize the connection weights. The `adam` solver is similar to stochastic gradient descent (SGD).
- `alpha` is the ℓ_2 regularization penalty to apply. Here, the model is using the default value.
- `learning_rate_init` is the initial learning rate to use in gradient descent solvers.
- `max_iter` is the maximum number of iterations that the solver will perform if it doesn't converge first.
- `tol` defines a tolerance threshold for the solver to exceed when minimizing cost. If the cost is not minimized by more than `tol` for a specified number of iterations, then the solver will stop. The value here is the default.
- `no_iter_change` is the number of iterations for which the solver can fail to exceed `tol` before it stops.
- `verbose`, when set to `True`, will print out the loss at each iteration.

Note: You're only creating one hidden layer to minimize the time spent on the grid search later. Multiple layers will slow the search down considerably.

b) Select the cell that contains the code listing, then select **Run**.

c) Examine the output.

```
Iteration 142, loss = 0.05191888
Iteration 143, loss = 0.05184846
Iteration 144, loss = 0.05175415
Iteration 145, loss = 0.05176166
Iteration 146, loss = 0.05165864
Iteration 147, loss = 0.05161039
Iteration 148, loss = 0.05154295
Iteration 149, loss = 0.05148317
Training loss did not improve more than tol=0.000100 for 10 consecutive epochs. Stopping.
Accuracy: 88%
```

- The loss at each iteration is output.
- Each iteration minimizes the loss more than the previous iteration.
- Iteration 149 was the 10th iteration for which the loss did not improve by more than `tol`, so the solver stopped there.
- You'll plot this loss minimization in the next step to get a better look.
- The accuracy of this model is 88%. As with any other classifier, you can use many different evaluation metrics. In this case, you'll just try to optimize accuracy.

13. Visualize the loss minimization through gradient descent.

a) Scroll down and view the cell titled **Visualize the loss minimization through gradient descent**, and examine the code listing below it.

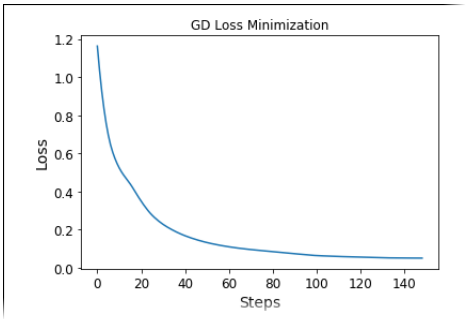
Visualize the loss minimization through gradient descent

In [ ]:
```
def plot_loss(model):
    plt.plot(model.loss_curve_)
    plt.title('GD Loss Minimization')
    plt.xlabel('Steps')
    plt.ylabel('Loss')

plot_loss(mlp)
```

The `loss_curve_` attribute conveniently returns an array of the loss value at each iteration.

b) Select the cell that contains the code listing, then select **Run**.

c) Examine the output.



The loss decreases dramatically for the first 20 or so iterations, but after that, the change is minor. If time were more of a factor, you might want to increase the value of `tol` so that the solver isn't wasting time on more iterations for little gain.

14. Visualize the neural network architecture.

a) Scroll down and view the cell titled **Visualize the neural network architecture**, and examine the code listing below it.

Visualize the neural network architecture

In [ ]:

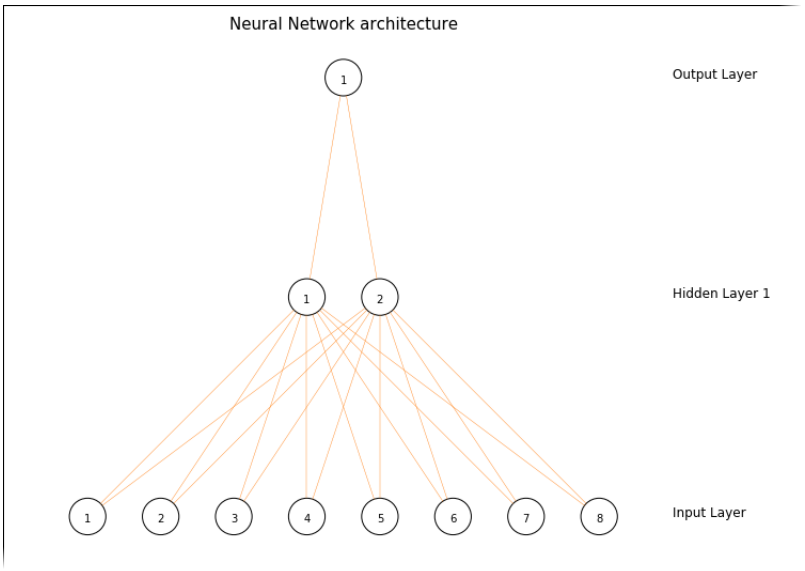
```
1  def nn_diagram(X, y, model, show_weights):
2
3      # Create structure of network from dataset shapes and hidden layer sizes.
4      nn_struct = np.hstack(([X.shape[1]], np.asarray(model.hidden_layer_sizes), [y.shape[1]]))
5
6      # Only plot weights if specified.
7      if show_weights == True:
8          network = VisNN.DrawNN(nn_struct, model.coefs_)
9      else:
10         network = VisNN.DrawNN(nn_struct)
11
12     network.draw()
13
14 nn_diagram(X_train, y_train, mlp, False)
```

This function uses the `VisualizeNN.py` module to draw a visual representation of the neural network.

- Line 4 creates a structure object using the shape of the training data and the labels, along with the size of the hidden layers. This object will be passed in to the class that creates the diagram.
- Lines 7 through 10 determine whether to draw the diagram with or without weights, according to the user's preference.
- Line 14 calls the function without weights.

b) Select the cell that contains the code listing, then select **Run**.

c) Examine the output.



The network architecture, as expected, is broken down into input, hidden, and output layers.

- The input layer includes eight neurons, each of which maps to a feature in the dataset.
- The hidden layer has two neurons. This is the number of neurons you arbitrarily specified when creating the `MLPClassifier()` object.
- The output layer has one neuron—the label classification of a data example (0 or 1).
- Each neuron in one layer is connected to all neurons in the next layer.

15. Retrieve the neuron weights and bias terms and redraw the network architecture.

a) Scroll down and select the code cell beneath the **Retrieve the neuron weights and bias terms and redraw the network architecture** title, then select **Run**.

b) Examine the output.

```
Weights between input layer and hidden layer:
[[-0.3438876   0.254666  ]
 [ 0.07710875 -0.35894019]
 [ 1.27571406 -1.05065641]
 [ 0.62432824 -1.29802377]
 [-0.42358403 -0.35945967]
 [ 0.09719079  0.53563612]
 [-0.1719664   0.10169442]
 [-0.09943472 -0.00328182]]

Weights between hidden layer and output layer:
[[ 2.06152438]
 [-2.24583236]]

Bias terms between input layer and hidden layer:
[0.48037193 1.30762661]

Bias terms between hidden layer and output layer:
[-2.74116309]
```

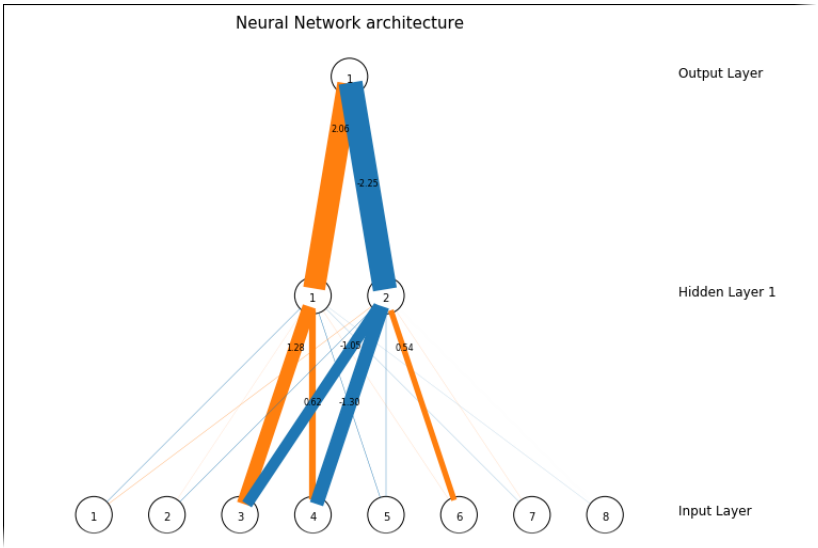
- The first array lists the weights of each neuron in the connections between the input layer and the hidden layer.
- The second array does likewise for the connections between the hidden and output layer.
- The third array lists the bias terms for the connections between the input layer and the hidden layer.
- The fourth array does likewise for the connection between the hidden and output layer.

c) Scroll down and select the next code listing cell.

```
In [ ]: 1 nn_diagram(X_train, y_train, mlp, True)
```

d) Select **Run**.

e) Examine the output.



This time, the network diagram outputs with the neuron weights.

- The thicker the line is, the stronger the weight.
- Orange lines indicate positive weights.
- Blue lines indicate negative weights.
- The actual weight value is displayed for any weight that is above 0.5 or below −0.5 (i.e., the weights of most significance).

16. **How does backpropagation generate the weights between the neurons of different layers in an MLP neural network?**

17. **What can you tell about the weights of this particular network structure?**

18. Fit an MLP model using grid search with cross-validation.

a) Scroll down and view the cell titled **Fit an MLP model using grid search with cross-validation**, and examine the code listing below it.

Fit an MLP model using grid search with cross-validation

```
1  from sklearn.model_selection import GridSearchCV
2
3  mlp = MLPClassifier(alpha = 0.0001,
4                      learning_rate_init = 0.001,
5                      max_iter = 500,
6                      tol = 1e-4,
7                      n_iter_no_change = 10,
8                      random_state = 87)
9
10 grid = {'hidden_layer_sizes': [(5), (6)],
11         'activation': ['logistic', 'tanh', 'relu'],
12         'solver': ['sgd', 'adam']}
13
14 search = GridSearchCV(mlp, param_grid = grid, scoring = 'accuracy', cv = 5, iid = False)
15
16 start = time()
17 search.fit(X_train, np.ravel(y_train))
18 end = time()
19 train_time = (end - start)
20
21 print('Grid search took {:.2f} seconds to find an optimal fit.'.format(train_time))
22 print(search.best_params_)
```

This code performs a grid search to determine optimal hyperparameters for the MLP model.

- On lines 3 through 8, the algorithm will use several of the same numeric values for the hyperparameters that were used before.
- On line 10, the grid begins by alternating between having one hidden layer with five neurons, and one hidden layer with six neurons.
- On line 11, each of the major activation functions is tried: the logistic (sigmoid) function, the tanh function, and the ReLU function.
- On line 12, two SGD-like weight optimization techniques are tried. These tend to be most useful in large datasets with thousands of data examples.
- To save time, the search field is relatively sparse. In a real-world situation, where time is less of a factor, you'd include many more possible combinations of hyperparameters. Also, grid search is being used here so that the outcome is more deterministic, but in a real-world situation, you'd use randomized search.
- On line 14, the grid search will be optimizing for accuracy and will perform five-fold cross-validation on the training data.

b) Select the cell that contains the code listing, then select **Run**.

Note: It may take up to 10 minutes for the search to complete.

c) Examine the output.

```
Grid search took 124.61 seconds to find an optimal fit.
{'activation': 'logistic', 'hidden_layer_sizes': 6, 'solver': 'sgd'}
```

- The optimal activation function is the logistic (sigmoid) function.
- The optimal number of neurons in the hidden layer is six.
- The optimal weight optimization technique is SGD.

d) Scroll down and select the next code listing cell.

```
In [ ]:
1 score = search.score(X_test, y_test)
2
3 print('Accuracy: {:.0f}%'.format(score * 100))
```

e) Select **Run**.

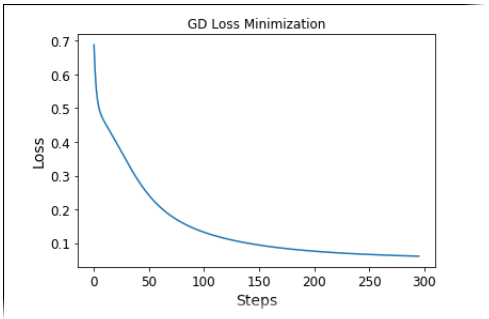
f) Examine the output and confirm that the model's accuracy increased.

```
Accuracy: 94%
```

19. Visualize the loss minimization of the optimized model.

a) Scroll down and select the code cell beneath the **Visualize the loss minimization of the optimized model** title, then select **Run**.

b) Examine the output.

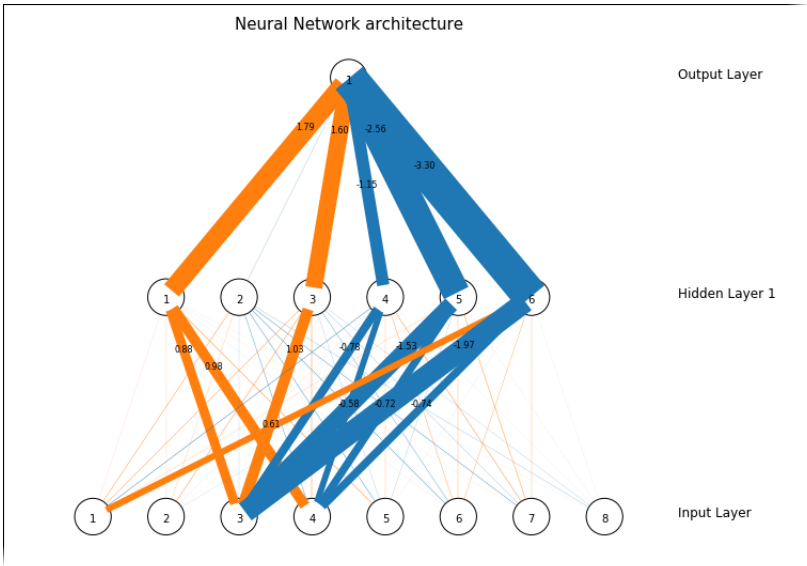


It takes about 300 steps for the solver to converge before failing to minimize more than the tolerance threshold.

20. Visualize the network structure of the optimized model.

a) Scroll down and select the code cell beneath the **Visualize the network structure of the optimized model** title, then select **Run**.

b) Examine the output.



- The input layer and output layer have the same number of neurons, but the hidden layer now has six.
- The weights between neurons have changed. While some of the weights are relatively weak, there are also plenty of stronger weights.
- Features 3 and 4 (`Light` and `CO2`) seem to have strong positive weight with hidden neurons 1 and 3, while also having strong negative weight with hidden neurons 4, 5, and 6.
- Feature 1 (`Temperature`) seems to have a relatively strong positive weight with hidden neuron 6.
- Hidden neurons 1 and 3 have strong positive weight with the output neuron.
- Hidden neurons 4, 5, and 6 have strong negative weight with the output neuron.
- Hidden neuron 2 seems to not have strong weight with any layer, input or output.

21. Examine the model's predictions on the test set.

a) Scroll down and select the code cell beneath the **Examine the model's predictions on the test set** title, then select **Run**.

b) Examine the output.

	Date	Temperature	RelativeHumidity	Light	CO2	HumidityRatio	ActualOccupancy	PredictedOccupancy
0	2/2/2015 23:49	20.650000	22.245000	0.000000	443.000000	0.003342	0	0
1	2/2/2015 21:10	20.890000	23.000000	0.000000	491.666667	0.003508	0	0
2	2/3/2015 19:56	21.245000	27.745000	0.000000	770.750000	0.004331	0	0
3	2/3/2015 5:51	20.290000	22.650000	0.000000	431.000000	0.003328	0	0
4	2/3/2015 2:12	20.525000	22.267500	0.000000	442.750000	0.003320	0	0
5	2/4/2015 8:50	21.200000	25.180000	454.000000	740.200000	0.003917	1	0
6	2/2/2015 17:19	22.500000	24.865000	433.000000	816.500000	0.004189	1	1
7	2/4/2015 9:56	23.200000	25.500000	722.000000	1011.400000	0.004485	1	1
8	2/4/2015 8:44	21.083333	25.200000	453.000000	719.500000	0.003892	1	0

22. Shut down the notebook and close the lab.

a) From the menu, select **Kernel→Shutdown**.

b) In the **Shutdown kernel?** dialog box, select **Shutdown**.

c) Close the lab browser tab and continue on with the course.

MODULE 4

Build Convolutional and Recurrent Neural Networks (CNN/RNN)

The following labs are for Module 4: Build Convolutional and Recurrent Neural Networks (CNN/RNN).

LAB 4-6
Building a CNN

Data File

~/Neural Networks/NeuralNetworks-Fashion.ipynb

Scenario

You work for an online retailer that sells various articles of clothing from many different brands. Each brand provides you with different suggestions for how to categorize each article to facilitate user searches. Rather than use the brands' suggestions, which often conflict with one another or aren't particularly useful, the storefront uses a categorization scheme that was developed in house. Currently, each new article must be manually categorized for searching by several employees. An employee visually examines a product and determine whether or not is a shoe, a shirt, a hat, etc. This is tedious work that can be automated using computer vision.

Thankfully, you have a rather large database of existing product images that have already been categorized. So, you'll use this dataset to train a convolutional neural network (CNN) to classify new product images. Once the model has achieved enough success, you'll be able to push it to production.

Note: Due to the stochastic nature of TensorFlow algorithms, there will be some randomness with your results. If you wish to make the results more deterministic, uncomment the relevant lines in the first code block.

---

1. Open the lab and notebook.
   a) From the Coursera lab page, select **Open Lab** to open a Jupyter Notebook session in another web browser tab.
   b) Select **Neural Networks/NeuralNetworks-Fashion.ipynb** to open it.

2. Load the dataset.
   a) Select the code cell beneath the **Import software libraries and load the dataset** title, then select **Run**.

```
Using TensorFlow backend.

Libraries used in this project:
- Python 3.7.3 (default, Mar 27 2019, 22:11:17)
[GCC 7.3.0]
- NumPy 1.16.2
- Matplotlib 3.0.3
- scikit-learn 0.20.3
- TensorFlow 2.0.0
- Keras 2.3.1

Loaded 60000 training records.
Loaded 10000 test records.
```

   b) Verify that the training and test sets were loaded with 60,000 and 10,000 records, respectively.

   This dataset—called Fashion-MNIST—includes numerical values that can be used to construct small grayscale images of different types of clothing. The dataset was loaded using Keras, a frontend to the TensorFlow deep learning library. You'll use Keras to build and train a CNN.

Note: Fashion-MNIST is another take on the MNIST dataset, an image dataset of handwritten numbers that is very commonly used to teach machine learning concepts. To learn more about Fashion-MNIST, follow this link: **https://github.com/zalandoresearch/fashion-mnist**.

3. Get acquainted with the dataset.

   a) Scroll down and select the code cell beneath the **Get acquainted with the dataset** title, then select **Run**.

   b) Examine the output.

```
Shape of feature space: (28, 28)

A few examples:

[[[  0   0   0 ...   0   0   0]
  [  0   0   0 ...   0   0   0]
  [  0   0   0 ...   0   0   0]
  ...
  [  0   0   0 ... 180   0   0]
  [  0   0   0 ...  72   0   0]
  [  0   0   0 ...  70   0   0]]

 [[  0   0   0 ...   0   0   0]
  [  0   0   0 ...   0   0   0]
  [  0   0   0 ...  39   1   0]
  ...
  [  0   0   0 ... 238   0   0]
  [  0   0   0 ... 131   0   0]
  [  0   0   0 ...   0   0   0]]

 [[  0   0   0 ...   0   0   0]
  [  0   0   0 ...   0   0   0]
  [  0   0   0 ...   7   0   0]
  ...
  [  0   0   0 ...   0   9   0]
  [  0   0   0 ...   0   3   0]
  [  0   0   0 ...   0   0   0]]]
```

   - The shape of the feature space shows that the training set is multi-dimensional; rather than an image having 28 features, it has 28 × 28 features. This corresponds to the dimensions of the image—it is 28 pixels wide and 28 pixels high.
   - Example images 7, 8, and 9 have their features printed. The features are truncated, but you can see that most of their values are 0, though some have actual positive values. Each number represents that particular pixel's intensity in grayscale. In other words, a 0 is completely black, whereas 255 is completely white.

   c) Scroll down and select the next code listing cell.

```
In [ ]:
class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
               'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']

for i in range(10):
    print('{} ({})'.format(class_names[i], np.unique(y_train)[i]))
```

   d) Select **Run**.

   e) Examine the output.

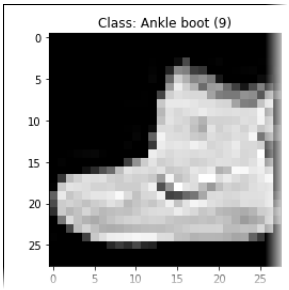
```
T-shirt/top (0)
Trouser (1)
Pullover (2)
Dress (3)
Coat (4)
Sandal (5)
Shirt (6)
Sneaker (7)
Bag (8)
Ankle boot (9)
```

   Each class label is mapped to its actual class name (i.e., the type of clothing). There are 10 total classes.

4. Visualize the data examples.

a) Scroll down and select the code cell beneath the **Visualize the data examples** title, then select **Run**.

b) Examine the output.



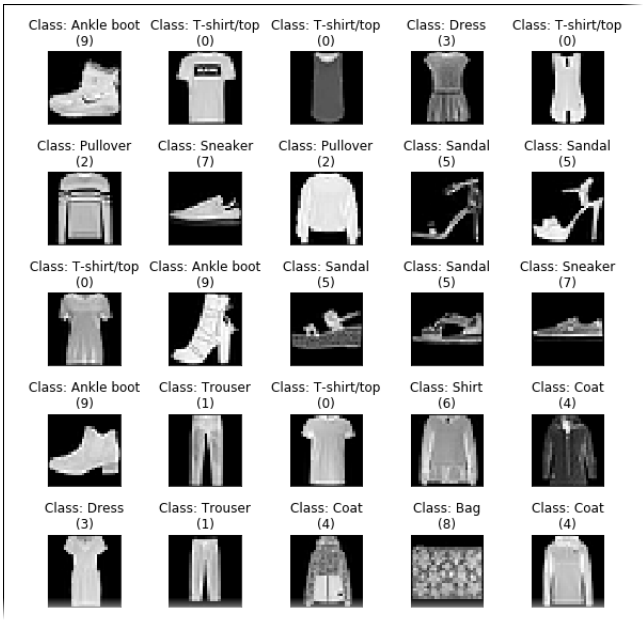
Using the image's grayscale features, Matplotlib was able to plot what the first image in the training set actually looks like. This image is classified as an ankle boot (9).

c) Scroll down and select the next code listing cell.

```
In [ ]:
fig, axes = plt.subplots(nrows = 5, ncols = 5, figsize = (8, 8))

for i, ax in zip(range(25), axes.flatten()):
    ax.imshow(X_train[i,:,:], cmap = 'gray')  # Plot training example.
    ax.title.set_text('Class: {}\n({})'.format(class_names[y_train[i]], y_train[i]))

# Turn off axis ticks for readability.
for ax in axes.flatten():
    ax.set_xticks([])
    ax.set_yticks([])

fig.tight_layout()
```

d) Select **Run**.

e) Examine the output.



The first 25 images from the training set are plotted in a grid.

5. Prepare the data for training with Keras.

a) Scroll down and view the cell titled **Prepare the data for training with Keras**, and examine the code listing below it.

Prepare the data for training with Keras

In [ ]:

```
1  # Reshape arrays to add greyscale flag.
2  X_train = X_train.reshape(-1, 28, 28, 1)
3  X_test = X_test.reshape(-1, 28, 28, 1)
4
5  from keras.utils import to_categorical
6
7  # One-hot encode the data for each label.
8  y_train = to_categorical(y_train)
9  y_test = to_categorical(y_test)
10
11 print('One-hot encoding for first image: {}'.format(y_train[0]))
```

Because the data is rather simple and uniform, not much data preparation needs to be done. However, in order for the CNN to predict a classification, the label needs to be one-hot encoded.

- Lines 2 and 3 reshape the data to a format that is supported by Keras. The first argument (`-1`) tells the function to reshape the dataset according to its total length (number of examples), which you want to preserve. You also need to preserve the 28 × 28 feature space in the next two arguments. The last argument (`1`) indicates to Keras that these images are in grayscale.
- On lines 8 and 9, the `to_categorical()` method is an easy way to one-hot encode values using the Keras library.

b) Select the cell that contains the code listing, then select **Run**.

c) Examine the output.

```
One-hot encoding for first image: [0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
```

Each example image is now one-hot encoded, where there is only one "activated" value (1) for an image, depending on its class. In this case, the first image has a 1 in the last column, indicating that it is labeled as class 9 (ankle boot). You can scroll up to the images you displayed earlier to verify that this is correct.

6. Split the datasets.

a) Scroll down and select the code cell beneath the **Split the datasets** title, then select **Run**.

b) Examine the output.

```
Training features:         (45000, 28, 28, 1)
Validation features:       (15000, 28, 28, 1)
Training labels:           (45000, 10)
Validation labels:         (15000, 10)
```

You're splitting the training dataset in order to have a validation holdout. Although the test set you loaded at the start is labeled, you'll treat it as the ultimate test case.

7. Build the CNN structure.

a) Scroll down and view the cell titled **Build the CNN structure**, and examine the code listing below it.

Build the CNN structure

In [ ]:

```
1  from keras.models import Sequential
2  from keras.layers import Conv2D, MaxPooling2D, Flatten, Dense
3  from keras.layers.advanced_activations import LeakyReLU
4
5  cnn = Sequential()
6
7  # Start stacking layers one-by-one.
8  cnn.add(Conv2D(filters = 32,
9                 kernel_size = (3, 3),  # First convolutional layer (32 output filters, 3x3 filter size)
10                input_shape = (28, 28, 1),
11                padding = 'same',
12                activation = 'linear'))  # Will add leaky ReLU layer next.
13 cnn.add(LeakyReLU(alpha = 0.1))
14 cnn.add(MaxPooling2D((2, 2), padding = 'same'))  # First pooling layer with 2x2 size.
15
16 cnn.add(Conv2D(64, (3, 3), padding = 'same', activation = 'linear'))
17 cnn.add(LeakyReLU(alpha = 0.1))
18 cnn.add(MaxPooling2D((2, 2), padding = 'same'))
19
20 cnn.add(Conv2D(128, (3, 3), padding = 'same', activation = 'linear'))
21 cnn.add(LeakyReLU(alpha = 0.1))
22 cnn.add(MaxPooling2D((2, 2), padding = 'same'))
23
24 cnn.add(Flatten())  # Connect convolution and dense layer.
25 cnn.add(Dense(10, activation = 'softmax'))  # Dense output layer with softmax activation.
26
27 print('The CNN structure has been built.')
```

This code builds the actual structure of the CNN with Keras.

- The `Sequential()` class indicates that you'll build the structure as a sequence of layers, which is an easy way to go about building a relatively simple network.
- Lines 8 through 12 add the first layer in the stack—the layer closest to the input. The `Conv2D()` object builds a convolutional layer. This particular layer has the following hyperparameters/ arguments:
  - `filters` specifies the number of output filters in the convolutional layer. In this case, there will be 32 filters. This number is somewhat arbitrary, as there is not necessarily a "best" number of filters to choose for any layer. However, the more filters there are, the longer it will take to train the network.
  - `kernel_size` specifies the dimensions of the filter itself. In this case, it will be 3 × 3 pixels.
  - `input_shape`, as the name suggests, is the shape of the input features the network will be training on. The last number indicates grayscale.
  - `padding` determines the padding to use, if any. A padding of `same` means that the layer will be padded in such a way that the output of the layer has the same dimensions as the layer's input.
  - `activation` is the activation function to use at the layer. Right now, this is a simple `linear` function. You could specify `relu` here, but this just uses the standard ReLU function that is susceptible to the vanishing gradients problem. You need to actually specify more advanced activation functions as their own layer. So, `linear` is acting as a placeholder until then.
- Line 13 adds an advanced activation function layer—in this case, leaky ReLU. The `alpha` argument is the slope coefficient.
- Line 14 adds a pooling layer after the convolution. The first argument defines the pooling size—in other words, the factor by which the image will be downscaled. So, a pooling layer of `(2, 2)` means that the image will be downscaled to half its initial size across both width and height. Also, padding is used.
- Lines 16 through 22 repeat this process, adding two more groups of convolutional and pooling layers. The only change is that the size of the convolution's output filter is being increased.
- Line 24 adds a "flattening" layer in order to reduce the dimensionality of the preceding output to just one. This is necessary in order to feed the multi-dimensional output of a convolution into a one-dimensional vector that is supported by the next fully connected layer.
- Line 25 adds the final layer, the fully connected (dense) layer that is similar to a traditional MLP. This layer uses the softmax activation function to generate a multi-class classification decision from the flattened input. The first argument specifies the number of possible outputs (class labels).

b) Select the cell that contains the code listing, then select **Run**.

```
The CNN structure has been built.
```

8. Compile the model and examine the layers.

a) Scroll down and view the cell titled **Compile the model and examine the layers**, and examine the code listing below it.

Compile the model and examine the layers

In [ ]:
```
cnn.compile(optimizer = 'adam',
            loss = 'categorical_crossentropy',
            metrics = ['accuracy'])

cnn.summary()
```

The `compile()` method takes the Keras CNN object built in the previous code block and configures it for training.

- `optimizer` specifies the loss optimization method to use. The `adam` method is similar to stochastic gradient descent (SGD).
- `loss` is the actual loss function to use. The `categorical_crossentropy` function is used with multi-class classification; it measures how much the predicted probability for a class diverges from the actual class label. The lower the value, the better. For example, if the network predicted a 0.97 for an image that was actually a 0, this would be a very large discrepancy and therefore lead to a higher loss value.
- `metrics` specifies a scoring metric to use besides just the loss. To keep things simple, you'll be evaluating accuracy.

b) Select the cell that contains the code listing, then select **Run**.

c) Examine the output.

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 28, 28, 32)        320
_________________________________________________________________
leaky_re_lu_1 (LeakyReLU)    (None, 28, 28, 32)        0
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 14, 14, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 14, 14, 64)        18496
_________________________________________________________________
leaky_re_lu_2 (LeakyReLU)    (None, 14, 14, 64)        0
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 7, 7, 64)          0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 7, 7, 128)         73856
_________________________________________________________________
leaky_re_lu_3 (LeakyReLU)    (None, 7, 7, 128)         0
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 4, 4, 128)         0
_________________________________________________________________
flatten_1 (Flatten)          (None, 2048)              0
_________________________________________________________________
dense_1 (Dense)              (None, 10)                20490
=================================================================
Total params: 113,162
Trainable params: 113,162
Non-trainable params: 0
```

This provides an overview of the CNN's structure. It lists each layer's type, its output shape, and the number of parameters at each convolution. The number of parameters is defined as: `number of output filters × (number of input filters × filter size + 1)`.

d) Scroll down and select the next code listing cell.

e) Select **Run**.

f) Wait for the `graphviz` library to download and install.

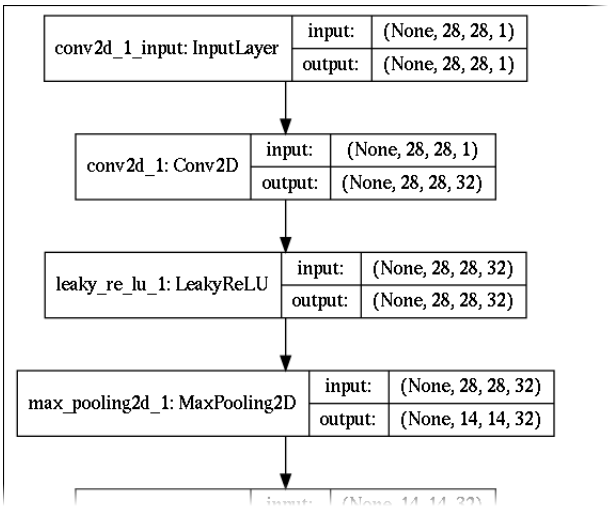
> **Note:** The library may still be installed from when you ran this code in the previous lab.

g) Scroll down and select the next code listing cell.

```
In [ ]:
1 from keras.utils import plot_model
2 plot_model(cnn, show_shapes = True, to_file = 'model.png')
```

h) Select **Run**.

i) Examine the output.



This is a more visual way of displaying the different layers of the CNN. Note that the input and output sizes change between layers. The size of each layer appears to decrease as you get closer and closer to the final output layer (dense network).

9. Train the model.

a) Scroll down and view the cell titled **Train the model**, and examine the code listing below it.

Train the model

```
In [ ]:
1 cnn_trained = cnn.fit(X_train, y_train,
2                       validation_data = (X_val, y_val),
3                       epochs = 1,
4                       verbose = 1)
```

You're using the training dataset along with the validation set to fit the CNN model. Due to time constraints, you'll only train for one epoch. In a real-world scenario, you'd want to train for several epochs.

b) Select the cell that contains the code listing, then select **Run**.

c) Examine the output.

```
Train on 45000 samples, validate on 15000 samples
Epoch 1/1
14272/45000 [========>.....................] - ETA: 52s - loss: 0.9581 - accuracy: 0.7890
```

Keras provides a way to observe the progress of the training while it is underway.

Note: It may take up to 10 minutes for training to complete.

d) While you wait, observe how the training loss decreases over time, while the accuracy gradually increases.

e) When training is complete, examine the final scores on the validation set.

```
Train on 45000 samples, validate on 15000 samples
Epoch 1/1
45000/45000 [==============================] - 81s 2ms/step - loss: 0.5645 - accuracy: 0.8431 - val_loss: 0.4933 - val_a
ccuracy: 0.8305
```

Note: Since there is some degree of randomness, your results may not align exactly with the results in the screenshot.

10. Evaluate the model on the test data.

a) Scroll down and view the cell titled **Evaluate the model on the test data**, and examine the code listing below it.

Evaluate the model on the test data

In [ ]:
```
1 eval_test = cnn.evaluate(X_test, y_test, verbose = 0)
2
3 print('Loss: {}'.format(round(eval_test[0], 2)))
4 print('Accuracy: {:.0f}%'.format(eval_test[1] * 100))
```

Since your test set is labeled, you'll evaluate the model on that test set as well.

b) Select the cell that contains the code listing, then select **Run**.

c) Examine the output.

```
Loss: 0.52
Accuracy: 83%
```

The scores on the test set should be fairly close to the scores on the validation set.

11. Make predictions on the test data.

a) Scroll down and select the code cell beneath the **Make predictions on the test data** title, then select **Run**.

b) Examine the output.

```
Actual class:    [9 2 1 1 6 1 4 6 5 7]
Predicted class: [9 2 1 1 6 1 2 6 5 7]
```

For the first 10 images, most of the predictions align with the actual class labels.

Note: As before, your results may differ due to randomness.

12. Visualize the predictions for several examples.

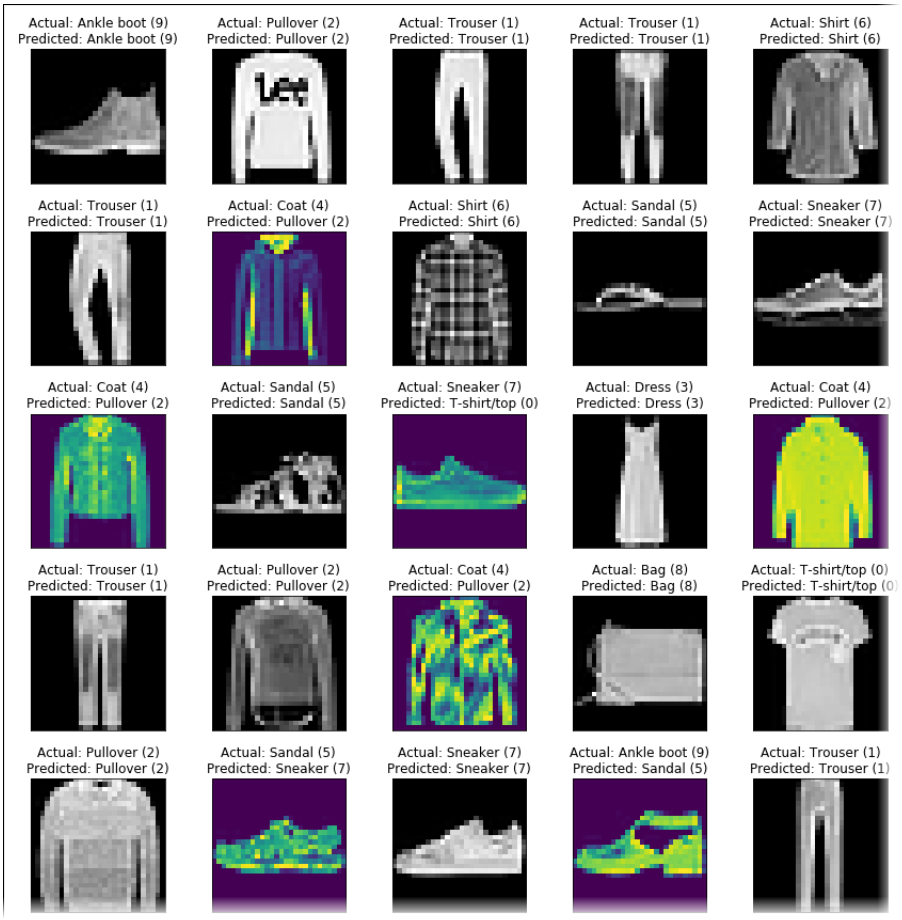
a) Scroll down and view the cell titled **Visualize the predictions for several examples**, and examine the code listing below it.

```
1  fig, axes = plt.subplots(nrows = 5, ncols = 5, figsize = (12, 12))
2
3  for i, ax in zip(range(25), axes.flatten()):
4
5      if actual[i] == prediction[i]:
6          ax.imshow(X_test[i].reshape(28, 28), cmap = 'gray')
7      else:
8          ax.imshow(X_test[i].reshape(28, 28))  # Highlight wrong predictions.
9
10     ax.title.set_text('Actual: {} ({})\nPredicted: {} ({})'.format(class_names[actual[i]], actual[i],
11                                                                   class_names[prediction[i]], prediction[i]))
12
13 # Turn off axis ticks for readability.
14 for ax in axes.flatten():
15     ax.set_xticks([])
16     ax.set_yticks([])
17
18 fig.tight_layout()
```

This will create a grid of images, where each image has its actual class label and the label that the model predicted. The `if` loop from lines 5 through 8 detects incorrect predictions and "highlights" any by plotting that image with a color palette. All correct predictions will be displayed as grayscale images.

b) Select the cell that contains the code listing, then select **Run**.

c) Examine the output.



Of the first 25 images, a few were incorrectly classified.

	**Note:** As before, your results may differ due to randomness.

13. Examine the incorrect predictions in the preceding screenshot (rather than your own results). Focus on the actual label vs. the predicted label for each image.

What can you tell about these incorrect predictions? From the perspective of your own human judgment, does it make sense that these images might be misclassified in the way that they were?

14. **What are some ways you might retrain this CNN model to improve its skill?**

15. Shut down the notebook and close the lab.

a) From the menu, select **Kernel→Shutdown**.
b) In the **Shutdown kernel?** dialog box, select **Shutdown**.
c) Close the lab browser tab and continue on with the course.

LAB 4-7
Building an RNN

Data File

~/Neural Networks/NeuralNetworks-IMDb.ipynb

Scenario

Another major component of your online storefront is user reviews. Users can leave written reviews for any product they purchase. These are typically more insightful than the normal five-star rating system, as such a system does not allow for nuanced and well-formed opinions. The business can learn a lot about which products are doing well in the public eye and which are doing poorly by considering these reviews. Up until now, human personnel have been reading through each review to determine whether that review is positive, negative, or neutral. This is tedious and an unnecessary waste of time, so you want to automate the process.

A machine should be able to quickly "read" through each review and, based on its language usage, determine whether the user did or did not like a product. This is sentiment analysis—a natural language processing (NLP) task—and one that can be dealt with by creating a recurrent neural network (RNN). So, you'll create a review classifier using an RNN.

Note: Due to the stochastic nature of TensorFlow algorithms, there will be some randomness with your results. If you wish to make the results more deterministic, uncomment the relevant lines in the first code block.

---

1. Open the lab and notebook.
   a) From the Coursera lab page, select **Open Lab** to open a Jupyter Notebook session in another web browser tab.
   b) Select **Neural Networks/NeuralNetworks-IMDb.ipynb** to open it.

2. Load the dataset.
   a) View the code cell beneath the **Import software libraries and load the dataset** title, then select **Run**.
   b) Verify that the training and test sets were both loaded with 25,000 records.
      - This is a dataset of movie reviews from IMDb, an online database that catalogues various information about movies, TV shows, and other video content. The dataset was loaded with Keras, which you'll use to build and train an RNN.
      - Rather than load every value of the dataset, the `num_words` argument is limiting the values to only the 10,000 most common words. Any word not within these parameters is represented with an out-of-value character. Constraining the dataset in this manner will help reduce training time.

3. Get acquainted with the dataset.
   a) Scroll down and select the code cell beneath the **Get acquainted with the dataset** title, then select **Run**.

b) Examine the output.

```
First example features:

[1, 14, 22, 16, 43, 530, 973, 1622, 1385, 65, 458, 4468, 66, 3941, 4, 173, 36, 256, 5, 25, 100, 43, 838, 112, 50, 670, 2
, 9, 35, 480, 284, 5, 150, 4, 172, 112, 167, 2, 336, 385, 39, 4, 172, 4536, 1111, 17, 546, 38, 13, 447, 4, 192, 50, 16,
6, 147, 2025, 19, 14, 22, 4, 1920, 4613, 469, 4, 22, 71, 87, 12, 16, 43, 530, 38, 76, 15, 13, 1247, 4, 22, 17, 515, 17,
12, 16, 626, 18, 2, 5, 62, 386, 12, 8, 316, 8, 106, 5, 4, 2223, 5244, 16, 480, 66, 3785, 33, 4, 130, 12, 16, 38, 619, 5,
25, 124, 51, 36, 135, 48, 25, 1415, 33, 6, 22, 12, 215, 28, 77, 52, 5, 14, 407, 16, 82, 2, 8, 4, 107, 117, 5952, 15, 256
, 4, 2, 7, 3766, 5, 723, 36, 71, 43, 530, 476, 26, 400, 317, 46, 7, 4, 2, 1029, 13, 104, 88, 4, 381, 15, 297, 98, 32, 20
71, 56, 26, 141, 6, 194, 7486, 18, 4, 226, 22, 21, 134, 476, 26, 480, 5, 144, 30, 5535, 18, 51, 36, 28, 224, 92, 25, 104
, 4, 226, 65, 16, 38, 1334, 88, 12, 16, 283, 5, 16, 4472, 113, 103, 32, 15, 16, 5345, 19, 178, 32]


Label: 1
```

- This output shows the features from just the first data example (i.e., a movie review).
- Each feature represents, in order, a word in the review.
- The numerical value of each feature represents the "rank" of the word in terms of its frequency within the dataset. Very common words are assigned lower numbers, whereas uncommon words are assigned very high numbers.
- The label of this example is 1. This dataset poses a binary classification problem, in which 1 represents a review that offers a positive sentiment about a movie, and 0 represents a review that offers a negative sentiment about a movie. Neutral reviews are not included in this dataset.

c) Scroll down and select the next code listing cell.

```
In [ ]:
# Decode sequence values into actual text.
index = datasets.imdb.get_word_index()
index_dict = dict([(value, key) for (key, value) in index.items()])
decode = ' '.join([index_dict.get(i - 3, '?') for i in X_train[0]])  # Replace unknown words with '?'
print(decode)
```

d) Select **Run**.

e) Examine the output.

```
? this film was just brilliant casting location scenery story direction everyone's really suited the part they played an
d you could just imagine being there robert ? is an amazing actor and now the same being director ? father came from the
same scottish island as myself so i loved the fact there was a real connection with this film the witty remarks througho
ut the film were great it was just brilliant so much that i bought the film as soon as it was released for ? and would r
ecommend it to everyone to watch and the fly fishing was amazing really cried at the end it was so sad and you know what
they say if you cry at a film it must have been good and this definitely was also ? to the two little boy's that played
the ? of norman and paul they were just brilliant children are often left out of the ? list i think because the stars th
at play them all grown up are such a big profile for the whole film but these children are amazing and should be praised
for what they have done don't you think the whole story was so lovely because it was true and was someone's life after a
ll that was shared with us all
```

- Each numerical feature value has been converted to a word. This is done by mapping the values to an existing dictionary for the dataset.
- A question mark (?) indicates a word that is not known.
- Comparing the actual words to their numerical values, you can see that the rankings make sense. For example, the second word ("this") has a rank of 14, which is rather common. The eighth word ("location") has a rank of 1,622, indicating it is much less common.
- The text of the review is streamlined; there is no punctuation, nor are there any capital letters. This helps to simplify the input for the neural network.
- Despite this, the text of the review is still legible, and you can see why it was labeled as being positive.

4. Examine some statistics about the reviews.

a) Scroll down and select the code cell beneath the **Examine some statistics about the reviews**, then select **Run**.

b) Examine the output.

```
Mean review length (in words): 239
Standard deviation (in words): 176
```

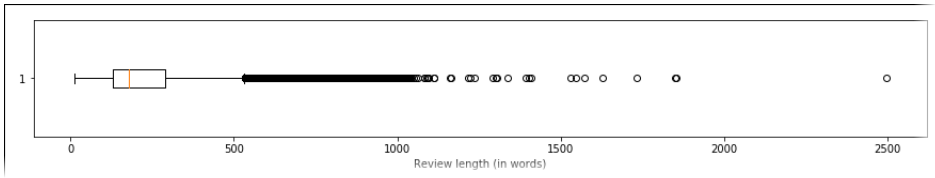
- The average length of a review is around 239 words.
- The standard deviation is around 176 words.

c) Scroll down and select the next code listing cell.

```
In [ ]:
1 plt.figure(figsize = (15, 2))
2 plt.boxplot(result, vert = False)
3 plt.xlabel('Review length (in words)')
4 plt.show()
```

d) Select **Run**.

e) Examine the output.



- This box plot confirms the mean and standard deviation figures.
- Looking at the upper whisker, it appears that the vast majority of reviews are under 500 words long. In an effort to simplify the model, you'll use this number as a guide when feeding these reviews as input.

5. Add padding to the data.

a) Scroll down and view the cell titled **Add padding to the data**, and examine the code listing below it.

Add padding to the data

```
In [ ]:
1 from keras.preprocessing import sequence
2
3 X_train = sequence.pad_sequences(X_train, maxlen = 500)
4 X_test = sequence.pad_sequences(X_test, maxlen = 500)
5
6 print('Number of features: {}'.format(X_train.shape[1]))
```

- Because the majority of the reviews are 500 words long, you'll truncate all of the reviews to that length. This is part of what the `pad_sequences()` function does.
- For reviews that are under 500 words long, `pad_sequences()` adds padding so that they are all that same size. This will make all of the input the same shape, simplifying the model.

b) Select the cell that contains the code listing, then select **Run**.

c) Examine the output.

```
Number of features: 500
```

This confirms that the number of features in each data example is 500. In other words, there are 500 words (known or unknown) for each review.

6. Split the datasets.

a) Scroll down and select the code cell beneath the **Split the datasets** title, then select **Run**.

b) Examine the output.

```
Training features:         (18750, 500)
Validation features:       (6250, 500)
Training labels:           (18750,)
Validation labels:         (6250,)
```

You're splitting the training dataset in order to have a validation holdout. Although the test set you loaded at the start is labeled, you'll treat it as the ultimate test case.

7. Build the RNN structure.

a) Scroll down and view the cell titled **Build the RNN structure**, and examine the code listing below it.

Build the RNN structure

In [ ]:

```
1  from keras.models import Sequential
2  from keras.layers import Embedding, LSTM, Dense
3  from keras.layers.advanced_activations import LeakyReLU
4
5  rnn = Sequential()
6
7  # Start stacking layers one-by-one.
8  rnn.add(Embedding(input_dim = 10000,  # Size of vocabulary (top 10,000 words).
9                    output_dim = 100,  # 100-dimensional vector embedding.
10                   input_length = 500))  # Length of review (in words).
11
12 rnn.add(LSTM(units = 64))  # 64-dimensional LSTM.
13 rnn.add(LeakyReLU(alpha = 0.1))
14
15 rnn.add(Dense(128, activation = 'linear'))
16 rnn.add(LeakyReLU(alpha = 0.1))
17 rnn.add(Dense(1, activation = 'sigmoid'))  # Dense output layer with sigmoid activation.
18
19 print('The RNN structure has been built.')
```

This code builds the actual structure of the RNN with Keras.

- You're using the `Sequential()` class to build the network structure layer-by-layer.
- Lines 8 through 10 add the first layer in the stack. The `Embedding()` object builds an embedded input layer. This particular layer has the following hyperparameters/arguments:
  - `input_dim` specifies the size of the input vocabulary. Recall that you constrained the input to the most common 10,000 words, so those are the dimensions of the input.
  - `output_dim` specifies the size of the embedding vector that the model will train. You could load pre-trained word embeddings, but in this case, the model will train its own. This model will create an embedding vector of 100 dimensions.
  - `input_length` specifies the total length of the input, which, in this case, is the number of words in a review. Recall that you truncated/padded each review so that they are all 500 words long.
- Line 12 adds a long short-term memory (LSTM) cell as the next layer. In this case, the cell's output will have 64 dimensions. This is an arbitrary size and can be tuned to improve performance, if necessary.
- Line 13 adds a leaky ReLU layer for the activation function.
- Lines 15 and 16 add a fully connected (dense) layer with leaky ReLU activation. In this case, the dense layer has an arbitrary size of 128.
- Line 17 is the final output layer. It has a single output and uses the sigmoid activation function to generate a binary classification decision.

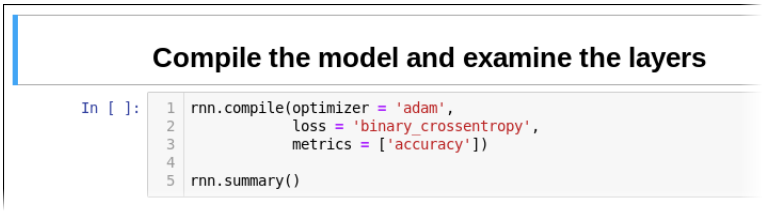
b) Select the cell that contains the code listing, then select **Run**.

```
The RNN structure has been built.
```

8. **What is word embedding, and why might it be beneficial to use in this case?**

9. Compile the model and examine the layers.

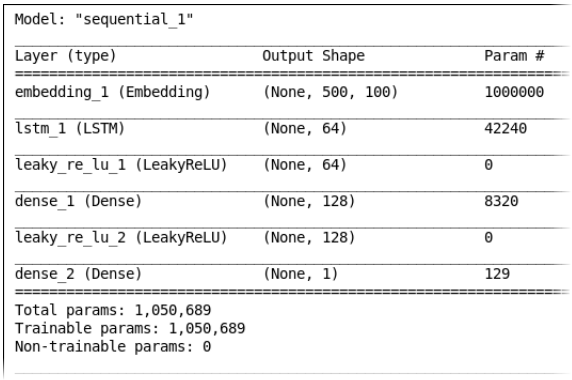
a) Scroll down and view the cell titled **Compile the model and examine the layers**, and examine the code listing below it.



The `compile()` method takes the Keras RNN object built in the previous code block and configures it for training. As with the CNN you built earlier, you'll use the `adam` optimizer and will return the model's accuracy score. You'll use the `binary_crossentropy` loss function, which is similar to `categorical_crossentropy`, except that it is more suited to binary classification problems.

b) Select the cell that contains the code listing, then select **Run**.

c) Examine the output.



This provides an overview of the RNN's structure. As with the CNN, each layer is listed, along with its output shape and number of parameters.

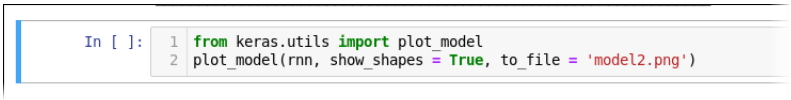
d) Scroll down and select the next code listing cell.

e) Select **Run**.

f) Wait for the `graphviz` library to download and install.

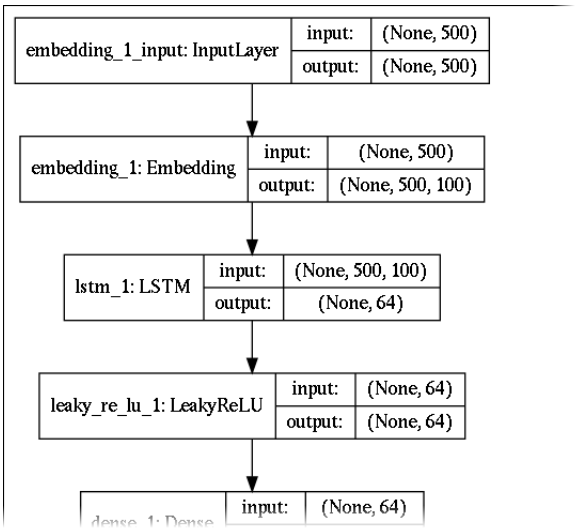
Note: The library may still be installed from when you ran this code in the previous lab.

g) Scroll down and select the next code listing cell.



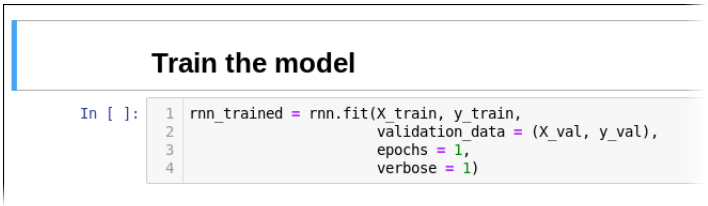
h) Select **Run**.

i) Examine the output.



10. Train the model.

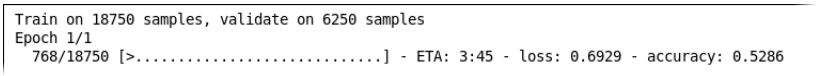
a) Scroll down and view the cell titled **Train the model**, and examine the code listing below it.



You're using the training dataset along with the validation set to fit the RNN model. Due to time constraints, you'll only train for one epoch. In a real-world scenario, you'd want to train for several epochs.

b) Select the cell that contains the code listing, then select **Run**.

c) Examine the output.



	Note: You can ignore the warning about converting to a dense tensor of unknown shape.
	Note: It may take up to 20 minutes for training to complete.

d) While you wait, observe how the training loss decreases over time, while the accuracy gradually increases.

e) When training is complete, examine the final scores on the validation set.

```
Train on 18750 samples, validate on 6250 samples
Epoch 1/1
18750/18750 [==============================] - 232s 12ms/step - loss: 0.4475 - accuracy: 0.7834 - val_loss: 0.3935 - val
_accuracy: 0.8522
```

Note: Since there is some degree of randomness, your results may not align exactly with the results in the screenshot.

11. Evaluate the model on the test data.

a) Scroll down and view the cell titled **Evaluate the model on the test data**, and examine the code listing below it.

Evaluate the model on the test data

In [ ]:
```
1 eval_test = rnn.evaluate(X_test, y_test, verbose = 0)
2
3 print('Loss: {}'.format(round(eval_test[0], 2)))
4 print('Accuracy: {:.0f}%'.format(eval_test[1] * 100))
```

Since your test set is labeled, you'll evaluate the model on that test set as well.

b) Select the cell that contains the code listing, then select **Run**.

c) Examine the output.

```
Loss: 0.4
Accuracy: 85%
```

The scores on the test set should be fairly close to the scores on the validation set.

Note: It may take a few minutes for the evaluation to finish.

12. Make predictions on the test data.

a) Scroll down and select the code cell beneath the **Make predictions on the test data** title, then select **Run**.

b) Examine the output.

```
Actual class:    [0 1 1 0 1 1 1 0 0 1]
Predicted class: [0 1 1 0 1 1 1 0 1 1]
```

The predictions align with the actual class labels for most of the reviews.

Note: As before, your results may differ due to randomness.

13. Examine a review that was correctly classified.

a) Scroll down and select the code cell beneath the **Examine a review that was correctly classified** title, then select **Run**.

b) Examine the output.

```
Actual review sentiment:    negative
Predicted review sentiment: negative

? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ?
? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ?
? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ?
? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ?
? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ?
? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ?
? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ?
? ? ? ? ? ? ? ? ? ? ? ? ? please give this one a miss br br ? ? and the rest of the cast rendered terrible performances
the show is flat flat flat br br i don't know how michael madison could have allowed this one on his plate he almost see
med to know this wasn't going to work out and his performance was quite ? so all you madison fans give this a miss
```

Note: The following summarizes the review in the preceding screenshot. The review that was pulled from your model may be different due to randomness.

- The model correctly predicted that this review is negative.
- Reading the review seems to confirm this—the user obviously didn't like the movie.
- Because this review is under 500 words long, it was padded at the beginning with unknown word values (question marks).

14. Examine a review that was incorrectly classified.

a) Scroll down and select the code cell beneath the **Examine a review that was incorrectly classified** title, then select **Run**.

b) Examine the output.

```
Actual review sentiment:    negative
Predicted review sentiment: positive

? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ?
? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ?
? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ?
? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ?
? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ?
? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? i generally love this type of movie however this time i found myself wanting to
kick the screen since i can't do that i will just complain about it this was absolutely idiotic the things that happen w
ith the dead kids are very cool but the alive people are absolute idiots i am a grown man pretty big and i can defend my
self well however i would not do half the stuff the little girl does in this movie also the mother in this movie is reck
less with her children to the point of neglect i wish i wasn't so angry about her and her actions because i would have o
therwise enjoyed the flick what a number she was take my advise and fast forward through everything you see her do until
the end also is anyone else getting sick of watching movies that are filmed so dark anymore one can hardly see what is b
eing filmed as an audience we are ? involved with the actions on the screen so then why the hell can't we have night vis
ion
```

Note: The following summarizes the review in the preceding screenshot. The review that was pulled from your model may be different.

- The model incorrectly predicted this review as being positive, when it is in fact negative.
- Reading the review, you can tell that it is negative, but you may be able to identify reasons why the model thought it was positive.
- As before, the review is padded to be 500 words long.

15. Examine the incorrect prediction in the preceding screenshot (rather than your own results).

Why do you think the model incorrectly classified this review as positive?

16. What are some ways you might retrain this RNN model to improve its skill?

17. Shut down the notebook and close the lab.

a) From the menu, select **Kernel→Shutdown**.
b) In the **Shutdown kernel?** dialog box, select **Shutdown**.
c) Close the lab browser tab and continue on with the course.

MODULE 5

Project

The following lab is for the Course 4 Project.

PROJECT 4

Building a CNN to Classify Handwritten Characters

Data File

~/Projects/CNN.ipynb

Scenario

A company that provides online ancestry search services has contracted with you to develop tools that they can use to convert historical documents scanned as images into content stored in searchable databases. Many of the original documents are handwritten, making the task more challenging.

As a proof-of-concept for how a CNN might be used to process scanned images, you will create a model to identify scanned characters in the MNIST database, which contains 60,000 scanned characters in the training set and 10,000 scanned characters in the test set.

Note: There is no "correct" answer for every line of code, as there are many ways to write code to do the same thing. You can write the code however you feel most comfortable, as long as it accomplishes the tasks set out for you.

Note: If you're stuck and need help, you can "borrow" code from existing notebooks, and/or look up documentation on how to use the various Python libraries.

Note: Sometimes, when you run code containing logic errors or bugs, you may corrupt the data contained in variables you created in previous code cells. To clear out such problems, you can select **Kernel→Restart & Clear Output**, then run each code cell again.

---

1. Open the lab and notebook.
   a) From the Coursera lab page, select **Open Lab** to open a Jupyter Notebook session in another web browser tab.
   b) Select **Projects/CNN.ipynb** to open it.
   c) Observe the notebook.

      Placeholder code cells have been provided in which you can add your own code. Comments provide hints on tasks you might perform in each code cell. The first code cell has already been completely programmed for you.

2. Import software libraries and load the dataset.
   a) Select the code listing under **Import software libraries and load the dataset**.
   b) Run the code and examine the results.

      The dataset is loaded for you. There are 60,000 training records and 10,000 test records.

3. Get acquainted with the dataset.
   a) In the code block under **Get acquainted with the dataset**, write statements to show the dimensions of the training and testing sets and their labels.
   b) Run the code cell and verify that the shape of the training data, training labels, testing data, and testing labels are shown.

4. Visualize the data examples.
   a) In the code block under **Visualize the data examples**, write statements to show a preview of the first 20 images in the training dataset. Label each image with its class, which you can obtain from the training label set.
   b) Run the code cell and verify that the first twenty images are shown along with their labels.

5. Prepare the data for training with Keras.
   a) In the code block under **Prepare the data for training with Keras**, write statements to reshape arrays to add the grayscale flag, and use one-hot encoding to encode the data for each label. Print the one-hot encoding data for the first image to confirm that it's being generated properly.
   b) Run the code cell and verify that one-hot encoding for the first image is shown.

6. Split the datasets.
   a) In the code block under **Split the datasets**, write statements to split the training and validation datasets and their labels. To confirm the split, print the shape of the training set, validation set, training labels, and validation labels.
   b) Run the code cell and verify that the data is being split into a training set, validation set, training label set, and validation label set.

7. Build the CNN structure.
   a) In the code block under **Build the CNN structure**, write statements to import the required libraries, create the model, add model layers, and print a message to confirm the structure has been built. The layers you will use for this model include:
      - Conv2D, 64 nodes, kernel size of 3, with ReLU activation, and an input shape of (28, 28, 1)
      - Conv2D, 32 nodes, kernel size of 3, with ReLU activation
      - Flatten (to connect the convolution and dense layers)
      - Dense, 10 nodes, with softmax activation
   b) Run the code cell and verify that the structure has been built.

8. Compile the model and summarize the layers.
   a) In the code block under **Compile the model and summarize the layers**, write statements to compile the model and summarize its layers once it's been compiled. Use the following options when you compile:
      - Use the `adam` optimizer.
      - Use the `categorical_crossentry` loss function.
      - Use the `accuracy` metric.
   b) Run the code cell and verify that the model has been compiled using the layers you specified.

9. Plot a graph of the model.
   a) In the code block under **Plot a graph of the model**, write statements to plot a graph of the model.
   b) Run the code cell and verify that the graph is shown.

10. Train the model.
   a) In the code block under **Train the model**, write statements to train the model over 1 epoch using the training data and labels.
   b) Run the code cell and wait for the model to be trained.

11. Evaluate the model on the test data.
   a) In the code block under **Evaluate the model on the test data**, write statements to evaluate the model on the test data, printing messages to show the loss and accuracy.
   b) Run the code cell and observe the loss and accuracy values.

12. Make predictions on the test data.

a) In the code block under **Make predictions on the test data**, write statements to make predictions on the test data, showing the first 30 examples of actual values compared to predictions.
b) Run the code cell and compare the actual values to the predicted values.

13. Visualize the predictions for 30 examples.

a) In the code block under **Visualize the predictions for 30 examples**, write statements to show the first 30 predictions along with the image, highlighting any incorrect predictions in color.
b) Run the code cell and compare the actual values to the predicted values.

14. Save and download the notebook.

a) From the menu, select **File→Save and Checkpoint**.
b) Select **File→Download as→Notebook (.ipynb)**.
c) Save the file to your local drive if it wasn't automatically.
d) Find the saved file and rename it using the following convention: **FirstnameLastname-CNN-Project.ipynb**.

For example: **JohnSmith-CNN-Project.ipynb**.

Note: Make sure not to lose this file. This is the project you'll be uploading and sharing to your peers for grading.

15. Shut down the notebook and close the lab.

a) Back in Jupyter Notebook, select **Kernel→Shutdown**.
b) In the **Shutdown kernel?** dialog box, select **Shutdown**.
c) Close the lab browser tab and continue on to the peer review.

Solutions

LAB 2-3: Exploring the General Structure of the Dataset

3. Which attributes do you think might have an influence on price?

A: Some attributes might seem important from a commonsense perspective—such as the lot size (`sqft_lot`), size of the living space, and whether the property is waterfront or has a view. Others such as location (`zipcode`, `lat`, and `long`) might be significant if they correspond to expensive neighborhoods. Other attributes might have a surprising influence on the price. Performing some statistical analysis will help to reveal which values actually correlate with price.

LAB 2-5: Analyzing a Dataset Using Visualizations

5. What patterns seem to exist regarding house prices and location?

A: There seems to be a cluster of expensive homes located around the lakes in Seattle and Bellevue (Lake Washington and Lake Sammamish). But there are also some expensive homes away from the lakes. Less expensive homes seem to be located in areas away from Seattle and Bellevue, such as the southern part of the county (Federal Way, Auburn, Covington, Maple Valley, and Enumclaw).

LAB 2-7: Setting Up a Machine Learning Model

2. Why would you use a linear regression algorithm to produce a real estate price estimator?

A: The purpose of the model is to predict the price at which a house with particular attributes will sell. This is a regression type of outcome. While other algorithms can also be used to produce a regression outcome, linear regression is simple, relatively easy to implement, and can produce a good result in many cases.

LAB 3-2: Building a Regularized Linear Regression Model

12. Are you satisfied with this MSE value? In other words, would you stop there and finalize the model? Why or why not?

A: Answers may vary. Since there is not one truly "correct" MSE value to shoot for, the decision to stop may come down to what's "good enough," or when the results no longer change significantly. Even though the second round of training produced a lower MSE, there may still be plenty of opportunity to continue tuning the hyperparameters to see if you can get an even better result.

LAB 3-3: Building an Iterative Linear Regression Model

6. Why is it important to scale down features, such as through standardization, when using an iterative cost minimization technique like gradient descent?

A: Scaling down features helps the model converge on the cost minimum faster, saving on training time.

10. Why is this?

A: The nature of a closed-form solution means that it will determine the model parameters that *best* minimize the cost function. Iterative approaches can get close to the best error value, but they can't do better.

LAB 3-4: Training Binary Classification Models

5. Given what you know about the dataset thus far, what features do you think might influence the survival rates?

A: Answers may vary. A passenger's socioeconomic status (`Pclass` and `Fare`) may correlate with how that passenger's rescue was prioritized compared to others. `Age` might also be a factor, as older passengers may have been slower to flee danger. `SibSp` and `Parch` might also influence survival rates, as passengers who traveled alone may not have received the same amount of help during rescue attempts as those who traveled with loved ones. Given the policy of "women and children first," `Sex` could also influence survival rate.

6. Why is such a correlation not relevant in classification problems like this one?

A: A correlation indicates how values increase or decrease in relation to one another. Since a classification label like `Survived` is categorical and not a continuous numeric variable, a correlation will not reveal useful information.

LAB 3-6: Evaluating a Classification Model

13. What does each quadrant indicate in terms of predicting survivors of the *Titanic*?

A: The top-left quadrant indicates that there were 132 instances where the model predicted a passenger would perish, and was correct in its prediction. The top-right quadrant indicates that there were 15 instances where the model predicted a passenger would survive, but was incorrect—those passengers died. The bottom-left quadrant indicates that there were 35 instances where the model predicted a passenger would perish, but was incorrect—those passengers survived. The bottom-right quadrant indicates that there were 41 instances where the model predicted a passenger would survive, and was correct in its prediction.

15. In what situation are precision and recall a better measure of a model's skill than accuracy?

A: Accuracy tends to only be useful in datasets where the class label values are balanced. In datasets with a class imbalance, accuracy may end up being high and yet entirely useless. So, precision and recall are a better summary of a model's skill when a class imbalance is present.

16. Is there any one of these measures you'd be more interested in optimizing than the others? Why or why not?

A: There is no "right" answer, necessarily, because it all comes down to the nature of the dataset and the problem you're trying to solve, which requires a somewhat subjective assessment. Because the class imbalance is rather small, you could argue that high accuracy is an acceptable target to shoot for. You could also argue that precision is important if you're trying to avoid false positives; for example, you want to avoid giving a passenger's relatives false hope by telling them the passenger survived when he or she did not. Recall might be more important to you if you're more concerned about minimizing false negatives; for example, you may want to avoid pronouncing someone as having perished when they actually survived, as that would cause problems for someone who has already suffered a great deal. Or, you may have no clear preference between precision and recall; in which case, the F_1 score is a good way to optimize for both. Ultimately, because this is a mostly academic exercise that isn't going to be applied for any serious purposes, it may be best to try and optimize all of these metrics.

18. What are the advantages of a ROC curve over a precision–recall curve, and vice versa? Given your domain knowledge, are you more interested in improving one of these curves over the other? Why or why not?

A: Once again, there is no "right" answer. ROC and its AUC are good for evaluating classifications for all possible thresholds, which can help you optimize multiple types of errors. However, the precision–recall curve tends to be better at minimizing either one of these errors, which is often more useful in cases of class imbalance. For the *Titanic* dataset, the ROC curve may be sufficient, but you could also argue that the point of this model is to predict examples of the minority class (survived), which a precision–recall curve is best at summarizing.

LAB 3-7: Tuning a Classification Model

4. Compared to the initial model, how has the confusion matrix changed for the optimized model?

A: There are slightly fewer true negatives; slightly more false positives; fewer false negatives; and more true positives.

6. What else might you do to continue improving your classification performance for this dataset?

A: Answers will vary. Tuning is an iterative process, and you'll often not be done after just the first iteration. You could continue to test your model's performance by optimizing for a metric other than F_1 score, such as optimizing for precision, recall, AUC, etc. You might also want to revisit your data preparation tasks to see if you can do more to optimize the data itself before training. In addition, rather than comparing models that use the same algorithm but different hyperparameters, you could try training a model using a different classification algorithm to see if it performs better than logistic regression.

LAB 3-8: Building a k-Means Clustering Model

8. How did the clusters form with regard to location?

A: Each cluster appears to be its own quadrant on the map, with cluster 0 representing the southeast; cluster 1 representing the southwest; cluster 2 representing the northeast; and cluster 3 representing the northwest.

10. How did the clusters form with regard to price per square foot?

A: Cluster 1 includes the lowest-priced homes; cluster 3 includes the second-lowest-priced homes; cluster 0 includes the second-highest-priced homes; and cluster 2 includes the highest-priced homes. Other than the heat map, this is also exhibited in the fact that the size of each data point corresponds to its total price.

15. Do you think this model is adequate in solving the problem of recommending similar houses to buyers who have expressed interest in a specific house? Why or why not?

A: Answers will vary. Given its unsupervised nature, it's very difficult to evaluate the performance of a clustering model. Your best weapons are knowing what you want out of the model, and performing clustering analysis. In this case, the number of clusters isn't supported much by domain knowledge; in other words, there's no set number of groups that houses need to fall into. However, you might argue that more clusters will be helpful to the real estate agents, as they will narrow down the houses more. Still, you may have to trust in analysis methods like silhouette analysis and elbow plots. Although the former was chosen to influence the final model, you could choose the latter and it would be no less valid. You might also find value in doing further analysis by plotting the clusters in more than just the latitude and longitude dimensions. Going back to domain knowledge and what you want out of the model, you may determine that some features are more important than others, which might influence your clustering decisions.

16. What are some reasons why this model may need to be retrained over time?

A: Answers may vary. There are several factors that could potentially require a retrain of this clustering model, most of which have to do with the dataset. In particular, this dataset was captured at a certain point in time, and therefore may not reflect future changes. For example, housing trends come and go, and what may have been a desirable trait when this data was collected may not be as desirable in the future. Likewise, changing economic factors can greatly affect the value of a house from year to year, so you'll likely need to update the model to account for this.

LAB 3-9: Building a Hierarchical Clustering Model

10. If this were a real-world problem rather than an artificial dataset, how might domain knowledge of that dataset help you determine the optimal number of clusters?

A: Answers may vary. Domain knowledge can inform the optimal number of clusters in several ways. The most obvious is if each data example needs to be placed into one of n groups, with n being known by you. Even if you don't have a specific number of groups in mind, the dataset might place certain constraints on the outcome, like bounding the number of clusters within a range. For example, if your objective is to create a recommendation system for customers based on their shared attributes, you might want to have less than five or so clusters, as it can be difficult to manage many different groups of customers. Alternatively, you might want to have a large number of clusters so that your recommendations are more targeted and therefore more valuable to smaller groups of customers.

LAB 4-1: Building a Decision Tree Model

5. Which of these remaining features do you think need to be one-hot encoded, and why?

A: One obvious candidate is `Sex`, as not only are the values categorical, they are currently in strings. The same goes for `Embarked`—there are only three options, each one a string. In both cases, directly converting the strings to numbers will be problematic, as the decision tree algorithm will think that 1 is ranked higher than 0. This is why one-hot encoding these features is important. Another candidate for one-hot encoding that might not be so obvious is `Pclass`. While the concept of passenger class implies some sort of order, the order is actually deceptive; a class of 1 (first class) is actually ranked higher than a class of 2 (second class). However, the decision tree algorithm will see this as the exact opposite, and class 3 will be ranked above the others. Therefore, it's safest to just one-hot encode each class type.

20. Are you satisfied with the results of the decision tree as compared to the logistic regression model? Do you think one model is more skillful than the other? If so, which one, and why?

A: Answers will vary greatly. Model evaluation and selection does not produce an objectively correct answer; much of it is up to the machine learning practitioner's judgment and expectations. If you happen to value precision more than recall for this particular problem, you might be more willing to depend on the decision tree model, as it produced a higher degree of decision. If you're focused on recall, then you may prefer the logistic regression model. When it comes to accuracy, F_1 score, and even ROC AUC and average precision, the results were not all that different. Likewise, you may be unsatisfied in the sense that you'd like to do more hyperparameter optimization to potentially get even better results from either model. After all, both searches only had narrow fields of hyperparameters to select from. You may also wish to withhold judgment until you've put the data through even more classification algorithms, as the optimal model may still be out there.

LAB 4-2: Building a Random Forest Model

3. Why is cross-validation typically not necessary when training a random forest model?

A: The bagging technique used in random forests randomly samples the training dataset for each individual tree in the forest; this ensures that the entire forest sees a representative sampling of the data. In other words, bagging helps to reduce overfitting in a way similar to cross-validation, so the latter is not strictly necessary.

12. How do the first two branches of each tree differ in terms of their splitting criteria?

A: The first tree (index 0) starts by splitting male and female passengers at the root decision node. It then splits based on whether or not female passengers are in third class. At the same time, it splits male passengers based on whether or not they have fewer than three siblings plus a spouse. The second tree (index 1) starts by splitting based on whether or not the passengers are in first class. It then splits based on whether passengers not in first class are male or female. At the same time, it splits first class passengers into male or female as well.

14. What advantage does a random forest have over a single decision tree?

A: Decision trees are prone to overfitting to the training data, whereas random forests reduce variance and therefore help mitigate overfitting issues.

21. How might you retrain the model to improve these scores even further?

A: Answers may vary, but the model might be improved by dropping another feature—for example, you could try dropping `Fare` since it had the least importance of the four features that were kept. This might be successful if `Fare` only adds noise to the data and doesn't truly contribute to the model's predictive power. Alternatively, you might add a feature back to the training—for example, `Pclass_1`, since it had the highest importance of the features that were dropped. This might improve the model if including `Pclass_1` contributes to the model's predictive power. Ultimately, you may need to try several different approaches and see which one works the best for your goals.

LAB 4-3: Building an SVM Model for Classification

6. Why are SVMs often better than other algorithms at handling datasets with outliers?

A: SVMs create margins of separation that help its decision boundary stay as far away from edge cases as possible, which helps reduce the effect of outliers on the model. Other algorithms may fail to handle outliers in this way, potentially overfitting the model.

12. How does this SVM boundary fit to the data as compared to the logistic regression boundary?

A: The SVM boundary seems to stay away from the edge data examples more successfully than the logistic regression model. It also incorporates the support-vector margins, where a few of the outliers are at or near those margins. Also, the outlier mentioned before appears to be classified correctly this time.

LAB 4-4: Building an SVM Model for Regression

10. How does SVM regression differ from SVM classification in terms of how the data examples are included or not included within the margins?

A: In classification, the ideal is to separate each data example so that none of the examples are placed within the margins. The support vectors are the data examples on the edge of the margins. In regression, the ideal is to include as many data examples as possible *within* the margins. All of the data examples outside of the margins are the support vectors.

LAB 4-5: Building an MLP

16. How does backpropagation generate the weights between the neurons of different layers in an MLP neural network?

A: A prediction is made for an example, and then the error between the prediction and values is calculated. Starting from the last hidden layer, the network computes how much each neuron in the hidden layer contributed to the error in each output layer neuron. This process is repeated for the next-to-last hidden layer, and so on, until reaching the input layer. The weights that were just returned were updated to account for the error gradients between neurons.

17. What can you tell about the weights of this particular network structure?

A: From the input to the hidden layer, the highest weights appear to be from features 3, 4, and 6 (`Light`, `CO2`, and `Day`), to both neurons in the hidden layer. Features 3 and 4 have relatively high positive weights with hidden neuron 1, whereas they tend to have relatively high negative weights with hidden neuron 2. The rest of the input features have much less significant weights that vary between positive and negative. From the hidden layer to the output layer, hidden neuron 1 has a high positive weight, whereas hidden neuron 2 has a high negative weight.

LAB 4-6: Building a CNN

13. What can you tell about these incorrect predictions? From the perspective of your own human judgment, does it make sense that these images might be misclassified in the way that they were?

A: Answers may vary. In most of these incorrect cases, the predicted class of clothing seemed to be visually similar to the actual class. For example, the image in row 2, column 2 was classified as a pullover by the model, but is actually a coat. A pullover and a coat are pretty similar, and by looking at the image itself, you might not even be able to determine the difference. There is at least one instance of an incorrect prediction being pretty far off: the image in row 3, column 3. The model classified this image as a t-shirt, but it is very clearly some kind of shoe (a sneaker, to be precise).

14. What are some ways you might retrain this CNN model to improve its skill?

A: Answers may vary. The model will very likely improve both its loss score and its accuracy by training over multiple epochs, rather than just one. Also, changing the size of the convolutions, as well as the size of their filters, may lead to a more skillful model. You won't always know the exact values to use for these hyperparameters, so it may be worth experimenting with different combinations to see if you can get better results.

LAB 4-7: Building an RNN

8. What is word embedding, and why might it be beneficial to use in this case?

A: Embedding is the process of representing a word in its own vector of n dimensions. Each vector combines into the network's overall embedded space. Words with similarities are placed closer within this space, improving the model's ability to recognize patterns. Embedding helps minimize the dimensionality of language-based inputs, which would otherwise require many thousands of features for each word in a vocabulary.

15. Why do you think the model incorrectly classified this review as positive?

A: There is not necessarily a correct answer, as you can't easily interrogate the model to find out the specifics of *why* it made a decision. However, it's possible that the model picked up on some key words and phrases that typically indicate positive sentiment. For example, the review opens by stating, "I generally *love this type of movie*, however …" The reviewer also praises one aspect of the movie as being "very cool." Later, were it not for a particularly unpleasant character, the reviewer states that he "would have otherwise *enjoyed the flick*." The network may have picked up on these words and phrases without understanding how context may have changed their meaning.

16. What are some ways you might retrain this RNN model to improve its skill?

A: Answers may vary. As with the CNN, the RNN model will very likely improve both its loss score and its accuracy by training over multiple epochs, rather than just one. Expanding the vocabulary above 10,000 words, or removing that constraint altogether, may also provide the model with more useful data. Lastly, reconfiguring the LSTM and dense layers to use different output sizes, as well as adding more layers, may also improve the model's skill. Once again, your best bet is to experiment with these hyperparameter configurations as much as you can.