

Prepare Build and Visualize Machine Learning Mode

Prepare

Exercise 1: Explore the data

Before you can prepare a dataset, you need to understand its content and structure. In the previous lab, you imported a dataset containing on-time arrival information for a major U.S. airline. That data included 26 columns and thousands of rows, with each row representing one flight and containing information such as the flight's origin, destination, and scheduled departure time. You also loaded the data into a Jupyter notebook and used a simple Python script to create a Pandas DataFrame from it.

A [DataFrame](#) is a two-dimensional labeled data structure. The columns in a DataFrame can be of different types, just like columns in a spreadsheet or database table. It is the most commonly used object in Pandas. In this exercise, you will examine the DataFrame — and the data inside it — more closely.

1. Return to the Data Science VM that you created in the previous lab. If you are not connected to it, use [X2Go](#) or the Xfce client of your choice to connect to the VM's Ubuntu desktop.
2. Open the Jupyter notebook that you created in the previous lab.

An easy way to reopen the notebook is to double-click the Jupyter icon on the desktop. Once Jupyter opens in a browser, click **flights** to navigate to the "flights" directory, and **FlightData.ipynb** to open the notebook.

The screenshot shows a Jupyter Notebook interface with the title 'FlightData'. The top bar indicates 'Last Checkpoint: 14 hours ago (autosaved)' and a 'Logout' button. The menu bar includes 'File', 'Edit', 'View', 'Insert', 'Cell', 'Kernel', and 'Help'. The 'Cell' menu is currently open, showing options like 'Run Cells', 'Run Cells and Select Below', 'Run Cells and Insert Below', 'Run All', 'Run All Above', and 'Run All Below'. The 'Run All' option is highlighted with a red box. Below the menu, the notebook content shows a code cell with the following Python code:

```
In [1]: import pandas as pd
df = pd.read_csv('flightdata.csv')
df.head()
```

The output of the code cell is displayed below it, showing the first five rows of the 'flightdata.csv' file. The output is a table with 10 columns: YEAR, QUARTER, MONTH, DAY_OF_MONTH, DAY_OF_WEEK, UNIQUE_CARRIER, TAIL_NUM, FL_NUM, and ORIGIN_AIRPORT_I. The first five rows are as follows:

	YEAR	QUARTER	MONTH	DAY_OF_MONTH	DAY_OF_WEEK	UNIQUE_CARRIER	TAIL_NUM	FL_NUM	ORIGIN_AIRPORT_I
0	2016	1	1	1	5	DL	N836DN	1399	1039
1	2016	1	1	1	5	DL	N964DN	1476	1143
2	2016	1	1	1	5	DL	N813DN	1597	1039
3	2016	1	1	1	5	DL	N587NW	1768	1474
4	2016	1	1	1	5	DL	N836DN	1823	1474

Below the table, it says '5 rows x 26 columns'.

The FlightData notebook

- Use the **Cell** -> **Run All** command to run the notebook.

This screenshot is similar to the previous one, but it focuses on the 'Cell' menu. The 'Cell' menu is open, and the 'Run All' option is highlighted with a red box. The notebook content shows the same code cell as before, but the output is partially obscured by the menu. The visible part of the output table is as follows:

	YEAR	QUA	MONTH	DAY_OF_MONTH	DAY_OF_WEEK	UNIQUE_CARRIER	TAIL_NUM	FL_NUM	ORIGIN_AIRPORT_I
0	2016				5	DL	N836DN	1399	1039
1	2016				5	DL	N964DN	1476	1143
2	2016	1	1	1	5	DL	N813DN	1597	1039
3	2016	1	1	1	5	DL	N587NW	1768	1474
4	2016	1	1	1	5	DL	N836DN	1823	1474

Below the table, it says '5 rows x 26 columns'.

Running the notebook

- The code that you added to the notebook in the previous lab creates a DataFrame from **flightdata.csv** and calls [DataFrame.head](#) on it to display the first five rows. One of the first things you typically want to know about a dataset is how many rows it contains. To get a count, use the **Insert** -> **Insert**

Cell Below command to insert a new cell into the notebook. Then type the following statement into the cell and run it by pressing **Ctrl+Enter**:

```
df.shape
```

Confirm that the DataFrame contains 11,231 rows and 26 columns:

```
In [2]: df.shape
```

```
Out[2]: (11231, 26)
```

Getting a row and column count

Column	Description
YEAR	Year that the flight took place
QUARTER	Quarter that the flight took place (1-4)
MONTH	Month that the flight took place (1-12)
DAY_OF_MONTH	Day of the month that the flight took place (1-31)
DAY_OF_WEEK	Day of the week that the flight took place (1=Monday, 2=Tuesday, etc.)
UNIQUE_CARRIER	Airline carrier code (e.g., DL)
TAIL_NUM	Aircraft tail number
FL_NUM	Flight number
ORIGIN_AIRPORT_ID	ID of the airport of origin
ORIGIN	Origin airport code (ATL, DFW, SEA, etc.)
DEST_AIRPORT_ID	ID of the destination airport
DEST	Destination airport code (ATL, DFW, SEA, etc.)
CRS_DEP_TIME	Scheduled departure time
DEP_TIME	Actual departure time
DEP_DELAY	Number of minutes departure was delayed
DEP_DEL15	0=Departure delayed less than 15 minutes 1=Departure delayed 15 minutes or more
CRS_ARR_TIME	Scheduled arrival time

ARR_TIME	Actual arrival time
ARR_DELAY	Number of minutes flight arrived late
ARR_DEL15	0=Arrived less than 15 minutes late 1=Arrived 15 minutes or more late
CANCELLED	0=Flight was not cancelled 1=Flight was cancelled
DIVERTED	0=Flight was not diverted 1=Flight was diverted
CRS_ELAPSED_TIME	Scheduled flight time in minutes
ACTUAL_ELAPSED_TIME	Actual flight time in minutes
DISTANCE	Distance traveled in miles

- Now take a moment to examine the 26 columns in the dataset. They contain important information such as the date that the flight took place (YEAR, MONTH, and DAY_OF_MONTH), the origin and destination (ORIGIN and DEST), the scheduled departure and arrival times (CRS_DEP_TIME and CRS_ARR_TIME), the difference between the scheduled arrival time and the actual arrival time in minutes (ARR_DELAY), and whether the flight was late by 15 minutes or more (ARR_DEL15).

Here is a complete list of the columns in the dataset. Times are expressed in 24-hour military time. For example, 1130 equals 11:30 a.m. and 1500 equals 3:00 p.m.

The dataset includes a roughly even distribution of dates throughout the year, which is important because a flight out of Minneapolis is less likely to be delayed due to winter storms in July than it is in January. But this dataset is far from being "clean" and ready to use. Let's write some Pandas code to clean it up.

Exercise 2: Clean the data

One of the most important aspects of preparing a dataset for use in machine learning is selecting the "feature" columns that are relevant to the outcome you are

trying to predict while filtering out columns that do not affect the outcome, could bias it in a negative way, or might produce [multicollinearity](#). Another important task is to eliminate missing values, either by deleting the rows or columns containing them or replacing them with meaningful values. In this exercise, you will eliminate extraneous columns and replace missing values in the remaining columns.

1. One of the first things data scientists typically look for in a dataset is missing values. There's an easy way to check for missing values in Pandas. To demonstrate, add a new cell to the notebook and execute the following code:

```
df.isnull().values.any()
```

Confirm that the output is "True," which indicates that there is at least one missing value somewhere in the dataset.

```
In [3]: df.isnull().values.any()
```

```
Out[3]: True
```

Checking for missing values

2. The next step is to find out where the missing values are. Add another cell to the notebook and execute the following code:

```
df.isnull().sum()
```

Confirm that you see the following output listing a count of missing values in each column:

YEAR	0
QUARTER	0
MONTH	0
DAY_OF_MONTH	0
DAY_OF_WEEK	0
UNIQUE_CARRIER	0
TAIL_NUM	0
FL_NUM	0
ORIGIN_AIRPORT_ID	0
ORIGIN	0
DEST_AIRPORT_ID	0
DEST	0
CRS_DEP_TIME	0
DEP_TIME	107
DEP_DELAY	107
DEP_DEL15	107
CRS_ARR_TIME	0
ARR_TIME	115
ARR_DELAY	188
ARR_DEL15	188
CANCELLED	0
DIVERTED	0
CRS_ELAPSED_TIME	0
ACTUAL_ELAPSED_TIME	188
DISTANCE	0
Unnamed: 25	11231

dtype: int64

Number of missing values in each column

- Curiously, the 26th column ("Unnamed: 25") contains 11,231 missing values, which equals the number of rows in the dataset. This column was mistakenly created because the CSV file that you imported contains a comma at the end of each line. To eliminate that column, add a new cell to the notebook and execute the following code:

- ```
df = df.drop('Unnamed: 25', axis=1)
df.isnull().sum()
```

Inspect the output and confirm that column 26 has disappeared from the DataFrame:

```

YEAR 0
QUARTER 0
MONTH 0
DAY_OF_MONTH 0
DAY_OF_WEEK 0
UNIQUE_CARRIER 0
TAIL_NUM 0
FL_NUM 0
ORIGIN_AIRPORT_ID 0
ORIGIN 0
DEST_AIRPORT_ID 0
DEST 0
CRS_DEP_TIME 0
DEP_TIME 107
DEP_DELAY 107
DEP_DEL15 107
CRS_ARR_TIME 0
ARR_TIME 115
ARR_DELAY 188
ARR_DEL15 188
CANCELLED 0
DIVERTED 0
CRS_ELAPSED_TIME 0
ACTUAL_ELAPSED_TIME 188
DISTANCE 0
dtype: int64

```

*The DataFrame with column 26 removed*

5. The DataFrame still contains a lot of missing values, but some of them are irrelevant because the columns containing them are not germane to the model that you are building. The goal of that model is to predict whether a flight you are considering booking is likely to arrive on time. If you know that the flight is likely to be late, you might choose to book another flight.

The next step, therefore, is to filter the dataset to eliminate columns that aren't relevant to a predictive model. For example, the aircraft's tail number probably has little bearing on whether a flight will arrive on time, and at the time you book a ticket, you have no way of knowing whether a flight will be cancelled, diverted, or delayed. By contrast, the scheduled departure time could have a *lot* to do with on-time arrivals. Because of the hub-and-spoke system used by most airlines, morning flights tend to be on time more often than afternoon or evening flights. And at some major airports, traffic stacks up during the day, increasing the likelihood that later flights will be delayed.

Pandas provides an easy way to filter out columns you don't want. Add a new cell to the notebook and execute the following code:



```
df = df[["MONTH", "DAY_OF_MONTH", "DAY_OF_WEEK", "ORIGIN", "DEST",
"CRS_DEP_TIME", "ARR_DEL15"]]
df.isnull().sum()
```

The output shows that the DataFrame now includes only the columns that are relevant to the model, and that the number of missing values is greatly reduced:

```
MONTH 0
DAY_OF_MONTH 0
DAY_OF_WEEK 0
ORIGIN 0
DEST 0
CRS_DEP_TIME 0
ARR_DEL15 188
dtype: int64
```

### *The filtered DataFrame*

- The only column that now contains missing values is the ARR\_DEL15 column, which uses 0s to identify flights that arrived on time and 1s for flights that didn't. Add another cell to the notebook and use the following code to show the first five rows with missing values:

```
df[df.isnull().values.any(axis=1)].head()
```

Pandas represents missing values with "NaN," which stands for *Not a Number*. The output shows that these rows are indeed missing values in the ARR\_DEL15 column:

|            | MONTH | DAY_OF_MONTH | DAY_OF_WEEK | ORIGIN | DEST | CRS_DEP_TIME | ARR_DEL15 |
|------------|-------|--------------|-------------|--------|------|--------------|-----------|
| <b>177</b> | 1     | 9            | 6           | MSP    | SEA  | 701          | NaN       |
| <b>179</b> | 1     | 10           | 7           | MSP    | DTW  | 1348         | NaN       |
| <b>184</b> | 1     | 10           | 7           | MSP    | DTW  | 625          | NaN       |
| <b>210</b> | 1     | 10           | 7           | DTW    | MSP  | 1200         | NaN       |
| <b>478</b> | 1     | 22           | 5           | SEA    | JFK  | 2305         | NaN       |

### *Rows with missing values*

- The reason these rows are missing ARR\_DEL15 values is that they all correspond to flights that were canceled or diverted. You could call `dropna` on the DataFrame to remove these rows. But since a flight that is canceled or diverted to another airport could be considered "late," let's use the `fillna` method to replace the missing values with 1s.

Add a new cell to the notebook and use the following code to replace missing values in the ARR\_DEL15 column with 1s and display rows 177 through 184:

```
df = df.fillna({'ARR_DEL15': 1})
df.iloc[177:185]
```

Confirm that the NaNs in rows 177, 179, and 184 were replaced with 1s indicating that the flights arrived late:

|     | MONTH | DAY_OF_MONTH | DAY_OF_WEEK | ORIGIN | DEST | CRS_DEP_TIME | ARR_DEL15 |
|-----|-------|--------------|-------------|--------|------|--------------|-----------|
| 177 | 1     | 9            | 6           | MSP    | SEA  | 701          | 1.0       |
| 178 | 1     | 9            | 6           | DTW    | JFK  | 1527         | 0.0       |
| 179 | 1     | 10           | 7           | MSP    | DTW  | 1348         | 1.0       |
| 180 | 1     | 10           | 7           | DTW    | MSP  | 1540         | 0.0       |
| 181 | 1     | 10           | 7           | JFK    | ATL  | 1325         | 0.0       |
| 182 | 1     | 10           | 7           | JFK    | ATL  | 610          | 0.0       |
| 183 | 1     | 10           | 7           | JFK    | SEA  | 1615         | 0.0       |
| 184 | 1     | 10           | 7           | MSP    | DTW  | 625          | 1.0       |

*NaNs replaced with 1s*

The dataset is now "clean" in the sense that missing values have been replaced and the list of columns has been narrowed to those most relevant to the model. But you're not finished yet. There is more to do to prepare the dataset for use in machine learning.

### Exercise 3: Bin departure times and add indicator columns

The CRS\_DEP\_TIME column of the dataset you are using represents scheduled departure times. The granularity of the numbers in this column — it contains more than 500 unique values — could have a negative impact on accuracy in a machine-learning model. This can be resolved using a technique called [binning](#) or quantization. What if you divided each number in this column by 100 and rounded down to the nearest integer? 1030 would become 10, 1925 would become 19, and so on, and you would be left with a maximum of 24 discrete values in this column. Intuitively, it makes sense, because it probably doesn't matter much whether a flight leaves at 10:30 a.m. or 10:40 a.m. It matters a great deal whether it leaves at 10:30 a.m. or 5:30 p.m.

In addition, the dataset's ORIGIN and DEST columns contain airport codes that represent categorical machine-learning values. These columns need to be converted into discrete columns containing indicator variables, sometimes known as "dummy" variables. In other words, the ORIGIN column, which contains five airport codes, needs to be converted into five columns, one per airport, with each column containing 1s and 0s indicating whether a flight originated at the airport that the column represents. The DEST column needs to be handled in a similar manner.

In this exercise, you will "bin" the departure times in the CRS\_DEP\_TIME column and use Pandas' [get dummies](#) method to create indicator columns from the ORIGIN and DEST columns.

1. Add a cell to the notebook and use the following command to display the first five rows of the DataFrame:

```
df.head()
```

Observe that the CRS\_DEP\_TIME column contains values from 0 to 2359 representing military times.

|   | MONTH | DAY_OF_MONTH | DAY_OF_WEEK | ORIGIN | DEST | CRS_DEP_TIME | ARR_DEL15 |
|---|-------|--------------|-------------|--------|------|--------------|-----------|
| 0 | 1     | 1            | 5           | ATL    | SEA  | 1905         | 0.0       |
| 1 | 1     | 1            | 5           | DTW    | MSP  | 1345         | 0.0       |
| 2 | 1     | 1            | 5           | ATL    | SEA  | 940          | 0.0       |
| 3 | 1     | 1            | 5           | SEA    | MSP  | 819          | 0.0       |
| 4 | 1     | 1            | 5           | SEA    | DTW  | 2300         | 0.0       |

*The DataFrame with unbinned departure times*

2. In a new cell, execute the following statements to bin the departure times:

```
3. import math
4.
5. for index, row in df.iterrows():
6. df.loc[index, 'CRS_DEP_TIME'] = math.floor(row['CRS_DEP_TIME'] / 100)
df.head()
```

Confirm that the numbers in the CRS\_DEP\_TIME column now fall in the range 0 to 23:

|   | MONTH | DAY_OF_MONTH | DAY_OF_WEEK | ORIGIN | DEST | CRS_DEP_TIME | ARR_DEL15 |
|---|-------|--------------|-------------|--------|------|--------------|-----------|
| 0 | 1     | 1            | 5           | ATL    | SEA  | 19           | 0.0       |
| 1 | 1     | 1            | 5           | DTW    | MSP  | 13           | 0.0       |
| 2 | 1     | 1            | 5           | ATL    | SEA  | 9            | 0.0       |
| 3 | 1     | 1            | 5           | SEA    | MSP  | 8            | 0.0       |
| 4 | 1     | 1            | 5           | SEA    | DTW  | 23           | 0.0       |

*The DataFrame with binned departure times*

7. Add a cell to the notebook and use the following statements to generate indicator columns from the ORIGIN and DEST columns, while dropping the ORIGIN and DEST columns themselves:

```
8. df = pd.get_dummies(df, columns=['ORIGIN', 'DEST'])
df.head()
```

Examine the resulting DataFrame and observe that the ORIGIN and DEST columns were replaced with columns corresponding to the airport codes present in the original columns. The new columns have 1s and 0s indicating whether a given flight originated at or was destined for the corresponding airport.

|   | MONTH | DAY_OF_MONTH | DAY_OF_WEEK | CRS_DEP_TIME | ARR_DEL15 | ORIGIN_ATL | ORIGIN_DTW | ORIGIN_JFK | ORIGIN |
|---|-------|--------------|-------------|--------------|-----------|------------|------------|------------|--------|
| 0 | 1     | 1            | 5           | 19           | 0.0       | 1          | 0          | 0          |        |
| 1 | 1     | 1            | 5           | 13           | 0.0       | 0          | 1          | 0          |        |
| 2 | 1     | 1            | 5           | 9            | 0.0       | 1          | 0          | 0          |        |
| 3 | 1     | 1            | 5           | 8            | 0.0       | 0          | 0          | 0          |        |
| 4 | 1     | 1            | 5           | 23           | 0.0       | 0          | 0          | 0          |        |

*The DataFrame with indicator columns*

9. Use the **File** -> **Save and Checkpoint** command to save the notebook.

The dataset looks very different than it did at the start, but it is now optimized for use in machine learning.

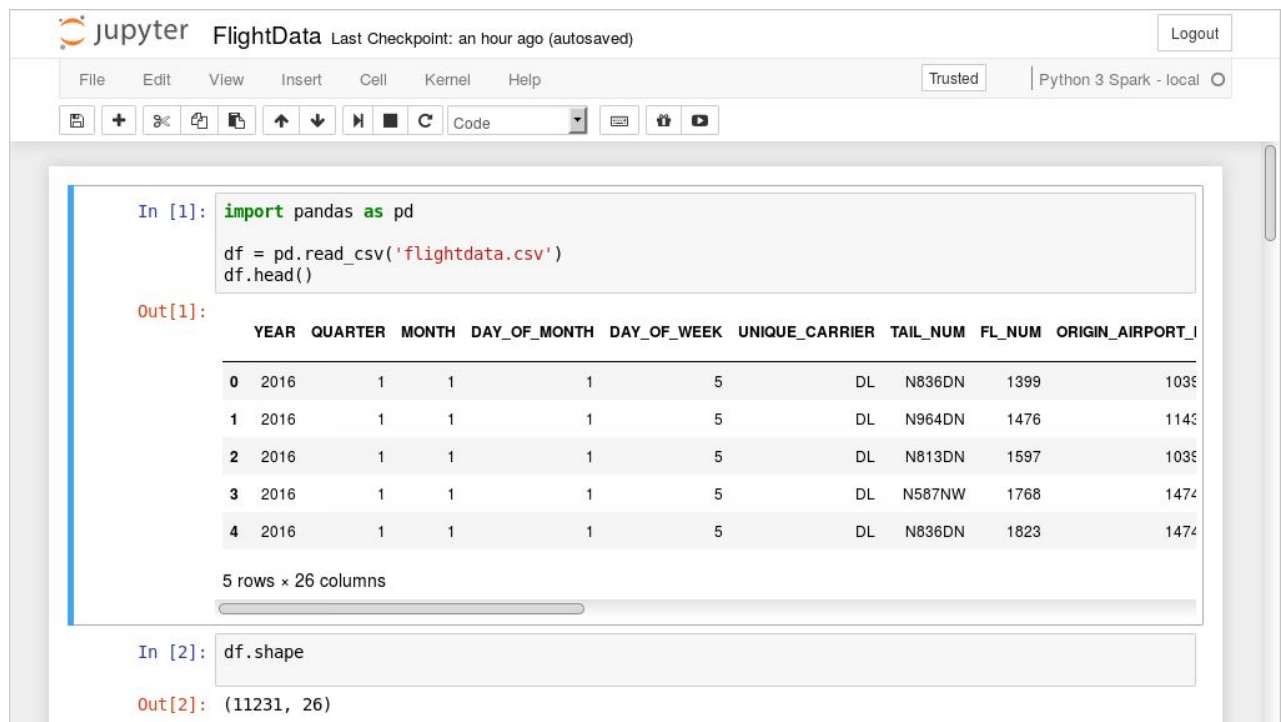
# Build

## Exercise 1: Split the data

To create a machine-learning model, you need two datasets: one for training and one for testing. In practice, you often have only one dataset, so you split it into two datasets. In this exercise, you will perform an 80-20 split on the DataFrame you prepared in the previous lab so you can use it to train a machine-learning model. You will also separate the DataFrame into feature columns and label columns. The former contains the columns used as input to the model (for example, the flight's origin and destination and the scheduled departure time), while the latter contains the column that the model will attempt to predict — in this case, the `ARR_DEL15` column which indicates whether a flight will arrive on time.

1. Return to the Data Science VM that you worked with in the previous lab. If you are not connected to it, use [X2Go](#) or the Xfce client of your choice to connect to the VM's Ubuntu desktop.
2. Open the Jupyter notebook that you worked with in the previous lab.

An easy way to reopen the notebook is to double-click the Jupyter icon on the desktop. Once Jupyter opens in a browser, click **flights** to navigate to the "flights" directory, and **FlightData.ipynb** to open the notebook.



The screenshot shows a Jupyter Notebook titled "FlightData" with a "Last Checkpoint: an hour ago (autosaved)" status. The interface includes a menu bar (File, Edit, View, Insert, Cell, Kernel, Help) and a toolbar with various icons. The notebook content is as follows:

```
In [1]: import pandas as pd
df = pd.read_csv('flightdata.csv')
df.head()
```

Out[1]:

|   | YEAR | QUARTER | MONTH | DAY_OF_MONTH | DAY_OF_WEEK | UNIQUE_CARRIER | TAIL_NUM | FL_NUM | ORIGIN_AIRPORT_I |
|---|------|---------|-------|--------------|-------------|----------------|----------|--------|------------------|
| 0 | 2016 | 1       | 1     | 1            | 5           | DL             | N836DN   | 1399   | 1035             |
| 1 | 2016 | 1       | 1     | 1            | 5           | DL             | N964DN   | 1476   | 1145             |
| 2 | 2016 | 1       | 1     | 1            | 5           | DL             | N813DN   | 1597   | 1035             |
| 3 | 2016 | 1       | 1     | 1            | 5           | DL             | N587NW   | 1768   | 1474             |
| 4 | 2016 | 1       | 1     | 1            | 5           | DL             | N836DN   | 1823   | 1474             |

5 rows x 26 columns

```
In [2]: df.shape
```

Out[2]: (11231, 26)

*The FlightData notebook*

3. Use the **Cell -> Run All** command to run the notebook. Then add a new cell to the notebook, enter the following statements, and press **Ctrl+Enter** to execute them:

4. 

```
from sklearn.model_selection import train_test_split
train_x, test_x, train_y, test_y = train_test_split(df.drop('ARR_DEL15',
axis=1), df['ARR_DEL15'], test_size=0.2, random_state=42)
```

The first statement imports Scikit-learn's [train test split](#) helper function. The second line uses the function to split the DataFrame into a training set containing 80% of the original data, and a test set containing the remaining 20%. The `random_state` parameter seeds the random-number generator used to do the splitting, while the first and second parameters are DataFrames containing the feature columns and the label column.

5. `train_test_split` returns four DataFrames. Use the following command to display the number of rows and columns in the DataFrame containing the feature columns used for training:

```
train_x.shape
```

6. Now use this command to display the number of rows and columns in the DataFrame containing the feature columns used for testing:

```
test_x.shape
```

How do the two outputs differ, and why?

Can you predict what you would see if you called `shape` on the other two DataFrames, `train_y` and `test_y`? If you're not sure, try it and find out.

## Exercise 2: Train a machine-learning model

There are many types of machine-learning models. One of the most common is the regression model, which uses one of a number of regression algorithms to produce a numeric value — for example, a person's age or the probability that a credit-card transaction is fraudulent. You will train a classification model, which seeks to resolve a set of inputs into one of a set of known outputs. A classic example of a classification model is one that examines e-mails and classifies them as "spam" or "not spam." Your model will be a binary classification model that predicts whether a flight will arrive on-time or late ("binary" because there are only two possible outputs).

One of the benefits of using Scikit-learn is that you don't have to build these models — or implement the algorithms that they use — by hand. Scikit-learn includes a variety of classes for implementing common machine-learning models. One of them is [RandomForestClassifier](#), which fits multiple decision trees to the data and uses averaging to boost the overall accuracy and control [overfitting](#).

1. Add a cell to the notebook. Use the following code to create a RandomForestClassifier object and train it by calling the [fit](#) method.

```
2. from sklearn.ensemble import RandomForestClassifier
3.
4. model = RandomForestClassifier(random_state=13)
 model.fit(train_x, train_y)
```

The output shows the parameters used in the classifier, including `n_estimators`, which specifies the number of trees in each decision-tree forest, and `max_depth`, which specifies the maximum depth of the decision trees. The values shown are the defaults, but you can override any of them when creating the RandomForestClassifier object.

```
In [16]: from sklearn.ensemble import RandomForestClassifier

 model = RandomForestClassifier(random_state=13)
 model.fit(train_x, train_y)

Out[16]: RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
 max_depth=None, max_features='auto', max_leaf_nodes=None,
 min_impurity_split=1e-07, min_samples_leaf=1,
 min_samples_split=2, min_weight_fraction_leaf=0.0,
 n_estimators=10, n_jobs=1, oob_score=False, random_state=13,
 verbose=0, warm_start=False)
```

### *Training the model*

5. Now call the [predict](#) method to test the model using the values in `test_x`, followed by the [score](#) method to determine the mean accuracy of the model:
6. `predicted = model.predict(test_x)`  
`model.score(test_x, test_y)`

Confirm that you see the following output:

```
In [17]: predicted = model.predict(test_x)
 model.score(test_x, test_y)

Out[17]: 0.86025812194036488
```

### *Testing the model*

The mean accuracy is 86%, which seems good on the surface. However, mean accuracy isn't always a reliable indicator of the accuracy of a classification model. Let's dig a little deeper and determine how accurate the model really is — that is, how adept it is at determining whether a flight will arrive on time.

## Exercise 3: Gauge the model's accuracy

There are several ways to measure the accuracy of a classification model. One of the best overall measures for a binary classification model is [Area Under Receiver Operating Characteristic Curve](#) (sometimes referred to as "ROC AUC"), which



essentially quantifies how often the model will make a correct prediction regardless of the outcome. In this exercise, you will compute an ROC AUC score for the model you built in the previous exercise and learn about some of the reasons why that score is lower than the mean accuracy output by the `score` method. You will also learn about other ways to gauge the accuracy of the model.

1. Before you compute the ROC AUC, you must generate *prediction probabilities* for the test set. These probabilities are estimates for each of the classes, or answers, the model can predict. For example, `[0.88199435, 0.11800565]` means that there's an 89% chance that a flight will arrive on time (`ARR_DEL15 = 0`) and a 12% chance that it won't (`ARR_DEL15 = 1`). The sum of the two probabilities adds up to 100%.

Add a cell to the notebook and use the following code to generate a set of prediction probabilities from the test data:

```
from sklearn.metrics import roc_auc_score
probabilities = model.predict_proba(test_x)
```

2. Now use the following statement to generate an ROC AUC score from the probabilities using Sckit-learn's [roc\\_auc\\_score](#) method:

```
roc_auc_score(test_y, probabilities[:, 1])
```

Confirm that the output shows a score of 67%:

```
In [19]: roc_auc_score(test_y, probabilities[:, 1])
```

```
Out[19]: 0.67438249049985388
```

### Generating an AUC score

Why is the AUC score lower than the mean accuracy computed in the previous exercise?

The output from the `score` method reflects how many of the items in the test set the model predicted correctly. This score is skewed by the fact that the dataset the model was trained and tested with contains many more rows representing on-time arrivals than rows representing late arrivals. Because of this imbalance in the data, you are more likely to be correct if you predict that a flight will be on time than if you predict that a flight will be late.

ROC AUC takes this into account and provides a more accurate indication of how likely it is that a prediction of on-time or late will be correct.

3. You can learn more about the model's behavior by generating a [confusion matrix](#), also known as an *error matrix*. The confusion matrix quantifies the



number of times each answer was classified correctly or incorrectly. Specifically, it quantifies the number of false positives, false negatives, true positives, and true negatives. This is important, because if a binary classification model trained to recognize cats and dogs is tested with a dataset that is 95% dogs, it could score 95% simply by guessing "dog" every time. But if it failed to identify cats at all, it would be of little value.

Use the following code to produce a confusion matrix for your model:

```
from sklearn.metrics import confusion_matrix
confusion_matrix(test_y, predicted)
```

The first row in the output represents flights that were on time. The first column in that row shows how many flights were correctly predicted to be on time, while the second column reveals how many flights were predicted as delayed but were not. From this, the model appears to be very adept at predicting that a flight will be on time.

```
In [20]: from sklearn.metrics import confusion_matrix
 confusion_matrix(test_y, predicted)

Out[20]: array([[1882, 54],
 [260, 51]])
```

### *Generating a confusion matrix*

But look at the second row, which represents flights that were delayed. The first column shows how many delayed flights were incorrectly predicted to be on time. The second column shows how many flights were correctly predicted to be delayed. Clearly, the model isn't nearly as adept at predicting that a flight will be delayed as it is at predicting that a flight will arrive on time. What you *want* in a confusion matrix is big numbers in the upper-left and lower-right corners, and small numbers (preferably zeros) in the upper-right and lower-left corners.

4. Other measures of accuracy for a classification model include *precision* and *recall*. Suppose the model was presented with three on-time arrivals and three delayed arrivals, and that it correctly predicted two of the on-time arrivals, but incorrectly predicted that two of the delayed arrivals would be on time. In this case, the precision would be 50% (two of the four flights it classified as being on time actually were on time), while its recall would be 67% (it correctly identified two of the three on-time arrivals). You can learn more about precision and recall from [https://en.wikipedia.org/wiki/Precision\\_and\\_recall](https://en.wikipedia.org/wiki/Precision_and_recall)

Sckit-learn contains a handy method named [precision\\_score](#) for computing precision. To quantify the precision of your model, execute the following statements:

```
from sklearn.metrics import precision_score

train_predictions = model.predict(train_x)
precision_score(train_y, train_predictions)
```

Examine the output. What is your model's precision?

```
In [21]: from sklearn.metrics import precision_score
 train_predictions = model.predict(train_x)
 precision_score(train_y, train_predictions)

Out[21]: 0.8875000000000001
```

### *Measuring precision*

5. Sckit-learn also contains a method named [recall\\_score](#) for computing recall. To measure your model's recall, execute the following statements:

```
6. from sklearn.metrics import recall_score
 recall_score(train_y, train_predictions)
```

What is the model's recall?

```
In [22]: from sklearn.metrics import recall_score
 recall_score(train_y, train_predictions)

Out[22]: 0.8846153846153846
```

### *Measuring recall*

7. Use the **File** -> **Save and Checkpoint** command to save the notebook.

In the real world, a trained data scientist would look for ways to make the model even more accurate. Among other things, he or she would try different algorithms and take steps to *tune* the chosen algorithm to find the optimum combination of parameters. Another likely step would be to expand the dataset to millions of rows rather than a few thousand and also attempt to reduce the imbalance between late and on-time arrivals. But for our purposes, the model is fine as is.

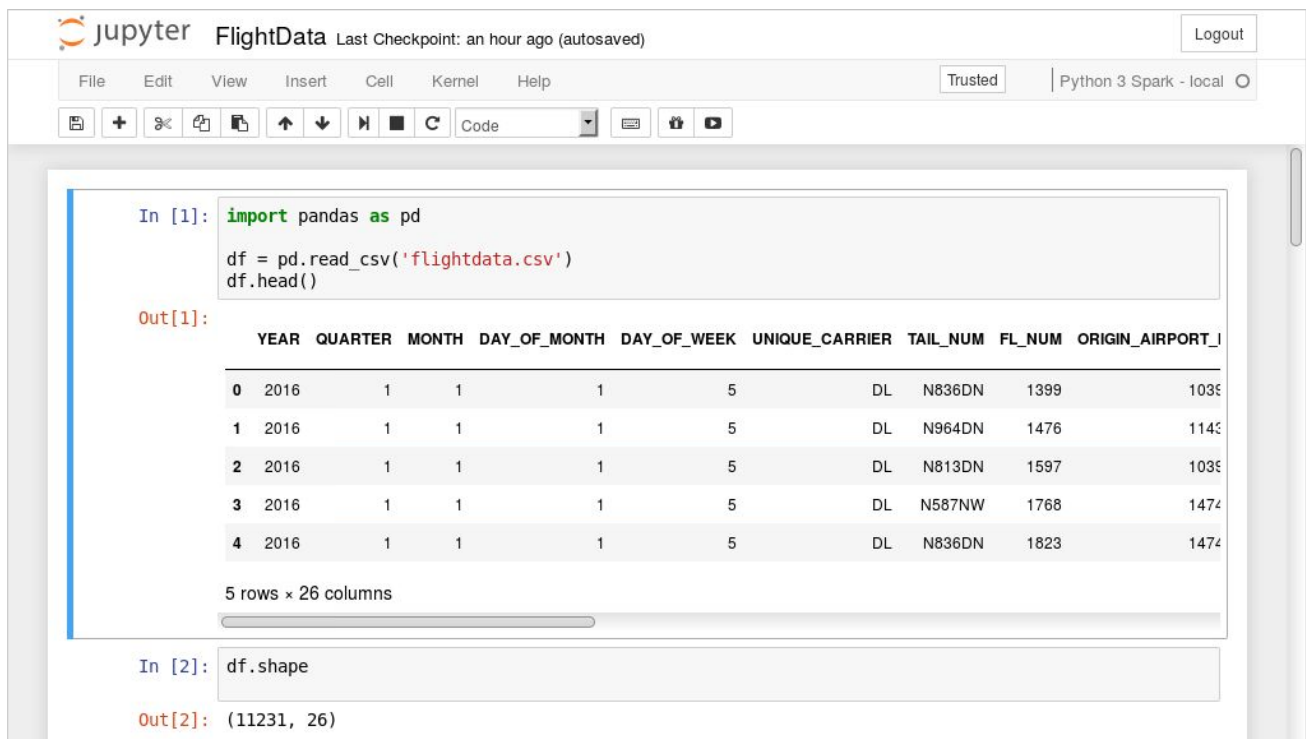
# Visualise

## Exercise 1: Import Matplotlib

In this exercise, you will import Matplotlib into the Jupyter notebook you have been working with and configure the notebook to support inline Matplotlib output.

1. Return to the Data Science VM that you worked with in the previous lab. If you are not connected to it, use [X2Go](#) or the Xfce client of your choice to connect to the VM's Ubuntu desktop.
2. Open the Jupyter notebook that you worked with in the previous lab.

An easy way to reopen the notebook is to double-click the Jupyter icon on the desktop. Once Jupyter opens in a browser, click **flights** to navigate to the "flights" directory, and **FlightData.ipynb** to open the notebook.



The screenshot shows a Jupyter Notebook titled "FlightData" with a "Trusted" security status and "Python 3 Spark - local" kernel. The first code cell imports pandas and reads a CSV file, displaying the first five rows of the data. The second code cell prints the shape of the DataFrame.

```
In [1]: import pandas as pd
df = pd.read_csv('flightdata.csv')
df.head()

Out[1]:
```

|   | YEAR | QUARTER | MONTH | DAY_OF_MONTH | DAY_OF_WEEK | UNIQUE_CARRIER | TAIL_NUM | FL_NUM | ORIGIN_AIRPORT_I |
|---|------|---------|-------|--------------|-------------|----------------|----------|--------|------------------|
| 0 | 2016 | 1       | 1     | 1            | 5           | DL             | N836DN   | 1399   | 1039             |
| 1 | 2016 | 1       | 1     | 1            | 5           | DL             | N964DN   | 1476   | 1143             |
| 2 | 2016 | 1       | 1     | 1            | 5           | DL             | N813DN   | 1597   | 1039             |
| 3 | 2016 | 1       | 1     | 1            | 5           | DL             | N587NW   | 1768   | 1474             |
| 4 | 2016 | 1       | 1     | 1            | 5           | DL             | N836DN   | 1823   | 1474             |

5 rows x 26 columns

```
In [2]: df.shape

Out[2]: (11231, 26)
```

### *The FlightData notebook*

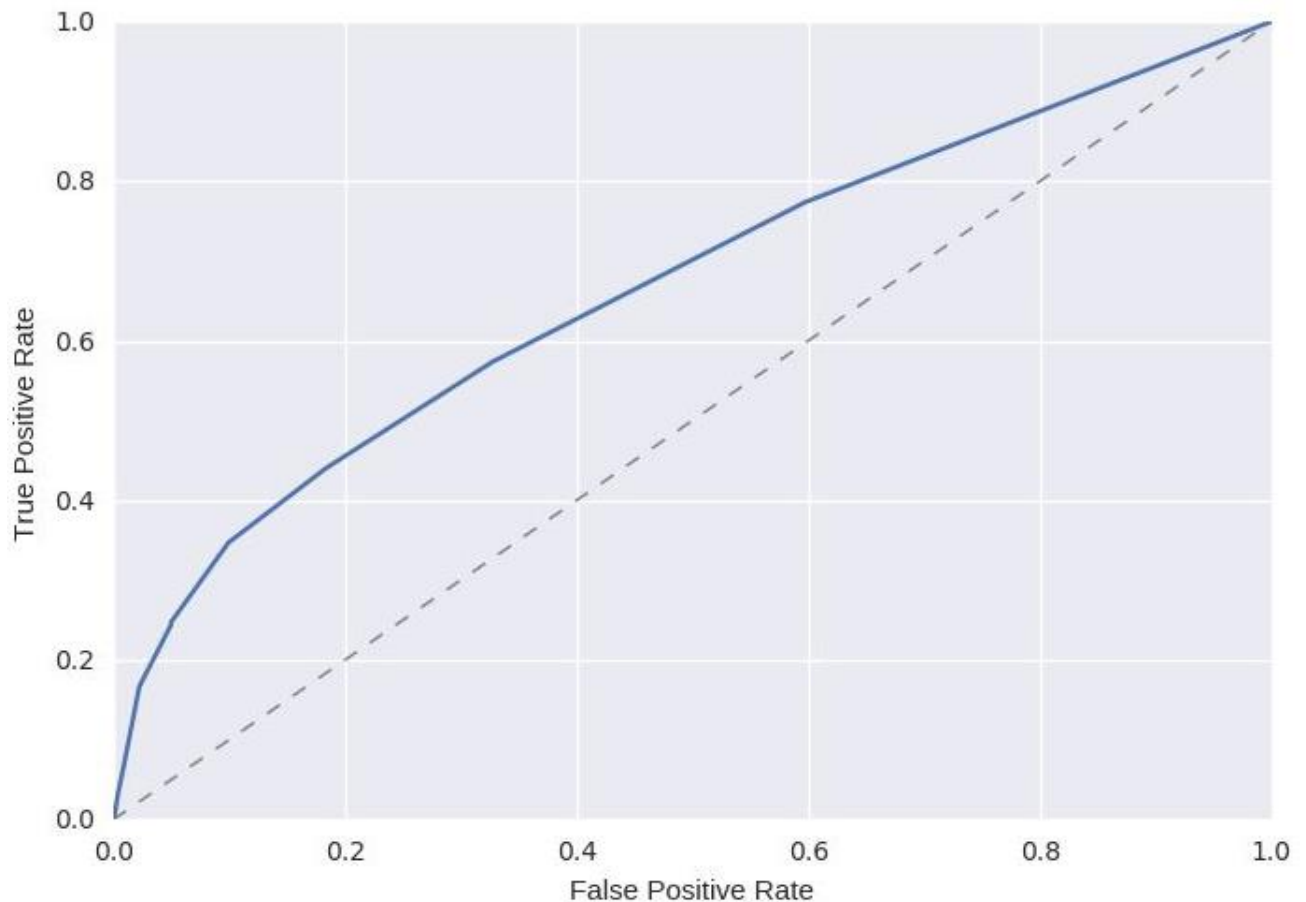
3. Use the **Cell -> Run All** command to run the notebook. Then add a cell to the notebook and execute the following statements. Ignore any warning messages that are displayed related to font caching:
4. `%matplotlib inline`
5. `import matplotlib.pyplot as plt`  
`import seaborn as sns`

The first statement is one of several [magic commands](#) supported by the Python kernel that you selected when you created the notebook. It enables Jupyter to render Matplotlib output in a notebook without making repeated calls to [show](#). And it must appear before any references to Matplotlib itself.

6. To see Matplotlib at work, execute the following code in a new cell to plot the [ROC curve](#) for the machine-learning model you built in the previous lab:

```
7. from sklearn.metrics import roc_curve
8.
9. fpr, tpr, _ = roc_curve(test_y, probabilities[:, 1])
10. plt.plot(fpr, tpr)
11. plt.plot([0, 1], [0, 1], color='grey', lw=1, linestyle='--')
12. plt.xlabel('False Positive Rate')
 plt.ylabel('True Positive Rate')
```

13. Confirm that you see the following output:



*ROC curve generated with Matplotlib*

The dotted line in the middle of the graph represents a 50-50 chance of obtaining a correct answer. The blue curve represents the accuracy of your model. More

importantly, the fact that this chart appears at all demonstrates that you can use Matplotlib in a Jupyter notebook.

## Exercise 2: Predict on-time arrivals

The reason you built a machine-learning model is to predict whether a flight will arrive on time or late. In this exercise, you will write a Python function that calls the machine-learning model you built in the previous lab to compute the likelihood that a flight will be on time. Then you will use the function to analyze several flights.

1. Add a cell to the notebook and enter the following function definition. Then press **Ctrl+Enter** to execute the cell.

```
2. def predict_delay(departure_date_time, origin, destination):
3. from datetime import datetime
4.
5. try:
6. departure_date_time_parsed = datetime.strptime(departure_date_time,
7. '%d/%m/%Y %H:%M:%S')
8. except ValueError as e:
9. return 'Error parsing date/time - {}'.format(e)
10.
11. month = departure_date_time_parsed.month
12. day = departure_date_time_parsed.day
13. day_of_week = departure_date_time_parsed.isoweekday()
14. hour = departure_date_time_parsed.hour
15.
16. origin = origin.upper()
17. destination = destination.upper()
18.
19. input = [{'MONTH': month,
20. 'DAY': day,
21. 'DAY_OF_WEEK': day_of_week,
22. 'CRS_DEP_TIME': hour,
23. 'ORIGIN_ATL': 1 if origin == 'ATL' else 0,
24. 'ORIGIN_DTW': 1 if origin == 'DTW' else 0,
25. 'ORIGIN_JFK': 1 if origin == 'JFK' else 0,
26. 'ORIGIN_MSP': 1 if origin == 'MSP' else 0,
27. 'ORIGIN_SEA': 1 if origin == 'SEA' else 0,
28. 'DEST_ATL': 1 if destination == 'ATL' else 0,
29. 'DEST_DTW': 1 if destination == 'DTW' else 0,
30. 'DEST_JFK': 1 if destination == 'JFK' else 0,
31. 'DEST_MSP': 1 if destination == 'MSP' else 0,
32. 'DEST_SEA': 1 if destination == 'SEA' else 0}]
33.
34. return model.predict_proba(pd.DataFrame(input))[0][0]
```

This function takes as input a date and time, an origin airport code, and a destination airport code, and returns a value between 0.0 and 1.0 indicating the probability that the flight will arrive at its destination on time. It uses the machine-learning model you built in the previous lab to compute the probability. And to call the model, it passes a DataFrame containing the input

values to `predict_proba`. The structure of the DataFrame exactly matches the structure of the DataFrame depicted in Exercise 3, Step 3 of [Lab 2](#). Note that dates input to the `predict_delay` function use the international date format `dd/mm/year`.

33. Add a cell to the notebook and use the code below to compute the probability that a flight from New York to Atlanta on the evening of October 1 will arrive on time. Note that the year you enter is irrelevant because it isn't used by the model.

```
predict_delay('1/10/2018 21:45:00', 'JFK', 'ATL')
```

Confirm that the output shows that the likelihood of an on-time arrival is 60%:

```
In [26]: predict_delay('1/10/2018 21:45:00', 'JFK', 'ATL')
```

```
Out[26]: 0.5999999999999998
```

*Predicting whether a flight will arrive on time*

34. Modify the code to compute the probability that the same flight a day later will arrive on time:

```
predict_delay('2/10/2018 21:45:00', 'JFK', 'ATL')
```

How likely is this flight to arrive on time? If your travel plans were flexible, would you consider postponing your trip for one day?

35. Now modify the code to compute the probability that a morning flight the same day from Atlanta to Seattle will arrive on time:

```
predict_delay('2/10/2018 10:00:00', 'ATL', 'SEA')
```

Is this flight likely to arrive on time?

You now have an easy way to predict, with a single line of code, whether a flight is likely to be on time or late. Feel free to experiment with other dates, times, origins, and destinations. But keep in mind that the results are only meaningful for the airport codes ATL, DTW, JFK, MSP, and SEA because those are the only airport codes the model was trained with.

### Exercise 3: Plot predictions

---

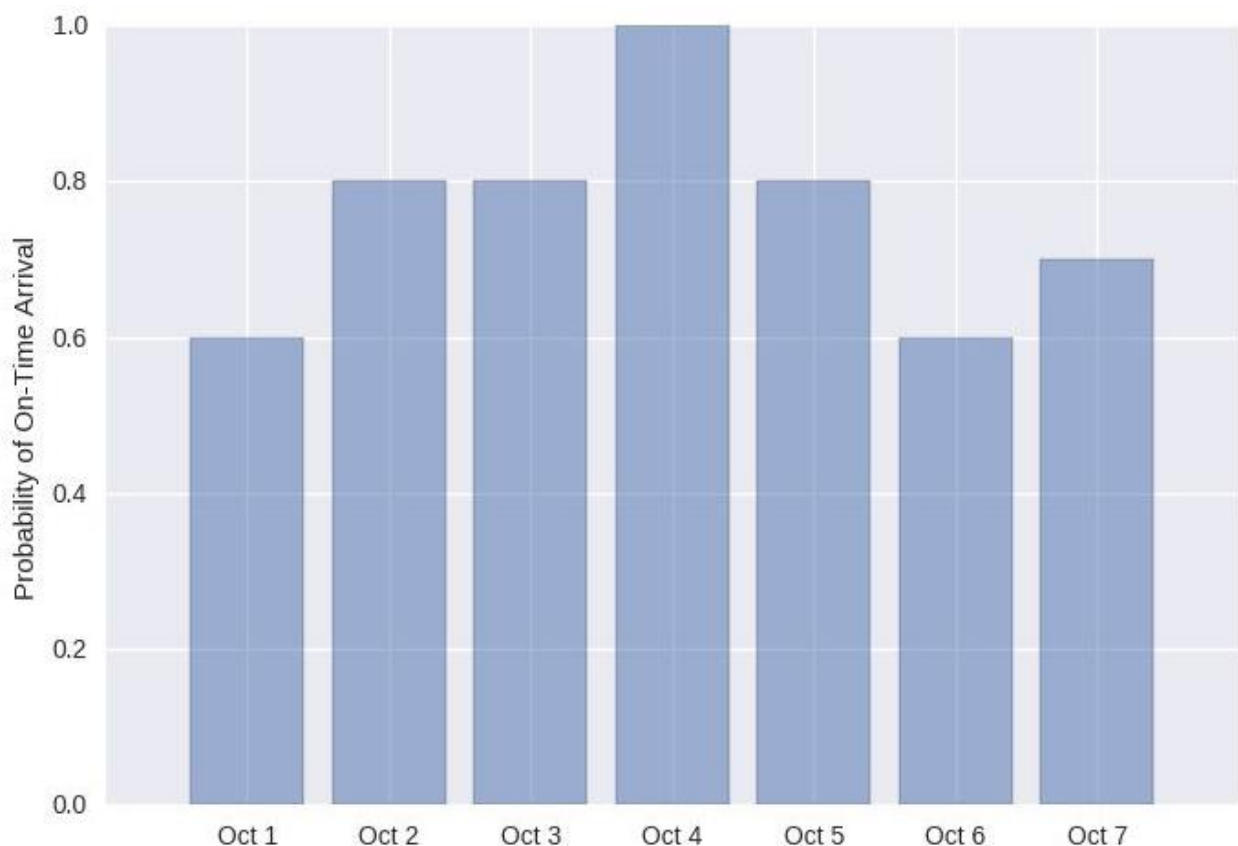
In this exercise, you will combine the `predict_delay` function you created in the previous exercise with Matplotlib to produce side-by-side comparisons of the same

flight on consecutive days and flights with the same origin and destination at different times throughout the day.

1. Add a new cell to the notebook and execute the following code to plot the probability of on-time arrivals for an evening flight from JFK to ATL over a range of days:

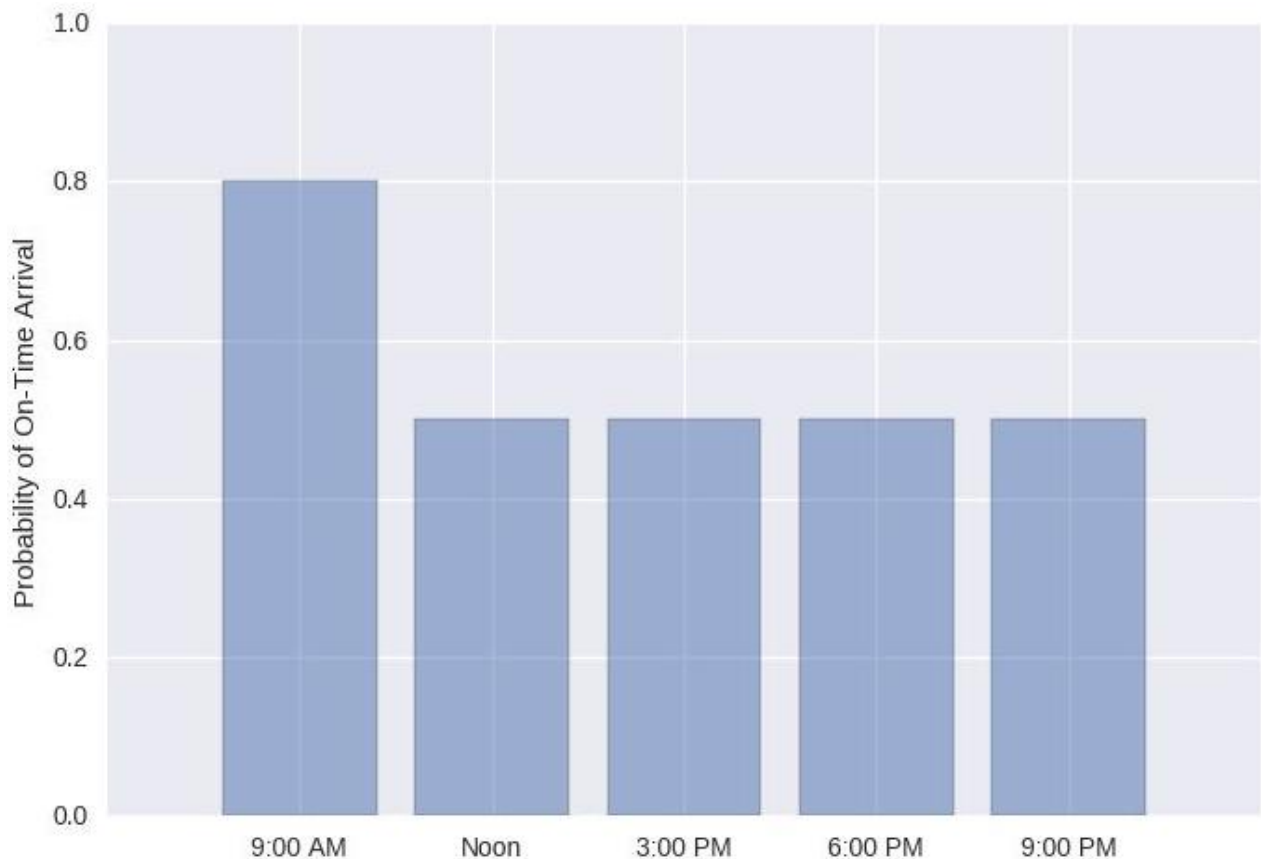
```
2. import numpy as np
3.
4. labels = ('Oct 1', 'Oct 2', 'Oct 3', 'Oct 4', 'Oct 5', 'Oct 6', 'Oct 7')
5. values = (predict_delay('1/10/2018 21:45:00', 'JFK', 'ATL'),
6. predict_delay('2/10/2018 21:45:00', 'JFK', 'ATL'),
7. predict_delay('3/10/2018 21:45:00', 'JFK', 'ATL'),
8. predict_delay('4/10/2018 21:45:00', 'JFK', 'ATL'),
9. predict_delay('5/10/2018 21:45:00', 'JFK', 'ATL'),
10. predict_delay('6/10/2018 21:45:00', 'JFK', 'ATL'),
11. predict_delay('7/10/2018 21:45:00', 'JFK', 'ATL'))
12. alabels = np.arange(len(labels))
13.
14. plt.bar(alabels, values, align='center', alpha=0.5)
15. plt.xticks(alabels, labels)
16. plt.ylabel('Probability of On-Time Arrival')
 plt.ylim((0.0, 1.0))
```

17. Confirm that the output looks like this:



*Probability of on-time arrivals for a range of dates*

18. Modify the code to produce a similar chart for flights leaving JFK for MSP at 1:00 p.m. on April 10 through April 16. How does the output compare to the output in the previous step?
19. On your own, write code to graph the probability that flights leaving SEA for ATL at 9:00 a.m., noon, 3:00 p.m., 6:00 p.m., and 9:00 p.m. on January 30 will arrive on time. Confirm that the output matches this:



*Probability of on-time arrivals for a range of times*

If you are new to Matplotlib and would like to learn more about it, you will find an excellent tutorial at <https://www.labri.fr/perso/nrougier/teaching/matplotlib/>. There is *much* more to Matplotlib than what was shown here, which is one reason why it is so popular in the Python community.

## Summary

---

In four hands-on labs, you learned how to:

- Import data into the VM using `curl`
- Create a Jupyter notebook in the VM
- Use [Pandas](#) to clean and prepare data



- Use [Scikit-learn](#) to build a machine-learning model
- Use [Matplotlib](#) to visualize the results

Pandas, Scikit-learn, and Matplotlib are three of the most popular Python libraries on the planet. With them, you can prepare data for use in machine learning, build sophisticated machine-learning models from the data, and chart the output. They are among dozens of tools preinstalled in Microsoft's Data Science VM, and they are just the tip of the iceberg in terms of what you can do with it. For more information, see <https://docs.microsoft.com/azure/machine-learning/data-science-virtual-machine/dsvm-tools-overview>.