Machine Learning in Python

Contents

1 Six Step Machine Learning Framework

There are six steps that we can take during the iterative process that cover the idea of *Data Modeling*:

1) Problem Definition - Is it Supervised or Unsupervised? Classification or Regression?
2) Data - What kind of data? Structured (csv/spreadsheets) or Unstructured (images/audio)?
3) Evaluation - How accurate do we need the model to be?
4) Features - What variables of the data can be used to help our model?
5) Modeling - Determine the right model based on the problem defined earlier.
6) Experimentation - How can we improve the model? What can we try next?

1.1 Step 1: Problem Definition

In this step, we ask ourselves "what problem are we trying to solve?". However, machine learning isn't always the optimal solution. Sometimes we must consider if a simple hand-coded instruction based system will solve our problem faster. The first task in this step is to match our problem to one of the main types of problems, which include the following.

Supervised Learning uses data and labels in a machine learning algorithm in order to predict an answer. The main types of supervised learning are:

- **Classification** : Determine which class a data point belongs two. Can be both binary (two options) or multi-class (more than two).
- **Regression** : Used to predict a *continuous* number.

Unsupervised Learning has data but no labels. It uses this data to find patterns and classify them into groups (clusters) than can be used to find an answer to the proposed questions. This method is called *Clustering*, putting groups of data together based on similarities, which can be used for things such as recommendation engines.

Transfer Learning takes machine learning models that already exists and fine tunes it in order to address the problem at hand.

Reinforcement Learning rewards the model for doing well or punishes it for doing poorly. For example, keeping a score and rewarding +1 when the model is correct and punishing with -1 when the model is wrong will teach the model to make choices that will maximize this 'score'.

1.2 Step 2: Data

Data comes mainly in two different types: structured and unstructured. **Structured data** contains rows and columns with all of the samples in the columns being in a similar format. **Unstructured data** could be things such as images, videos, or audio files.

Within these two data types, there is *static* and *streaming* data types. **Static data** is data that does not change over time (often in csv files) which can be all contained in one file separated by commas. **Streaming data** is data that is constantly changed over time (ex: news headlines effect on stock prices).

A common data science workflow consists of: reading in static data from a csv file into a Jupyter notebook, exploring data and performing data analysis with pandas, using matplotlib to visualize any patterns/trends in the data, and finally using a machine learning model from scikit to predict using patterns.

1.3 Step 3: Evaluation

All models are used to find insights about the future, but an **evaluation metric** is a measure of how well an algorithm is at predicting the future. In this step, we want to find what percent accuracy defines success for our model. For example, if we are classifying medical records to determine heart disease diagnosis we may want our model to be over 99% accurate.

We use different **metrics** in order to classify the accuracy of our model based on its type:
- Classification : We use accuracy, precision, and recall.
- Regression : Mean absolute error (MAE), mean square error (MSE), root mean squared error (RMSE).
- Recommendation : We use precision at K.

1.4 Step 4: Features

Features is another word for different forms of data. This could be things such as columns (feature variables) that are used to predict a new feature (target variable).

Features can be both **numerical** (number value) and **categorical** (string, Boolean, etc.), while the user can create a new feature, called a **derived feature** by using existing ones.

An important note is that features work best within a machine learning algorithm if many of the samples have it (not many missing values in the column). **Feature coverage** is how many samples have different features (ideally we want every sample to have the same features with minimal missing values).

1.5 Step 5: Modeling

Modeling can be split into 3 different parts: choosing/training a model, tuning a model, and model comparison. However, before we do this, we must **split the data** into 3 different sets to use in each of these parts:
- Training set : Part 1, data used to train the model (70% - 80%).
- Validation set : Part 2, data used to tune/improve the model (10% - 15%).
- Test set : Part 3, data used to test the accuracy of our model (10% - 15%).

We feed our model the training set to set up a baseline, then we input the validation set to see if we can improve the model (*model tuning*), and finally we check the results with the test set (validate our model is accurate). Be sure that the model never sees the validation/test split while training because this will throw off the true accuracy of the model.

Generalization is the ability for a machine learning model to perform well on data it hasn't seen before. However, we must be cautious not to let the validation set be the same as the test set (this can lead to memorization and no real learning for the model).

In part 1 of modeling, we will **chose the model** to use for the current problem. When working with *structured data*, decision trees like Random Forests and Gradient Boost are the best choices. When working with *unstructured data* we typically chose Deep Learning, Neural Networks, or Transfer Learning.

After choosing a model, we want to begin to train it (the main goal is to line up the inputs and outputs). We do this by using feature variables (x) to predict target variables (y). Another goal is to minimize time between experiments (one possible way is to use a small amount of the training set to begin with if it is a large dataset). We must consider if an increase in accuracy is worth the increase in time to run.

In part 2 of modeling, we will **tune the model** to try and improve our model. Using the validation set, we adjust *hyperparameter* based on the model we chose. For example, increasing trees in a Random Forest or the number of layers in a Neural Network.

In part 3 of modeling, we **compare models** using the test data to see how it works in real world applications. A good model will yield similar results on all 3 sets of data (possibly a slight decline from training to test set). If the training set performance is a much higher percent than the test set, then we have **underfitting**. If the test set performance is higher than the training set performance, then we have **overfitting**.

One of the main causes for overfitting is *data leakage*, which happens when some of your test data leaks into the training data. For example, this is similar to seeing the final exam before taking it (memorization). We must make sure to properly use the correct data set for the modeling step.

One of the main causes for underfitting is *data mismatch*, which happens when the data your testing on is different than the data you are testing on. This could be caused by having different features in both of the sets.

Some fixes for underfitting include: more advanced models, increase hyperparameters, reduce amount of features, or longer training. Some fixes for overfitting include: collecting more data or trying a less advanced models.

2 Pandas: Data Analysis

2.1 Series, Data Frames, and CSVs

Series are 1-D objects that are created by passing a list into a pd.Series( ) method. **Data Frames** are 2-D objects that are created with multiple series or reading in a csv file, with the pd.DataFrame( ) or pd.read_csv( ) method. Note that the column will be referred to with *axis=1* and the row is referred to by *axis=0*. Note that you can read in csv files from a URL as long as it is in a 'raw' format.

```
import pandas as pd

cars = pd.Series(['BMW', 'Toyota', 'Honda'])
colors = pd.Series(['Red', 'Blue', 'White'])

car_data = pd.DataFrame({'Car Make': cars, 'Color': colors})

car_sales = pd.read_csv('car-sales.csv')
```

After altering an imported Data Frame or creating a new one, we can **export** it as a csv file. The file will save in the current directory (folder) that the project is in. Note that you must pass the parameter *index=False* so that it does not export the index as a column (causes duplicate index columns).

```
car_sales.to_csv('exported-car-sales.csv', index=False)
```

2.2 Describing Data

We can use the attribute *.dtypes* to get information about the data types for the data frames columns. We can get a list of the column names for our data frame by using the *.columns* attribute.

We can use the .describe() method to get statistical information about the numeric columns. We can use the .info() method to get a general overview of the data types, columns, index range, etc. (combination of the above attributes).

2.3 Selecting/Viewing Data

We can use the **.loc**[ ] method to access rows based on index number. Note that if there are multiple rows with the same index number, it will return all of the values with that index number.

We can use the **.iloc**[ ] method to access rows based on index position. This will always only return a single row.

Both .loc and .iloc allow us to us slicing when accessing rows. However, .loc will gives us up to and including the last specified row while .iloc will gives us all up to but *not* including the last specified row.

```
car_sales.loc[:3] # gives us rows 0, 1, 2, 3
car_sales.iloc[:3] # gives us rows 0, 1, 2 but not 3
```

We can filter results by using **Boolean indexing** when trying to find specific column values.

```
car_sales[car_sales['Make'] == 'Toyota']
```

We can cross over two columns to get an aggregate table by using .crosstab( ) with the first parameter being the rows and the second parameter being the columns. Similarly, if we want to **compare multiple columns**, we can use the .groupby( ) method. In the below example, we can group by the 'Make' column and take the mean for each numeric varaible column.

```
pd.crosstab(car_sales['Make'], car_sales['Doors'])

car_sales.groupby(['Make']).mean()
```

Pandas allows us to plot numeric columns with built in methods as well. We can use *.plot()* on a column in our data frame to see a line graph of the values. We can use the *.hist()* method to see a histogram plot of our column values, which shows the frequency of points on our data frame column.

2.4 Manipulating Data

Python has built in **string methods** that lets use access and manipulate string type columns in Pandas by using the *.str* accessor. Note that in order to save any changes, you have to reassign the column to the original data frame.

```
car_sales['Make'] = car_sales['Make'].str.lower()
```

Missing data (NaN values) can be common with data frames. We can **fill this missing values** with the .fillna( ) method and passing in a value we want to insert into all NaN values in a column. Note that we can change values in a column without having to reassign it by passing the *inplace=True* parameter if the method allows us to.

```
car_sales_missing = pd.read_csv('car_sales_missing_data.csv')

car_sales_missing['Odometer'].fillna(car_sales_missing['Odometer'].mean(),
                                     inplace=True)
```

We can **drop all rows with missing values** by using the .dropna( ) method. Note that we can either use reassignment or *inplace=True* when using this method.

```
car_sales_missing.dropna(inplace=True)
```

We can **delete columns** by using the .drop( ) method and specify the column name, axis=1, and using inplace=True. Note that this method can also be used for rows if axis=0 is specified along with the row you want to remove from the data frame.

```
car_sales_missing.drop('Total Fuel Used', inplace=True)
```

When splitting the data into training, validation, and test sets we often want to randomize the order so that the model does not memorize inputs. One way we can **shuffle data** is by using the Pandas .sample( ) method and passing the *frac=1* parameter to change the original order. Note that we data frames become larger, we can use smaller sets by changing *frac* to a smaller value (between 0 and 1).

```
car_sales_shuffled = car_sales.sample(frac=1)
```

If we want to **reset the index** after shuffling data, we can use the .reset_index( ) method and passing the *drop=True* parameter so that it does not make a new column with the previous index values.

```
car_sales_shuffled.reset_index(drop=True)
```

We can **apply functions** (NumPy or lambda) to columns with the .apply( ) method.

```
car_sales['Odometer (KM)'] = car_sales['Odometer (KM)'].apply(lambda x : x / 1.6)
```

3 NumPy

3.1 Arrays and Attributes

NumPy's main **datatype** is an ndarray (n-dimensional array). We can also **create a data frame** from a NumPy array.

```
import numpy as np

a1 = np.array([1, 2, 3])
a1.shape # returns (3,)

a2 = np.array([[1, 2, 3],
               [4, 5, 6]])
a2.ndim # returns 2

df = pd.DataFrame(a2)
```

We can use built in functions to **create arrays** filled with specified values, random values, or filled with one given value.

```
ones = np.ones((2,3)) # create a 2x3 array of 1's

zeros = np.zeros((2,3)) # create a 2x3 array of 0's

range_arr = np.arange(0, 10, 2) # create a 1D array from 0 to 10 (inc by 2)

random_arr = np.random.randint(0, 10, size=(3,5))
# create 3x5 array of random ints between 0 and 10

random_arr_2 = np.random.random((5,3)) # create a 5x3 array of values between 0 and 1

Random_arr_3 = np.random.rand(5,3) # create 5X3 array of random values
```

When creating a random array, the values will always be different if you run the code cell again. However, we can use the **random seed** function to always create the same numbers for a given random array. This allows us to create reproducible results as well. Note that this can be any number (it doesn't matter) but you will have to run the seed call in every cell you want reproducible results in.

```
np.random.seed(6)
```

3.2 Array Functions

We can perform **aggregation** on our NumPy arrays. We want to be sure that we use Python methods on Python datatypes but use NumPy's methods on NumPy arrays. For example, a list could use sum() but a NumPy array should use np.sum(). This is to optimize speed while doing calculations since NumPy uses broadcasting and C for array operations.

```
large_arr = np.random.random(100000)

%timeit sum(large_arr) # approx 19.9 ms per loop (526 times slower than np method)
%timeit np.sum(large_arr) # approx 34 us (micro) per loop
```

We often discuss the spread of the data we are working with. **Variance** is the measure of the average degree to which each number is different from the mean. Note that higher variance means a wider range of numbers while a lower variance is a smaller range of numbers. **Standard Deviation** is a measure of how spread out a group of numbers is from the mean (this number is just the square root of the variance).

```
high_var = np.array([1, 100, 200, 300, 4000, 5000])
low_var = np.array([2, 4, 6, 8, 10])

np.var(high_var), np.var(low_var) # (4296133.47, 8.0)
np.std(high_var), np.std(low_var) # (2072.712, 2.828)

np.mean(high_var), np.mean(low_var) # (1600.1667, 6.0)
```

We can **change the shape** on a NumPy array when we need to line up our inputs into a specific shape. One way we can do this is with the .reshape( ) method to change the dimensions of our array. It will contain the same information as before but with a different shape (could be used when needing to match dimensions in order to multiply). We can use **transpose** to switch the axis and change the shape of our array. The difference is reshape allows us to create custom shapes while transpose only switches axis.

```
a2.shape # (2, 3)
a2_reshape = a2.reshape(2, 3, 1) # change into a 3D matrix

a2_transpose = a2.T
a2_transpose.shape # (3, 2)
```

We can perform different types of multiplication on NumPy arrays.

Element wise: $\begin{bmatrix} A & B \\ C & D \end{bmatrix}_{2x2} * \begin{bmatrix} E & F \\ G & H \end{bmatrix}_{2x2} = \begin{bmatrix} A*E & B*F \\ C*G & D*H \end{bmatrix}_{2x2}$

Dot product: $\begin{bmatrix} A & B & C \\ D & E & F \\ G & H & I \end{bmatrix}_{3x3} * \begin{bmatrix} J & K \\ L & M \\ N & O \end{bmatrix}_{3x2} = \begin{bmatrix} A*J+B*L+C*N & A*K+B*M+C*O \\ D*J+E*L+F*N & D*K+E*M+F*O \\ G*J+H*L+I*N & G*K+H*M+I*O \end{bmatrix}_{3x2}$

Note that for the dot product, the inside numbers must match (row length of first matrix must match column length of second matrix).

```
np.random.seed(0)

mat1 = np.random.randint(10, size=(5,3)) # range 0-10 of 5x3 array
mat2 = np.random.randint(10, size=(5,3)) # range 0-10 of 5x3 array

np.multiply(mat1, mat2) # element wise multiplication

mat1.shape, mat2.T.shape # ((5, 3), (3, 5))

mat3 = np.dot(mat1, mat2.T)
mat3.shape # (5, 5)
```

Dot product example

```
np.random.seed(0)

sales_amount = np.random.randint(20, size=(5,3)) # 5x3 matrix of sales

weekly_sales = pd.DataFrame(sales_amounts,
                            index = ['Mon','Tues','Wed','Thrs','Fri','Sat','Sun'],
                            columns = ['Almond', 'Peanut', 'Cashew'])

prices = np.array([10, 8, 12])
prices.shape # (3,) but we need (1,3)

butter_prices = pd.DataFrame(prices.reshape(1,3),
                             index = ['Price'],
                             columns = ['Almond', 'Peanut', 'Cashew'])

butter_prices.shape, weekly_sales.shape # ((1, 3), (5, 3))
daily_sales = butter_prices.dot(weekly_sales.T)
# creates a 1x5 data frame with Price as row and days as columns

weekly_sales['Total ($)'] = daily_sales.T # flip so it become 5x1 matrix
```

3.2.1 Sorting Arrays

We can sort NumPy array's in different ways. Using the np.sort( ) method will sort each row an ascending order. We can use np.argsort( ) to get an array where each index is the position in the row that the values would be sorted in.

```
rand_arr = np.random.randint(10, size=(3,5))

np.argmin(a1) # index for minimum value (0)
np.argmax(a1) # index for maximum value (2)

np.argmax(rand_array, axis=0) # return the index pos for the max value in each column
np.argmax(rand_array, axis=1) # return the index pos for the max value in each row
```

3.2.2 Turn Images into NumPy Arrays

We can read in images using matplotlib into a NumPy array by using the imread( ) function. It takes the pixel values (color values) and stores them in a NumPy array.

```
import matplotlib.image import imread

panda = imread('images/panda.png')
panda.size, panda.shape # (2446500, (2330, 3500, 3))
```

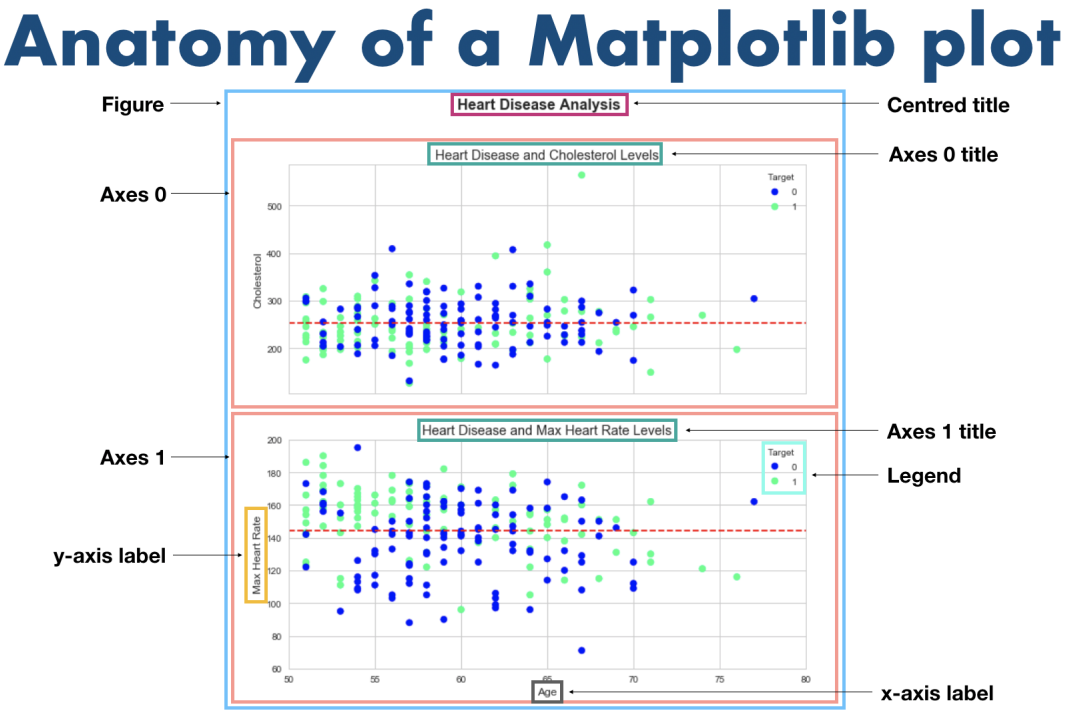
Turning your data into numbers is one of the biggest steps in machine learning. Now with this array, and after training our model, we would be able to feed in the array and the model would compare numeric values to predict what it thinks the picture is using patterns in the data.

4 Matplotlib: Plotting and Visualizing

4.1 Types of Plots

When graphing our plots, we want to follow the object oriented API pattern of assigning a figure and axes object to create a graph. Lets take a look at a typical workflow we would want to follow:

```
import matplotlib.pyplot as plt

# 1. Prepare data
x = [1, 2, 3, 4]
y = [11, 22, 33, 44]

# 2. Setup plot (create our objects)
fig, ax = plt.subplots(figsize=(10, 10)) # (width, height)

# 3. Plot data
ax.plot(x, y)

# 4. Customize plot
ax.set(title='Sample Plot',
       xlabel = 'x-axis',
       ylabel = 'y-axis')

# 5. Save and show your figure
fig.savefig('sample-plot.png)
```



We can create figures using data from NumPy arrays. Some common plot types we might use are:

```
x = np.linespace(0, 10, 100) # 100 evenly spaced numbers between 0 and 10

fig, ax = plt.subplots()

ax.plot(x, x**2) # create a line graph
ax.scatter(x, np.exp(x)) # create a scatter plot
ax.bar(['Almonds', 'Peanuts'], [10, 12]) # create a bar chart
ax.hist(x) # create a histogram
```

4.1.1 Subplots

We can create figures with subplots that can each contain different data or be different graph types. In order to access the subplots, we chose one of the following methods. If we pass in a tuple when creating the objects, we can access by calling a graph on the given object. If we create the objects without tuples, we must use indexing in order to access the graph at the position we want.

```
x = np.random.randn(1000)

# Create a 10x5 figure with 4 subplots (2 in each row/column)
fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(nrows = 2,
                                             ncols = 2,
                                             figsize = (10,5))
# access sublots: ax1.plot(), ax2.hist(), etc.

# Create a 10x5 figure with 4 subplots (2 in each row/column)
fig, ax = plt.subplots(nrows = 2,
                       ncols = 2,
                       figsize = (10,5))
# access subplots: ax[0,0].plot(), ax[0,1].scatter(), etc.
```

4.2 Plotting from Pandas DataFrames

Before we begin to plot our data, we need to make sure that the columns are numeric. We can do this with *regular expression (regex)*. We also may want to create new columns from data that already exist. Note that we can **plot directly** from a Pandas data frame using a built-in matplotlib method with the parameter *kind=* to specify the type of graph we want.

```
car_sales = pd.read_csv('car-sales.csv')

car_sales['Price'] = car_sales['Price'].str.replace('[\$\,\.]', '') # remove symbols
car_sales['Price'] = car_sales['Price'].str[:-2] # remove extra zeros
car_sales['Price'] = car_sales['Price'].astype(int) # cast as an integer

# create a sale date column for each row in our data frame
car_sales['Sale Date'] = pd.date_range('1/1/2020', periods = len(car_sales))

# create total sales column by summing all prices
car_sales['Total Sales'] = car_sales['Price'].cumsum()

car_sales.plot(x='Odometer (KM)', y='Price', kind='scatter')
plt.show()
```

It is easy to use the built in plotting methods when creating simple graphs, but for creating more advanced plots we want to use the object-oriented API for matplotlib. Creating a figure and axes will allow us to add more features and variables to our plots.

```
heart_disease = pd.read_csv('heart-disease.csv')

over_50 = heart_disease[heart_disease['age'] > 50]

# create a scatter plot of age vs. chol, colored by the target values (OO Method)
fig, ax = plt.subplots(figsize=(10, 6))
scatter = ax.scatter(x=over_50['age'], y=over_50['chol'], c=over_50['target'])

ax.set(title='Heart Disease and Cholesterol Levels', xlabel='Age',
       ylabel='Cholesterol')
ax.legend(*scatter.legend_elements(), title='Target') # legend elements from 'c='

ax.axhline(over_50['chol'].mean(), linestyle='--')
```

We can also create **subplots** directly from our Pandas data frames. By default, it uses the same scale and range for every subplot (which can start to look messy). Similar to above, we need to use the object-oriented API for creating subplots from axes objects. (note that the below code will output a similar graph to the 'anatomy of a matplotlib plot' picture above).

```
fig, (ax0, ax1) = plt.subplots(nrows=2, ncols=1, figsize=(10,10))

# Plot and customize axis 0 graph
scatter = ax0.scatter(x=over_50['age'], y=over_50['chol'], c=over_50['target'])
ax0.set(title='Heart Disease and Cholesterol Levels', xlabel='Age',
        ylabel='Cholesterol')
ax0.legend(*scatter.legend_elements(), title='Target')
ax0.axhline(over_50['chol'].mean(), linestyle='--') # mean line

# Plot and customize axis 1 graph
scatter = ax1.scatter(x=over_50['age'], y=over_50['thalach'], c=over_50['target'])
ax1.set(title='Heart Disease and Max Heart Rate', xlabel='Age',
        ylabel='Max Heart Rate')
ax1.legend(*scatter.legend_elements(), title='Target')
ax1.axhline(over_50['thalach'].mean(), linestyle='--') # mean line

fig.suptitle('Heart Disease Analyis', fontsize=16, fontweight='bold')
```

4.3 Customizing Plots

We can customize our plots to not only make them more visually appealing but also to more easily read the data from them. We can see all the available styles by using the *plt.style.available* function. You want to run the *plt.style.use()* method at the beginning of the notebook to make sure all graphs are the same style. Below we changed the graph from above and added: cmap, set_xlim, set_ylim.

```
plt.style.use('seaborn-whitegrid') # change the style of our plot

fig, (ax0, ax1) = plt.subplots(nrows=2, ncols=1, figsize=(10,10))

# Plot and customize axis 0 graph
scatter = ax0.scatter(x=over_50['age'], y=over_50['chol'], c=over_50['target'],
                      cmap='winter') # this changes the color scheme
ax0.set(title='Heart Disease and Cholesterol Levels', xlabel='Age',
        ylabel='Cholesterol')
ax0.set_xlim([50, 80]) # change the x-axis range limits
ax0.set_ylim([60, 200]) # change the y-axis range limits
ax0.legend(*scatter.legend_elements(), title='Target')
ax0.axhline(over_50['chol'].mean(), linestyle='--') # mean line

# Plot and customize axis 1 graph
scatter = ax1.scatter(x=over_50['age'], y=over_50['thalach'], c=over_50['target'],
                      cmap='winter') # this changes the color scheme
ax1.set(title='Heart Disease and Max Heart Rate', xlabel='Age',
        ylabel='Max Heart Rate')
ax1.set_xlim([50, 80]) # change the x-axis range limits
ax1.set_ylim([60, 200]) # change the y-axis range limits
ax1.legend(*scatter.legend_elements(), title='Target')
ax1.axhline(over_50['thalach'].mean(), linestyle='--') # mean line

fig.suptitle('Heart Disease Analyis', fontsize=16, fontweight='bold')

fig.savefig('Heart-disease-analysis.png') # saves to current directory
```

5 Scikit-learn: Machine Leaning Models

5.1 Workflow

We will follow a typical workflow when working with machine learning models. We will go over each step in detail, but this section is a general overview of how our it will look. We have the following steps:

1) Getting the data ready.
2) Choose the right estimator/algorithm for our problem.
3) Fit the model/algorithm and use it to make predictions on our data.
4) Evaluating a model.
5) Improve a model.
6) Save and load a trained model.

```
import pandas as pd
import numpy as np

# Step 1: getting data ready
heart_disease = pd.read_csv('data/heart-disease.csv')
X = heart_disease.drop('target', axis=1) # create features matrix
y = heart_disease['target'] # create labels matrix

# Step 2: Chose the model and hyperparameters
from sklearn.ensemble import RandomForestClassifier
clf = RandomForestClassifier(n_estimators = 100)
clf.get_params() # show the parameter values

#Step 3: Fit the model to the training data
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2)
clf.fit(X_train, y_train) # find patterns in training data
y_preds = clf.predict(X_test)

# Step 4: Evaluate the model on training and test data
clf.score(X_train, y_train) # 1.0
clf.score(X_test, y_test) # 0.754

from sklearn.metrics import classification report, confusion_matrix, accuracy_score
classification_report(y_test, y_preds) # compare test labels to pred labels
confusion_matrix(y_test, y_preds)
accuracy_score(y_test, y_preds) # 0.754

# Step 5: Improve the model
np.random.seed(42)
for i in range(10, 100, 10):
  print(f'Trying model with {i} estimators...')
  clf = RandomForestClassifier(n_estimators = i).fit(X_train, y_train)
  print(f'Model accuracy on test set: {clf.score(X_test, y_test) * 100:.2f}%')
  print("") # after running loop, we see 20 estimators gets us to 83% accuracy

# Step 6: Save model and load it
import pickle

pickle.dump(clf, open('random_forst_model_1.pkl', 'wb')) # saves to a file in dir

loaded_model = pickle.load(open('random_forst_model_1.pkl', 'rb'))
loaded_model.score(X_test, y_test)
```

5.2 Step 1: Getting Data Ready

There are three main things we must to when getting our data ready to be used in a ML model:

1) Split the data into features (X) and labels (y).
2) Filling (also called imputing) or disregarding missing values.
3) Converting non-numerical values to numerical values (called feature coding).

Lets begin with **step 1**:

```
heart_disease = pd.read_csv('heart-disease.csv')

# Split into features and labels
X = heart_disease.drop('target', axis=1)
y = heart_disease['target']

# Split into training and test sets
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2)
```

Lets jump to **step 3**, where we want to **convert categorical to numerical data**. We will use a different data set as an example since our heart-disease.csv is already all numeric. Our 'Make' column is categorical data with strings representing the values, so we need to convert these to numerical in order to use in our machine learning model.

One Hot Encoding is a process used to turn categories into numbers. It encodes values into 0's and 1's that allows us to feed it into a machine learning model. We can also do this is with the pd.get_dummies( ) method, which does the same thing as OneHotEncoder.

Car	Colour
0	Red
1	Green
2	Blue
3	Red

Car	Red	Green	Blue
0	1	0	0
1	0	1	0
2	0	0	1
3	1	0	0

```
np.random.seed(42)
car_sales = pd.read_csv('car-sales-extended.csv')

X = car_sales.drop['Price', axis=1]
y = car_sales['Price']

# Convert categorical to numerical data
from sklearn.preprocessing import OneHotEncoder
from sklearn.compose import ColumnTransformer

# Option 1: One Hot Encoder method
categorical_features = ['Make', 'Colour', 'Doors']
one_hot = OneHotEncoder()
transformer = ColumnTransformer([('one_hot', one_hot, categorical_features)],
                                remainder='passthrough')
transformed_X = transformer.fit_transform(X)

# Option 2: Pandas get_dummies() method
dummies = pd.get_dummies(car_sales[['Make', 'Colour', 'Doors']])

# Fit the model with the new data
X_train, X_test, y_train, y_test = train_test_split(transformed_X, y,
                                                    test_size = 0.2)

from sklearn.ensemble import RandomForestRegressor # predicts numbers
model = RandomForestRegressor()
model.fit(X_train, y_train)
model.score(X_test, y_test) # .3043 (not much data for each sample)
```

For **step 2**, there are two main ways of dealing with missing data:

1) Fill them with some value (also known as imputation).
2) Remove the samples with missing data altogether.

Before we can convert our data into numerical values, we must drop or fill the NaN values. Lets look at the first option, which is **filling the missing data with Pandas**. We want to drop any samples that do not have a label since we don't want to try and predict them ourselves.

```
car_sales_missing = pd.read_csv('data/car-sales-extended-missing-data.csv')

car_sales_missing.isna().sum() # around 50 values missing in each column

################ Fill in missing values using Pandas ################
car_sales_missing['Make'].fillna('missing', inplace=True)
car_sales_missing['Colour'].fillna('missing', inplace=True)
car_sales_missing['Odometer (KM)'].fillna(car_sales_missing['Odometer (KM)'].mean(),
                                          inplace=True)
car_sales_missing['Doors'].fillna(4, inplace=True)
car_sales_missing.dropna(inplace=True) # drop missing values in Price column

# Split data and concert to numerical
X = car_sales_missing.drop['Price', axis=1]
y = car_sales_missing['Price']

categorical_features = ['Make', 'Colour', 'Doors']
one_hot = OneHotEncoder()
transformer = ColumnTransformer([('one_hot', one_hot, categorical_features)],
remainder='passthrough')
transformed_X = transformer.fit_transform(X)
```

Our second option for missing values is to **fill NaN values with scikit-learn**. An important note is that you want to split your train/test data before filling missing values so that you don't use test data to fill training data (and vice versa).

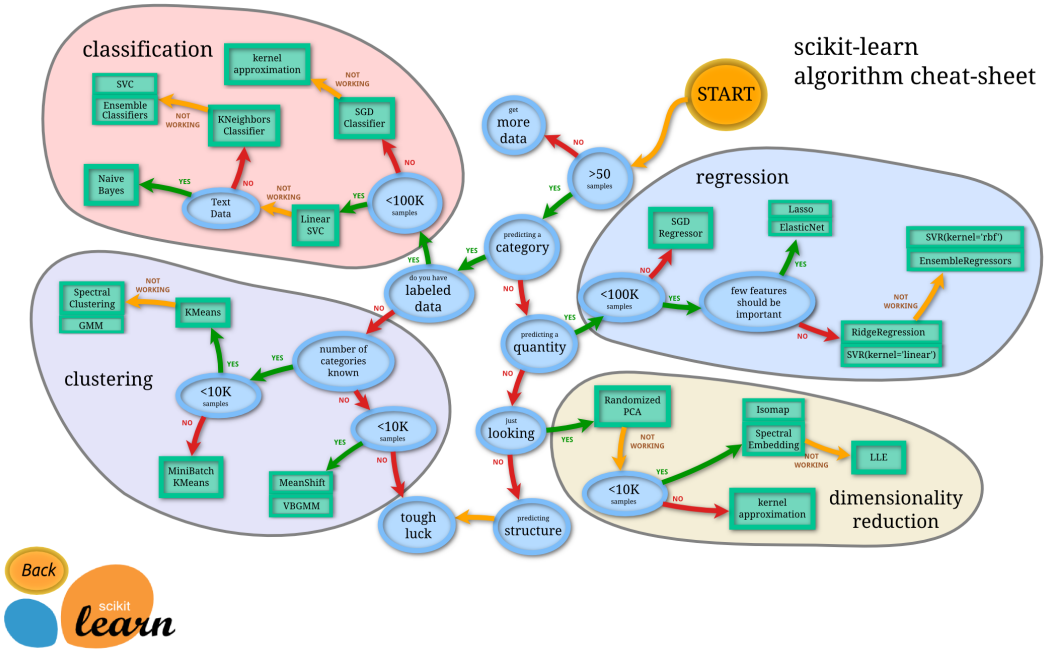
```
car_sales_missing = pd.read_csv('data/car-sales-extended-missing-data.csv')

car_sales_missing.dropna(subset=['Price'], inplace=True) # drop rows with no labels
X = car_sales_missing.drop['Price', axis=1]
y = car_sales_missing['Price']

# Split into train/test data before filling missing values
np.random.seed(42)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

################ Fill missing values using scikit-learn ################
from sklearn.impute import SimpleImputer
from sklearn.compose import ColumnTransformer

# Fill categorical values with 'missing' & numerical values with mean
cat_imputer = SimpleImputer(strategy="constant", fill_value="missing") # categorical
door_imputer = SimpleImputer(strategy="constant", fill_value=4) # door column
num_imputer = SimpleImputer(strategy="mean") # numerical

cat_features = ["Make", "Colour"]
door_feature = ["Doors"]
num_features = ["Odometer (KM)"]

# Create an imputer (something that fills missing data)
imputer = ColumnTransformer([
  ("cat_imputer", cat_imputer, cat_features), # (name, imputer, features)
  ("door_imputer", door_imputer, door_feature),
  ("num_imputer", num_imputer, num_features) ])
```

```
29
30  # Fill train and test values separately
31  filled_X_train = imputer.fit_transform(X_train)
32  filled_X_test = imputer.transform(X_test)
33
34  # Get our transformed data array's back into DataFrame's
35  car_sales_filled_train = pd.DataFrame(filled_X_train,
36                                        columns=["Make", "Colour", "Doors",
37                                                 "Odometer (KM)"])
38
39  car_sales_filled_test = pd.DataFrame(filled_X_test,
40                                       columns=["Make", "Colour", "Doors",
41                                                "Odometer (KM)"])
42
43  # Use OneHotEncoder to encode the data
44  categorical_features = ["Make", "Colour", "Doors"]
45  one_hot = OneHotEncoder()
46  transformer = ColumnTransformer([("one_hot", one_hot, categorical_features)],
47                                  remainder="passthrough")
48
49  # Fill train and test values separately
50  transformed_X_train = transformer.fit_transform(car_sales_filled_train)
51  transformed_X_test = transformer.transform(car_sales_filled_test)
52
53  np.random.seed(42)
54  from sklearn.ensemble import RandomForestRegressor
55
56  model = RandomForestRegressor(n_estimators=100)
57  model.fit(transformed_X_train, y_train)
58  model.score(transformed_X_test, y_test) # 0.2123 (lower score due to less samples)
```

5.3 Step 2: Choosing the Right Model

Scikit-learn uses **estimator** as another term for machine learning model or algorithm. There are two different types of estimators we will focus on:

1) Classification - predicting whether a sample is one thing or another.
2) Regression - predicting a number value.



Click here for the link to the picture (all green boxes are clickable).

5.3.1 Regression models

We will use the dataset within the scikit-learn library that contains information on housing in Boston. Looking at the picture above, we have over 50 samples and we want to predict a quantity. We also have less than 100k samples and more than a few features are important, so we will use *RidgeRegression*.

```
from sklearn.datasets import load_boston
from sklearn.linear_model import Ridge
np.random.seed(42)

boston = load_boston() # loads in as a dictionary
boston_df = pd.DataFrame(boston['data'], columns=boston['feature_names'])
boston_df['target'] = pd.Series(boston['target'])
len(boston_df) # 506 samples

X = boston_df.drop('target', axis=1)
y = boston_df['target']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

model = Ridge()
model.fit(X_train, y_train)
model.score(X_test, y_test) # R^2 = 0.6662
```

From this point, our Ridge model is currently working. But what if it wasn't? Also, how can we improve our R^2 score for the model? If the model wasn't working, we can use **ensemble regressors**, which use several base estimators to improve over a single estimator. One example of this method is by using the *RandomForestRegressor*. We will use the train/test data from above.

```
from sklearn.ensemble import RandomForestRegressor
np.random.seed(42)

rf = RandomForestRegressor(n_estimators = 100)
rf.fit(X_train, y_train)
rf.score(X_test, y_test) # R^2 = 0.87397
```

5.3.2 Classification models

We will use the heart-disease.csv to classify whether or not someone has heart disease. We have more than 50 samples and we are predicting a category. We have labeled data and less than 100k samples, so we want to use a *LinearSVC* classifier (we find this by looking at the image map above).

```
from sklearn.svm import LinearSVC
np.random.seed(42)

heart_disease = pd.read_csv('data/heart-disease.csv')

X = heart_disease.drop('target', axis=1)
y = heart_disease['target']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

clf = LinearSVC(max_iter=10000)
clf.fit(X_test, y_test)
clf.score(X_test, y_test) # 0.4754
```

This is a binary classification problem, and our model in only at a 47% accuracy. This is not good since there are only two classes, so if we were to guess we would only get half of them correct (this could mean the model isn't finding all of the patterns). Looking at the image above, we see that we want to use **ensemble classifiers** in order to improve our score.

```
from sklearn.ensemble import RandomForestClassifier
np.random.seed(42)

clf = RandomForestClassifier(n_estimators = 100)
clf.fit(X_train, y_train)
clf.score(X_test, y_test) # R^2 = 0.85246
```

The key takeaway from this section is that if you have **structured data**, use ensemble methods as the estimator. The Random Forest is known for its robustness and ability to have a high R^2 score. If you have **unstructured data**, use deep/transfer learning.

5.4 Step 3: Making Predictions with our Model

Classification:

There are two ways to use a trained model to make predictions. One way is with the **.predict( )** function. This will return an array of labels, one for each sample/row, which we can then compare to the truth labels (y_test). We will use the RandomForestClassifier model from above as our trained estimator.

```
# Compare predictions to truth labels to evalaute the model
y_preds = clf.predict(X_test)
np.mean(y_preds == y_test) # 0.85246
```

The second way we can make predictions is with the **.predict_proba( )** function. This will return an array of probabilities for each classification label (ordered by the label of classes). The label with the largest probability will be the class for which the model assigns the prediction.

```
clf.predict_proba(X_test) # first value in array is [0.89, 0.11]
clf.predict(X_test) # first value is 0 (since label 0 had larger probability)
```

Regression:

The scikit-learn .predict( ) method can also be used on regression models. We can then compare the predictions to the truth using mean_absolute_error, which takes the difference between the prediction and truth and takes the average of them all (plus/minus value).

```
X = boston_df.drop('target', axis=1)
y = boston_df['target']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

model = RandomForestRegressor(n_estimators = 100).fit(X_train, y_train)
y_preds = model.predict(X_test)

from sklearn.metrics import mean_absolute_error
mean_absolute_error(y_test, y_preds) # 2.12 away for target
```

5.5 Step 4: Evaluating a Model

There are 3 different ways to evaluate a scikit-learn model/estimator:

1) Estimator 'score' method.
2) The 'scoring' parameter.
3) Problem-specific metric functions.

We will begin with the **score method**. The score() function has a default evaluation metric built in to it. For *regression*, it will use the coefficient of determination (R^2). For *classification*, it returns the mean accuracy on the data that is passed in.

```
from sklearn.ensemble import RandomForestClassifier
np.random.seed(42)

X = heart_disease.drop('target', axis=1)
y = heart_disease['target']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

clf = RandomForestClassifier(n_estimators = 100).fit(X_train, y_train)
clf.score(X_test, y_test) # 0.852459

```

Next we will look at the **scoring parameter**. We can use a **cross validation** method, which splits the data into 'k' different splits and is evaluated on each one. Note that we pass X and y into the function, not the test data splits. It avoids getting lucky high scores by training and evaluating on all of the data. We can then take the average of the scores to find our value.

Note that if the *scoring=* parameter is set to None, it will use the default scoring parameter of our estimator (R^2 for regression and *mean accuracy* for classification).

```
from sklearn.model_selection import cross_val_score
np.random.seed(42)

clf_cross_val_score = np.mean(cross_val_score(clf, X, y, cv=5)) # 0.8248
```

5.5.1 Evaluating a Classification Model

There are 4 main evaluation metrics that we can use for classification models: accuracy, area under ROC curve, confusion matrix, and classification report.

Lets begin with the **accuracy** evaluation metric:

```
from sklearn.model_selection import cross_val_score
from sklearn.ensemble import RandomForestClassifier
np.random.seed(42)

X = heart_disease.drop('target', axis=1)
y = heart_disease['target']

clf = RandomForestClassifier(n_estimators = 100)
c_v_score = cross_val_score(clf, X, y, cv=5)
np.mean(c_v_score) # 0.8248
```

Next, lets look at the area under the receiver operating characteristic curve (**AUC/ROC**). ROC curves are a comparison of a model's true positive rate (tpr) vs a models false positive rate (fpr).

- True positive: model predicts 1 when truth is 1.
- False positive: model predicts 1 when truth is 0.
- True negative: model predicts 0 when truth is 0.
- False negative: model predicts 0 when truth is 1.

```
from sklearn.metrics import roc_curve

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

clf = RandomForestClassifier(n_estimators = 100).fit(X_test, y_test)
y_probs = clf.predict_proba(X_test)

y_probs_positive = y_probs[:, 1] # only keep columns with 1's probability
```

```
fpr, tpr, threshold = roc_curve(y_test, y_probs_positive)

def plot_roc_curve(fpr, tpr):
  plt.plot(fpr, tpr, color='orange', label='ROC') # roc curve
  plt.plot([0,1], [0,1], color='darkblue', linestyle='--', label='Guessing') # base
  plt.xlabel('False Positive Rate (FPR)')
  plt.ylabel('True Positive Rate (TPR)')
  plt.title('Receiver Operating Characteristic (ROC) Curve')
  plt.legend()
  plt.show()

plot_roc_curve(fpr, tpr)

from sklearn.metrics import roc_auc_score
roc_auc_score(y_test, y_probs_positive) # 0.84967
```

A **confusion matrix** is a quick way to compare the labels a model predicts and the actual labels it was supposed to predict. In essence, it gives you an idea of where the model is getting 'confused'.

It is easiest to visualize a confusion matrix with the *pd.crosstab( )* function, but we can also get more advanced with Seaborn's *heatmap()* function.

```
from sklearn.metrics import confusion_matrix

y_preds = clf.predict(X_test)
conf_mat = confusion_matrix(y_test, y_preds)
# create a table which shows true pos/neg and false pos/neg results
pd.crosstab(y_test, y_preds, rownames=['Actual Label'], colnames=['Predicted Label'])

import seaborn as sns

def plot_conf_mat(conf_mat):
  fig, ax = plt.subplots(figsize=(3,3))
  ax = sns.heatmap(conf_mat, annot=True, cbar=False)
  plt.xlabel('True Label')
  plt.ylabel('Predicted Label')
  plt.show()
```

The final evaluation metric is a **classification report**. It is a collection of the following metrics:

- Precision : proportion of positive identifications (1) that were correct.
- Recall : proportion of actual positives that were correctly classified.
- F1 score : combination of recall and precision (want a score near 1.0).
- Support : the number of samples each metric was calculated on.
- Accuracy : the percent of correct predictions.
- Macro average : average of precision, recall, and F1 score between classes (watch for class imbalance).
- Weighted Average : same as macro average, but with respect to number of samples.

Note that for data with *class imbalances* (more samples in one class than another), you want to use a metric other than just accuracy.

```
from sklearn.metrics import classification_report

classification_report(y_test, y_preds)
```

Classification metrics summary:

- **Accuracy** is good to start with if all classes are balanced.
- **Precision/Recall** become more important when there is a class imbalance.
- If false positive predictions are worse than false negatives, aim for higher precision.
- If false negative predictions are worse than false positives, aim for higher recall.
- **F1-score** is a combination of precision and recall.

5.5.2 Evaluating a Regression Model

There are 3 main model evaluation metrics that we will focus on: R^2 (coefficient of determination), MAE (mean absolute error), and MSE (mean squared error). First lets set up a regression model to evaluate.

```
import sklearn.ensemble import RandomForestRegressor
np.random.seed(42)

X = boston_df.drop('target', axis=1)
y = boston_df['target']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

model = RandomForestRegressor(n_estimators = 100).fit(X_train, y_train)
```

R^2 is the default return value from the .score( ) method. This compares your models prediction to the mean of the target. Values can range from negative infinity to 1. If your model predicts the mean every time, it will have a value of 0. But if your model perfectly predicts a range of numbers it's R^2 value would be 1.

```
from sklearn.metrics import r2_score

mode.score(X_test, y_test) # 0.87387

# create an array of y_test mean values to compare of predictions
y_test_mean = np.full(len(y_test), y_test.mean())

r2_score(y_test, y_test_mean) # 0.0
r2_score(y_test, y_test) # 1.0
```

Now lets take a look at **MAE** (mean absolute error). This is the average of the absolute value of the difference between predicted and actual values. It gives you an idea of how wrong your models prediction are on average (plus/minus the given value from the actual value).

```
from sklearn.metrics import mean_absolute_error

y_preds = model.predict(X_test)
mae = mean_absolute_error(y_test, y_preds) # 2.12
```

Finally lets look at **MSE** (mean squared error). This will always be higher than MAE because it squares the errors. It squares the difference between predicted and actual values, then takes the mean value.

```
from sklearn.metrics import mean_squared_error

mse = mean_squared_error(y_test, y_preds) # 9.242
```

Which regression metric should you use? You want to minimize MSE and MAE while maximizing the R^2 value.

- **R^2** is similar to accuracy. It gives you a quick indication but doesn't tell you how wrong your model is in terms of difference for each prediction.
- **MAE** gives better indication of how fart off each of your models prediction are on average.
- **MSE** will amplify larger differences.

5.5.3 Cross Validation and Scoring parameter

By default the cross validation method uses the default scoring parameter: for classification this is *mean accuracy*, and for regression this is the R^2 value. Note that for regression, the MSE and MAE are negative since as the value gets larger (moves from negative to positive) it means are model is performing better. In the future, it could be helpful to write a function that will neatly output these metrics for the user.

```
from sklearn.model_selection import cross_val_score
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import RandomForestRegressor
np.random.seed(42)

# Classification model scoring
X = heart_disease.drop('target', axis=1)
y = heart_disease['target']
clf = RandomForestClassifier(n_estimators = 100)

cv_acc = cross_val_score(clf, X, y, cv=5)
cv_precision =  cross_val_score(clf, X, y, cv=5, scoring='precision')
cv_recall = cross_val_score(clf, X, y, cv=5, scoring='recall')
cv_f1 = cross_val_score(clf, X, y, cv=5, scoring='f1')

# Regression model scoring
X = boston_df.drop('target', axis=1)
y = boston_df['target']
model = RandomForestRegressor(n_estimators = 100)

cv_r2 = cross_val_score(model, X, y, cv=5)
cv_mae = cross_val_score(model, X, y, cv=5, scoring='neg_mean_absolute_error')
cv_mse = cross_val_score(model, X, y, cv=5, scoring='neg_mean_squared_error')
```

5.5.4 Metric Functions with scikit-learn

Lets begin with classification evaluation functions:

```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
np.random.seed(42)

X = heart_disease.drop('target', axis=1)
y = heart_disease['target']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
clf = RandomForestClassifier(n_estimators = 100).fit(X_train, y_train)

y_preds = clf.predict(X_test)
accuracy_score(y_test, y_preds) # 0.8525
precision_score(y_test, y_preds) # 0.8484
recall_score(y_test, y_preds) # 0.875
f1_score(y_test, y_preds) # 0.8615
```

Next, lets look at regression evaluation functions:

```
from sklearn.metrics import r2_score, mean_absolute_error, mean_squared_error
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split
np.random.seed(42)

X = boston_df.drop('target', axis=1)
y = boston_df['target']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
model = RandomForestRegressor(n_estimators = 100).fit(X_train, y_train)

y_preds = model.predict(X_test)
r2_score(y_test, y_preds) # 0.8739
mean_absolute_error(y_test, y_preds) # 2.122
mean_squared_error(y_test, y_preds) # 9.24
```

5.6 Step 5: Improving a Model

The first predictions you make are the **baseline predictions**. The first model you build is the **baseline model**. We want to focus on how to improve both of these in two different ways.

The first is from a data perspective:

1) Could we collect more data? (generally the more data, the better).
2) Could we improve our data?

The second is from a model perspective:

1) Is there a better model? (moving from simple to complex models).
2) Could we improve the current model? (hyperparameters).

The difference between parameters and hyperparameters are that **parameters** are patterns in data that are model will find, while **hyperparameters** are settings on a model that we can adjust to *potentially* improve its ability to find patterns.

We can use the .get_params() method to see the hyperparameters that we are able to adjust.

```
from sklearn.ensemble import RandomForestClassifier

clf = RandomForestClassifier()
clf.get_params()
```

There are **three ways to adjust hyperparameters** that will will look at: manually by hand, randomly with RandomizedSearchCV, and exhaustively with GridSearchCV.

5.6.1 Tuning Hyperparameters Manually

When tuning hyperparameters, we will us the **validation set** (10-15% of the data). Lets begin by writing a function that will print evaluation metrics from a *classification* model:

```
def evaluate_preds(y_true, y_preds):
  accuracy = accuracy_score(y_true, y_preds)
  precision = precision_score(y_true, y_preds)
  recall = recall_score(y_true, y_preds)
  f1 = f1_score(y_true, y_preds)
  metric_dict = {'accuracy': round(accuracy, 2),
                 'precision': round(precision, 2),
                 'recall': round(recall, 2),
                 'f1': round(f1, 2) }
  print('Acc: {}'.format(accuracy * 100:0.2f))
  print('Precision: {}'.format(precision:0.2f))
  print('Recall: {}'.format(recall:.2f))
  print('F1 Score: {}'.format(f1:0.2f))
  return metric_dict
```

Next lets split the data into training, validation, and test split (by hand):

```
from sklearn.ensemble import RandomForestClassifier
np.random.seed(42)

heart_disease_shuffled = heart_disease.sample(frac=1)
X = heart_disease_shuffled.drop('target', axis=1)
y = heart_disease_shuffled['target']

train_split = round(0.7 * len(heart_disease_shuffled)) # 70% of data
valid_split = round(train_split + 0.15 * len(heart_disease_shuffled)) # 15% of data
X_train, y_train = X[:train_split], y[:train_split]
X_valid, y_valid = X[train_split:valid_split], y[train_split:valid_split]
X_test, y_test = X[valid_split:], y[valid_split:]
```

We can now use these splits to see how our model is performing and tune any hyperparameters:

```
clf = RandomForestClassifier()
clf.fit(X_train, y_train)
y_preds = clf.predict(X_valid) # baseline predictions
baseline_metrics = evaluate_preds(y_valid, y_preds) # 80%, 0.77, 0.92, 0.84

# Create a second classifier with different hyperparameters & make predictions
clf_2 = RandomForestClassifier(n_estimators=100)
clf_2.fit(X_train, y_train)
y_preds_2 = clf_2.predict(X_valid)
clf_2_metrics = evaluate_preds(y_valid, y_preds_2) # 82.2%, 0.84, 0.84, 0.84
# we see a slight boost in accuracy, higher precision, lower recall, same f1
```

5.6.2 Tuning Hyperparameters with RandomizedSearchCV

We can tune hyperparameters using **RandomizedSearchCV**. We will create a dictionary of the hyperparameters we want to tune with the parameter as the key and the value as the parameter values we want to try. We don't need a validation set since we will use a k-fold cross validation. The model below will test 50 different combinations (5 cv sets for 10 different models).

```
from sklearn.model_selection import RandomizedSearchCV
np.random.seed(42)

grid = {'n_estimators': [10, 100, 200, 500, 1000, 1200],
        'max_depth': [None, 5, 10, 20, 30],
        'max_features': ['auto', 'sqrt'],
        'min_samples_split': [2, 4, 6],
        'min_samples_leaf': [1, 2, 4]}

X = heart_disease_shuffled.drop('target', axis=1)
y = heart_disease_shuffled['target']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
clf = RandomForestClassifier(n_jobs=1) # n_jobs is dedicated CPU power

# Setup RandomizedSearchCV
rs_clf = RandomizedSearchCV(estimator=clf,
                            param_distributions=grid,
                            n_iter=10, # number of models to try
                            cv=5, # creates 5 different validation sets
                            verbose=2)

# Fit the RandomizedSearchCV version of clf
rs_clf.fit(X_train, y_train)
rs_clf.best_params_ # shows which combination got best results

# Make predictions with best hyperparameters & evaluate
rs_y_preds = rs_clf.predict(X_test)
rs_metrics = evaluate_preds(y_test, rs_y_preds) # 81.97%, 0.77, 0.86, 0.81
```

5.6.3 Tuning Hyperparameters with GridSearchCV

For **GridSearchCV**, it will try every single combination that is possible from our grid. Since we know what the best hyperparameters are for our model from the previous RandomizedSearchCV, we can use those to influence which ones we will keep when shrinking the search space in our new grid. We now will only test 60 different models (3*1*2*1*2 choices in our grid multiplied by 5 cv gives us 60 models).

```
from sklearn.model_selection import GridSearchCV, train_test_split
np.random.seed(42)

grid_2 = {'n_estimators': [100, 200, 500],
          'max_depth': [None],
          'max_features': ['auto', 'sqrt'],
          'min_samples_split': [6],
          'min_samples_leaf': [1, 2]}

# Use the data splits and original clf model from previous code block
# Setup GridSearchCV
gs_clf = RandomizedSearchCV(estimator=clf,
                            param_grid=grid_2,
                            cv=5,
                            verbose=2)

# Fit the GridSearchCV version of clf
gs_clf.fit(X_train, y_train)
gs_clf.best_params_

gs_y_preds = gs_clf.predict(X_test)
gs_metrics = evaluate_preds(y_test, gs_y_preds) # 78.7%, 0.74, 0.82, 0.78
```

The final step is to compare the different models metrics.

```
compare_metrics = pd.DataFrame({'baseline': baseline_metrics,
                                'clf_2': clf_2_metrics,
                                'random search': rs_metrics,
                                'grid search': gs_metrics})

compare_metrics.plot.bar(figsize=(10,8));
```

5.7 Step 6: Saving and Loading a Model

After we train and tune a model, we might want to save it so that we can use it outside of our Jupyter Notebook. There are two different ways in which we can save a model: with Python's 'pickle' module, or with the 'joblib' module.

```
import pickle

# Save model in our current working directory using Pickle
pickle.dump(gs_clf, open('gs_random_random_forest_model_1.pkl', 'wb'))

# Load saved model
loaded_pickle_model = pickle.load(open('gs_random_random_forest_model_1.pkl', 'rb'))

# Make predictions using model
pickle_y_preds = loaded_pickle_model.predict(X_test)
evaluate_preds(y_test, pickle_y_preds)
```

```
from joblib import dump, load

dump(gs_clf, filename='gs_random_forest_model_1.joblib') # save a joblib model

loaded_job_model = load(filename='gs_random_forest_model_1.joblib') # load in model

# Make and evaluate joblib predictions
joblib_y_preds = loaded_job_model.predict(X_test)
evaluate_preds(y_test, joblib_y_preds)
```

5.8 Putting It All Together

We will use the **scikit-learn Pipeline** class as a way to string together a bunch of scikit processes in one hit (similar to a function). In one cell block, we want to: fill missing data, convert it to numeric, and build a model on the data.

```
# Modules for getting data ready
import pandas as pd
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import OneHotEncoder

# Modelling tools
from sklearn.ensemle import RandomForestRegressor
from sklearn.model_selection import train_test_split, GridSearchCV

# Setup random seed
import numpy as np
np.random.seed(42)

# Import data and drop rows with missing labels
data = pd.read_csv('data/car-sales-extended-missing-data.csv')
data.dropna(subset=['Price'], inplace=True)

# Define different features and transformer pipeline
categorical_features = ['Make', 'Color']
categorical_transformer = Pipeline(steps=[
  ('imputer', SimpleImputer(strategy='constant', fill_value='missing')),
  ('onehot', OneHotEncoder(handle_unknown='ignore'))]) # turn data into numbers

door_feature = ['Doors']
door_transformer = Pipeline(steps=[
  ('imputer', SimpleImputer(strategy='constant', fill_value='4'))])

numeric_features = ['Odometer (KM)']
numeric_transformer = Pipeline(steps=[
  ('imputer', SimpleImputer(strategy='mean'))])

# Setup preprocessing steps (fill missing values and convert to numeric)
preprocessor = ColumnTransformer(
                    transformer=[
                      ('cat', categorical_transformer, categorical_features),
                      ('door', door_transformer, door_feature),
                      ('num', numeric_transformer, numeric_features) ])

# Create a preprocessing and modelling pipeline
model = Pipeline(steps=[('preprocessor', preprocessor),
                        ('model', RandomForestRegressor()) ])

# Split data
X = data.drop('Price', axis=1)
y = data['Price']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

# Fit and score the model
model.fit(X_train, y_train)
model.score(X_test, y_test) # 0.1822
```

Now that we have trained our mode, we can use GridSearchCV or RandomizedSearchCV with our Pipeline to tune the hyperparameters to try and improve our models score.

The double underscore in the pipe_grid below is used as an accessor. We can think of it as the first value being the step name in the Pipeline we want to access, and following values being the step action within the Pipeline to look for that we want to change.

```
# Use GridSearchCV with our regression Pipeline
from sklearn.model_selection import GridSearchCV

pipe_grid = {
  'preprocessor__num__imputer__strategy': ['mean', 'median'],
  'model__n_estimators': [100, 1000],
  'model__max_depth': [None, 5],
  'model__max_features': ['auto'], # 'model' step, change max_features of model
  'model__min_samples_split': [2, 4]
} # 16 different combinations of parameters

gs_model = GridSearchCV(model, pipe_grid, cv=5, verbose=2) # 16*5 = 80 models
gs_model.fit(X_train, y_train)

gs_model.score(X_test, y_test) # 0.3338 (double the original model score)
```

5.9 Milestone Project 1 (Classification)

We will use the heart disease dataset for this project. Following the typical work flow that we covered in section one, lets map out our data modeling steps.

Step 1: Problem Definition
Given clinical parameters about a patient, can we predict whether or not they have heart disease?

Step 2: Data
The original data came from the Cleveland data from the UCI Machine Learning Repository.There is also a version of it available on Kaggle.

Step 3: Evaluation
If we can reach 95% accuracy at predicting whether or not a patient has heart disease during the proof of concept, we'll pursue the project.

Step 4: Features
This is where you'll get different information about each of the features in your data. We can look at the link above for the UCI Irvine repository to see a description of the 14 features that are within our dataset.

Import Tools

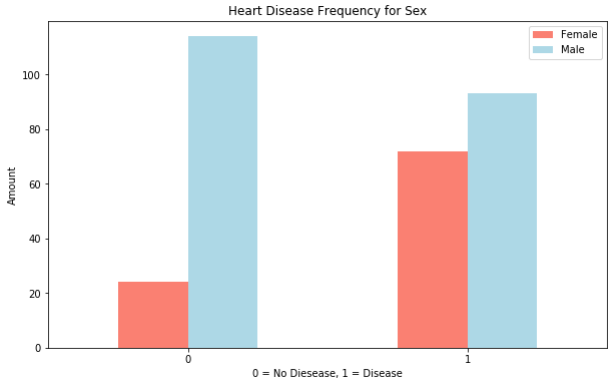
```
# Regular EDA (exploratory data analysis) and plotting libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# Models and Evaluations from Scikit-Learn
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import RandomForestClassifier

from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.model_selection import RandomizedSearchCV, GridSearchCV
from sklearn.metrics import confusion_matrix, classification_report
from sklearn.metrics import precision_score, recall_score, f1_score
from sklearn.metrics import plot_roc_curve
```

5.9.1 Data Exploration (EDA)

The goal here is to find out more about the data and become a subject matter export on the dataset.

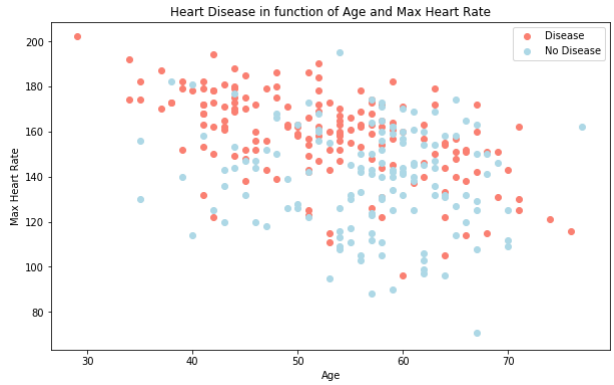
1. What question(s) are you trying to solve?
2. What kind of data do we have and how do we treat different types?
3. What's missing from the data and how do you deal with it?
4. Where are the outliers and why should you care about them?
5. How can you add, change or remove features to get more out of your data?

```
df = pd.read_csv("../data/heart-disease.csv") # data folder from previous dir
df.shape # (303, 14)
df["target"].value_counts() # 1 (165), 0 (138). Even distribution
df.info() # show data types, number of rpws/cols, etc.
df.isna().sum() # find any missing values in each column
df.describe() # get numerical analysis for each column

# Compare target column to the sex column
df.sex.value_count() # 1 (207 males), 0 (96 females)
pd.crosstab(df.target, df.sex).plot(kind="bar", figsize=(10, 6),
                                    color=["salmon", "lightblue"])
plt.title("Heart Disease Frequency for Sex")
plt.xlabel("0 = No Diesease, 1 = Disease")
plt.ylabel("Amount")
plt.legend(["Female", "Male"]);
plt.xticks(rotation=0);
```



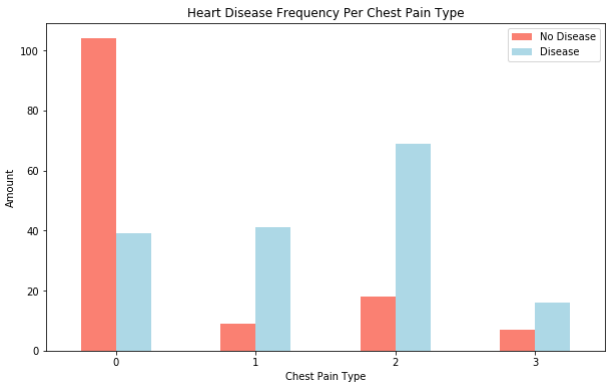
We can see that for our graph, females have about a 75% chance of having heart disease in our data set. For males, they have around a 50% of having heart disease. We also know that there are about twice as many males than females in our data set as well.

```
# Compare age column against max heart rate column for heart disease
plt.figure(figsize=(10,6))
# create scatter plot of age vs max heart rate (positive for heart disease)
plt.scatter(df.age[df.target==1], df.thalach[df.target==1], c='salmon')
# create scatter plot of age vs max heart rate (negative for heart disease)
plt.scatter(df.age[df.target==0], df.thalach[df.target==0], c='salmon')
plt.title("Heart Disease in function of Age and Max Heart Rate")
plt.xlabel("Age")
plt.ylabel("Max Heart Rate")
plt.legend(["Disease", "No Disease"]);
```



All we can see is a downward trend for both target values. As age increases, the maximum heart rate decreases in the individual. There is no real stand out pattern here, but it is a good analysis for understanding the data.

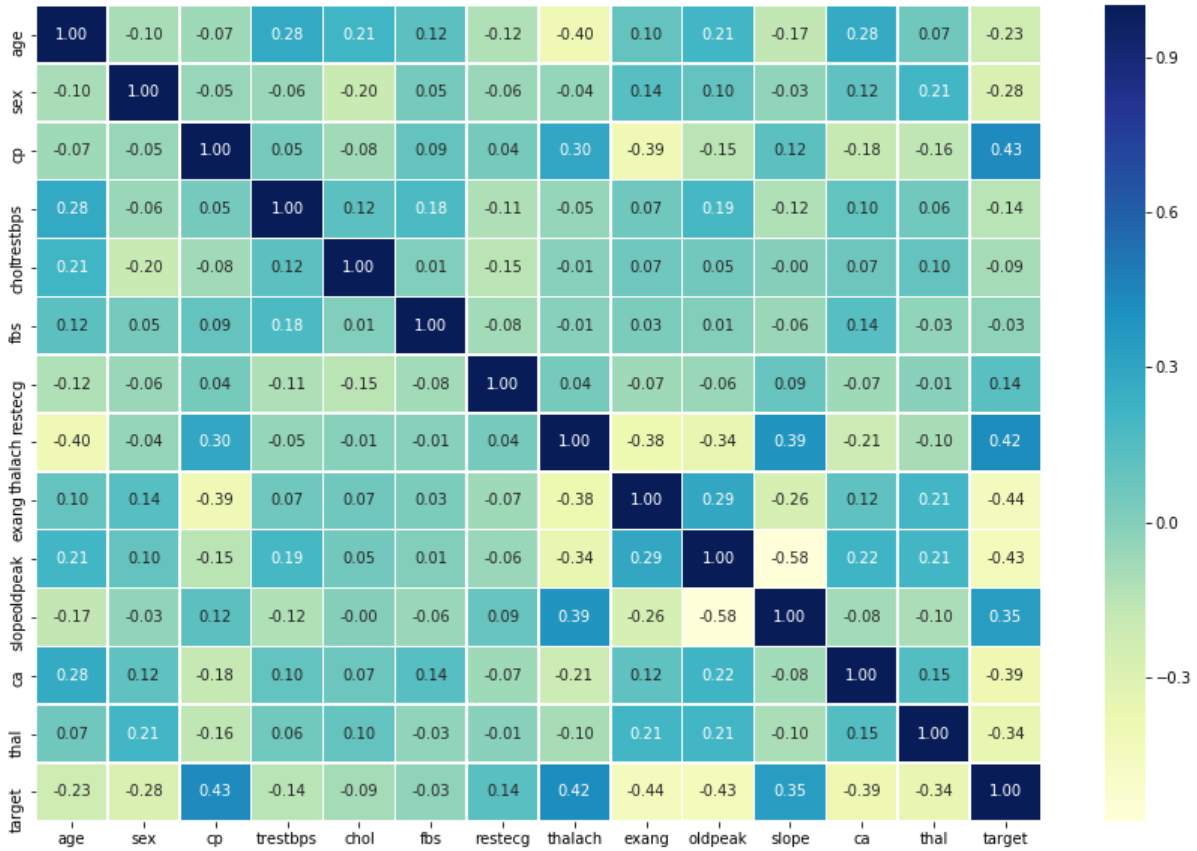
```
# Does chest pain (cp) correlate to having heart disease?
### note that chest pain can have 4 different values (0, 1, 2, 3) see UCI for details

pd.crosstab(df.cp, df.target).plot(kind='bar', figsize=(10,6),
                                   color=['salmon', 'lightblue'])
plt.title("Heart Disease Frequency Per Chest Pain Type")
plt.xlabel("Chest Pain Type")
plt.ylabel("Amount")
plt.legend(["No Disease", "Disease"])
plt.xticks(rotation=0);
```



An interesting find is that for heart disease 1 and 2 (which are both suppose to be not related to the heart), we see that they both have very high numbers for the patients who have this chest pain and have a target value of 1.

We want to build a correlation matrix to tell us how related the independent variables are to one another. To make the visualization nicer, we can make a heatmap of our correlation matrix.

```
corr_matrix = df.corr()
fig, ax = plt.subplots(figsize=(15, 10))
ax = sns.heatmap(corr_matrix, annot=True,
                 linewidths=0.5, fmt=".2f", cmap="YlGnBu");
bottom, top = ax.get_ylim() # get current limit values
ax.set_ylim(bottom + 0.5, top - 0.5) # fix first/last row cut off
```



5.9.2 Modeling (Step 5)

Before we begin modeling, we must split the data into training and test sets.

```
np.random.seed(42) # reproducible results

X = df.drop('target', axis=1)
y = df['target']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
```

Referring back to the picture in section 5.3, we can see that KNeighborsClassifier and a RandomForestClassifier are the ideal choices. However, we will also include a LogisticRegression model since it is made for classification (not regression). The easiest way to test these models will be to put them in a dictionary, write a function to train/test them, and finally compare the scores for each model.

```
models = {"Logistic Regression": LogisticRegression(),
          "KNN": KNeighborsClassifier(),
          "Random Forest": RandomForestClassifier()}

def fit_and_score(models, X_train, X_test, y_train, y_test):
  """
  Fits and evaluates given machine learning models.
  models : a dict of different Scikit-Learn machine learning models
  """
  np.random.seed(42) # Set random seed
  model_scores = {} # Make a dictionary to keep model scores
  for name, model in models.items():
    model.fit(X_train, y_train) # Fit the model to the data
    model_scores[name] = model.score(X_test, y_test) # Evaluate append score
  return model_scores

model_scores = fit_and_score(models, X_train, X_test, y_train, y_test)
"""
{'Logistic Regression': 0.8852459016393442,
 'KNN': 0.6885245901639344,
 'Random Forest': 0.8360655737704918}
"""
```

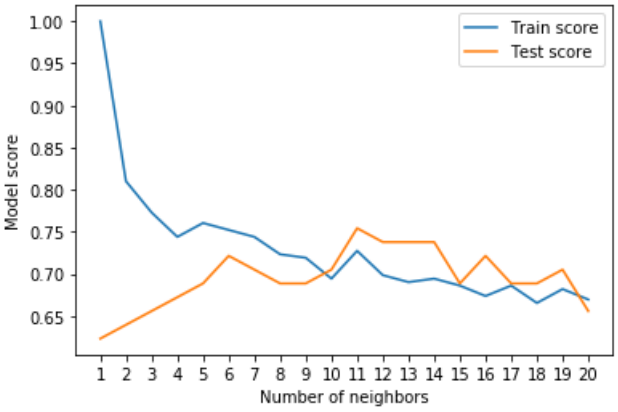
5.9.3 Tuning Hyperparameters

Our KNN classifier was our lowest score in the previous subsection, so lets see if we can improve this score by tuning the hyperparameters by hand.

```
train_scores = []
test_scores = []

neighbors = range(1, 21) # Create a list of differnt values for n_neighbors
knn = KNeighborsClassifier() # Setup KNN instance

for i in neighbors: # Loop through different n_neighbors
  knn.set_params(n_neighbors=i) # Set n_neighbors value
  knn.fit(X_train, y_train) # Fit the algorithm
  train_scores.append(knn.score(X_train, y_train)) # Update the training scores list
  test_scores.append(knn.score(X_test, y_test)) # Update the test scores list

plt.plot(neighbors, train_scores, label="Train score")
plt.plot(neighbors, test_scores, label="Test score")
plt.xticks(np.arange(1, 21, 1))
plt.xlabel("Number of neighbors")
plt.ylabel("Model score")
plt.legend()
```

```
print(f"Maximum KNN score on the test data: {max(test_scores)*100:.2f}%")
```

```
Maximum KNN score on the test data: 75.41%
```



We can see that a n_neighbor value of 11 gives us the highest score on the test data. We were able to go from 68.8% to 75.4% by only tuning one hyperparameter. However, even with the tuning it still has a lower score than LogisticRegression or the RandomForestClassifier. Because of this, we will discard the model and focus mainly on the other two.

Now we will tune the LogisticRegression and RandomForestClassifier models with RandomizedSearchCV.

```
# Create a hyperparameter grid for LogisticRegression model
log_reg_grid = {"C": np.logspace(-4, 4, 20),
                "solver": ["liblinear"]}

# Create a hyperparameter grid for RandomForestClassifier model
rf_grid = {"n_estimators": np.arange(10, 1000, 50),
           "max_depth": [None, 3, 5, 10],
           "min_samples_split": np.arange(2, 20, 2),
           "min_samples_leaf": np.arange(1, 20, 2)}

############################## Tune LogisticRegression ##############################
np.random.seed(42)

# Setup random hyperparameter search for LogisticRegression
rs_log_reg = RandomizedSearchCV(LogisticRegression(), # model to use
                                param_distributions=log_reg_grid, # hyper grid
                                cv=5, # 5 different data set splits
                                n_iter=20, # 20 different models
                                verbose=True) # 100 different fits

rs_log_reg.fit(X_train, y_train) # Fit random hyperparameter search model

rs_log_reg.best_params_ # {'solver': 'liblinear', 'C': 0.23357214690901212}
rs_log_reg.score(X_test, y_test) # 0.8852 (no increase in score)

############################ Tune RandomForestClassifier ############################
np.random.seed(42)

# Setup random hyperparameter search for RandomForestClassifier
rs_rf = RandomizedSearchCV(RandomForestClassifier(), # model to use
                           param_distributions=rf_grid, # hyper grid
                           cv=5, # 5 different data set splits
                           n_iter=20, # 20 different models
                           verbose=True) # 100 different fits

rs_rf.fit(X_train, y_train) # Fit random hyperparameter search model

rs_rf.best_params_ # {'n_estimators': 210, 'min_samples_split': 4,
                   #  'min_samples_leaf': 19, 'max_depth': 3}
rs_rf.score(X_test, y_test) # 0.8688 (increase of 0.032)
```

After the results above, the LogisticRegression model has the higher score and we will focus on that one. Now that we have used RandomizedSearchCV (which chooses hyperparameters from our grid at random), we will use GridSearchCV with our model (which exhaustively tries all combinations within our hyperparameter grid).

```
# Different hyperparameters for our LogisticRegression model
log_reg_grid = {"C": np.logspace(-4, 4, 30),
                "solver": ["liblinear"]}

# Setup grid hyperparameter search for LogisticRegression
gs_log_reg = GridSearchCV(LogisticRegression(),
                          param_grid=log_reg_grid, # 30 different combinations
                          cv=5, # 5 different data splits
                          verbose=True) # 150 total fits

gs_log_reg.fit(X_train, y_train) # Fit grid hyperparameter search model

gs_log_reg.best_params_ # {'C': 0.20433597178569418, 'solver': 'liblinear'}
gs_log_reg.score(X_test, y_test) # 0.8852 (again no increase)
```

5.9.4 Evaluating the Model

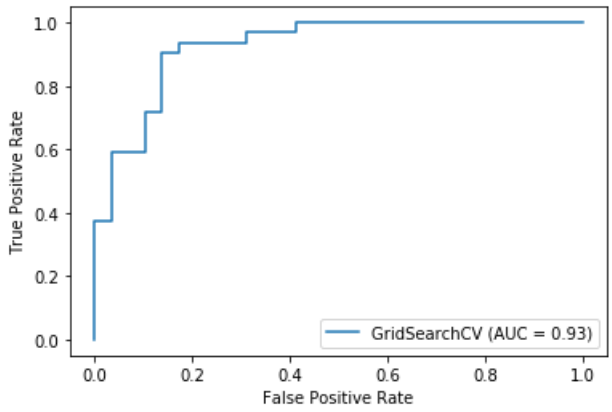
We want to evaluate our model beyond just accuracy. The first metric we want to see is a **ROC curve**, which is the true positive rate plotted against the false positive rate (we can use the sklearn built in function for this). A perfect ROC curve would get an AUC score of 1.

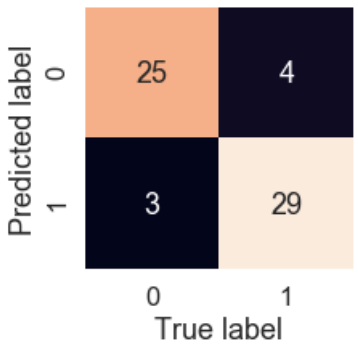
```
y_preds = gs_log_reg.predict(X_test) # Make predictions with tuned model

# Plot ROC curve and calculate and calculate AUC metric
plot_roc_curve(gs_log_reg, X_test, y_test)
```

Next we want to see a **confusion matrix**, but we can use a heatmap to make it more easily readable.

```
sns.set(font_scale=1.5)

def plot_conf_mat(y_test, y_preds):
  fig, ax = plt.subplots(figsize=(3, 3))
  ax = sns.heatmap(confusion_matrix(y_test, y_preds), annot=True,   cbar=False)
  plt.xlabel("True label")
  plt.ylabel("Predicted label")

  bottom, top = ax.get_ylim()
  ax.set_ylim(bottom + 0.5, top - 0.5)

plot_conf_mat(y_test, y_preds)
```

Below are the visualizations for the above code (ROC curve on left, confusion matrix on right):





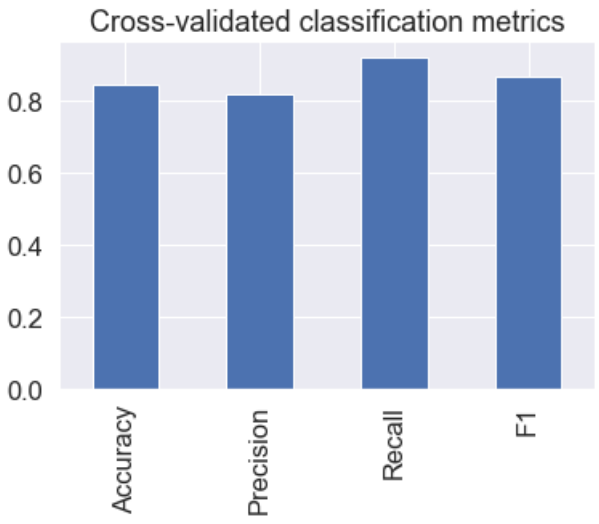
Now we will look at a **classification report**, which gives us precision, recall, and f1 score.

- *Precision* : indicates the proportion of positive (1) predictions that were actually correct.
- *Recall* : indicates the proportion of actual positives (1) that were correctly classified.
- *f1* : a combination of recall and precision (should be near or equal to 1).

```
classification_report(y_test, y_preds)
#              precision    recall  f1-score
# 0            0.89         0.86    0.88
# 1            0.88         0.91    0.89
# accuracy                          0.89
```

Note that this report is only used on the original data split. We need to use **cross validation** to find k-fold cross validation scores for all of the metrics listed in the classification report. We will need to create a new LogisticRegression model that is not already trained on data (but use the best hyperparameters found earlier).

```
# Create a new classifier with best parameters
gs_log_reg.best_params_ # {'C': 0.20433597178569418, 'solver': 'liblinear'}
clf = LogisticRegression(C=0.20433597178569418, solver="liblinear")

# Cross-validated accuracy
cv_acc = cross_val_score(clf, X, y, cv=5, scoring="accuracy")
np.mean(cv_acc) # 0.8446994535519124

# Cross-validated precision
cv_precision = cross_val_score(clf, X, y, cv=5, scoring="precision")
np.mean(cv_precision) # 0.8207936507936507

# Cross-validated recall
cv_recall = cross_val_score(clf, X, y, cv=5, scoring="recall")
np.mean(cv_recall) # 0.9212121212121213

# Cross-validated f1-score
cv_f1 = cross_val_score(clf, X, y, cv=5, scoring="f1")
np.mean(cv_f1) # 0.8673007976269721

# Visualize cross-validated metrics
cv_metrics = pd.DataFrame({"Accuracy": cv_acc, "Precision": cv_precision,
                           "Recall": cv_recall, "F1": cv_f1}, index=[0])

cv_metrics.T.plot.bar(title="Cross-validated classification metrics", legend=False);
```

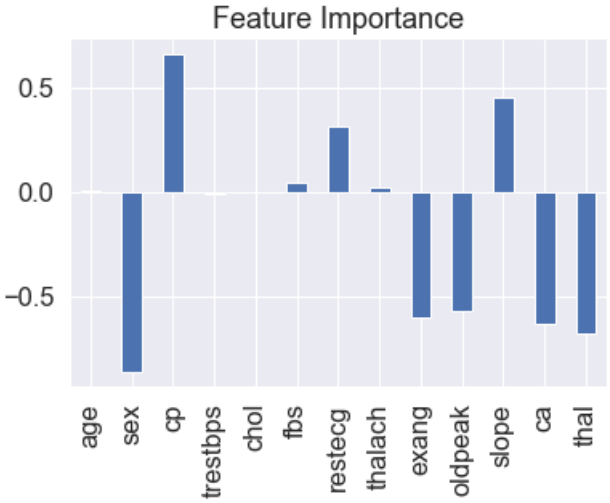


We see that our recall (the proportion of positive class (1) values were correctly predicted) is the highest score in our cross validation evaluation. In the future, it would be helpful to turn the above code into a function that would output the scores and create a graph to visualize them.

5.9.5 Find Most Important Features

Feature importance is finding which features contributed most to the outcomes of the model and how did they contribute. For our model, which characteristics contribute most to heart disease? Note that finding this is different for each machine learning model (search for "*model name*" feature importance).

```
# Fit an instance of LogisticRegression (with the best hyperparameters from earlier)
clf = LogisticRegression(C=0.20433597178569418, solver="liblinear")
clf.fit(X_train, y_train)

clf.coef_ # only used for LogisticRegression
# [[ 0.00316727, -0.86044582,  0.66067073, -0.01156993, -0.00166374,
#    0.04386131,  0.31275787,  0.02459361, -0.60413038, -0.56862852,
#    0.45051617, -0.63609863, -0.67663375]]

# Match coef's of features to columns from data frame (into an dict)
feature_dict = dict(zip(df.columns, list(clf.coef_[0])))

# Visualize feature importance
feature_df = pd.DataFrame(feature_dict, index=[0])
feature_df.T.plot.bar(title="Feature Importance", legend=False);
```



We can see how much each of these independent variables contribute to finding the target variable. For example, as the value for sex increases then the ratio value for target decrease (since it is a negative correlation). It goes from a 3:1 ratio to a 1:2 ratio. For slope, as it increases so should the target value ratio.

```
pd.crosstab(df["sex"], df["target"])
pd.crosstab(df["slope"], df["target"])
```

target	0	1
sex		
0	24	72
1	114	93

target	0	1
slope		
0	12	9
1	91	49
2	35	107

Step 6: Experimentation

If you haven't hit your evaluation metric yet, ask yourself:

- Could you collect more data?
- Could you try a better model? Like CatBoost or XGBoost?
- Could you improve the current models? (beyond what we've done so far)
- If your model is good enough (you have hit your evaluation metric) how would you export it and share it with others?

NOTE: Search for "'classification dataset" to find new data that you can use for machine learning models.

5.10 Milestone Project 2 (Regression)

This is a **time series data set**, meaning that we will be using sales from the past to predict sales for the future. We will use the bulldozer datasets for this project. Following the typical work flow that we covered in section one, lets map out our data modeling steps.

Step 1: Problem Definition
How well can we predict the future sale price of a bulldozer, given its characteristics and previous examples of how much similar bulldozers have been sold for?

Step 2: Data
The data is downloaded from the Kaggle Bluebook for Bulldozers competition. Note that there are 3 main datasets already split for us:

1) Train.csv is the training set, which contains data through the end of 2011.
2) Valid.csv is the validation set, which contains data from January 1, 2012 - April 30, 2012
3) Test.csv is the test set, which contains data from May 1, 2012 - November 2012.

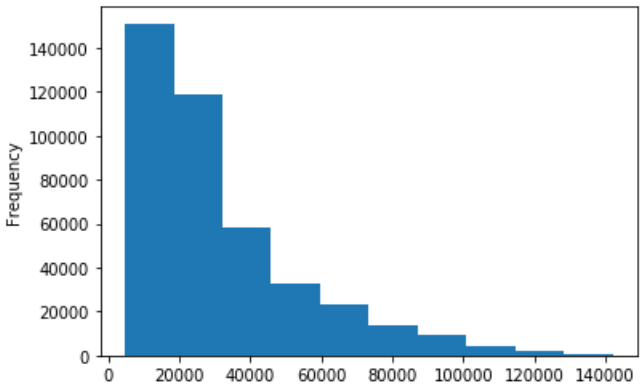
Step 3: Evaluation
The evaluation metric for this competition is the RMSLE (root mean squared log error) between the actual and predicted auction prices. Note that the goal for most regression evaluation metrics is to minimize the error. For example, our goal for this project will be to build a machine learning model which minimizes RMSLE.

Step 4: Features
Kaggle provides a data dictionary detailing all of the features of the dataset. You can view this data dictionary on Google Sheets.

5.10.1 Exploring the Data (EDA)

Since the data frames that we read in live in RAM, we need to set *low_memory=False* so that it can read in the file without an error. For plotting, we must use subsections (approx 1000 observations) since plotting over 400,000 observations in one plot would not only take a long time but also be unreadable.

```
# Import Training and Validation Sets
df = pd.read_csv('data/bluuebook-for-bulldozers/TrainAndValid.csv', low_memory=False)

df.info()
# 412,698 rows and 53 columns
# dtypes: float64 (3), int64 (5), object (45)
# memory usage: 166.9+ MB (very large file)

df.SalePrice.plot.hist()
```



Plotting the distribution of our target varaible gives us an idea of where most bulldozers prices are centered around. As we can see, many are sold for below $20,000. We can also try plotting this target variable against other important variables in other graphs to get a better understanding.

When we work with time series data, we want to enrich the time & date component as much as possible. We can do that by telling pandas which of our columns has dates in it using the *parse_dates* parameter, known as **parsing dates**.

```
# Import data again but this time parse dates
df = pd.read_csv("data/bluebook-for-bulldozers/TrainAndValid.csv",
                 low_memory=False, parse_dates=["saledate"])

df.saledate.dtype # '<M8[ns]' (same as datetime64[ns]) which is a datetime object
# ex: turns MM/DD/YYYY into YYYY-MM-DD
```

When working with time series data, it may be helpful to **sort** the DataFrame by date. We also may want to make a **copy** of the original DataFrame so that when we manipulate it, we still have the original data if we need it.

```
# Sort DataFrame in date order
df.sort_values(by=["saledate"], inplace=True, ascending=True)

# Make a copy of the original DataFrame to perform edits on
df_tmp = df.copy()
```

We can enrich our dataset with **feature engineering** (step 4), which means we create new features or change already existing columns to enhance our dataset. This is valuable to most time series projects that you may work on.

We can use the pandas DatetimeIndex attributes to access specific values within the datetime object of the saledate column.

```
# Create new columns from attritubes of the saledate column
df_tmp["saleYear"] = df_tmp.saledate.dt.year
df_tmp["saleMonth"] = df_tmp.saledate.dt.month
df_tmp["saleDay"] = df_tmp.saledate.dt.day
df_tmp["saleDayOfWeek"] = df_tmp.saledate.dt.dayofweek
df_tmp["saleDayOfYear"] = df_tmp.saledate.dt.dayofyear

# Now we've enriched our DataFrame with date time features, we can remove 'saledate'
df_tmp.drop("saledate", axis=1, inplace=True)
```

5.10.2 Converting Strings to Categories

We have a lot of missing data values in our DataFrame, along with many columns being objects instead of numerical data types. In order to use this data one a ML model, we must convert everything to numeric along with no missing values before fitting the model.

We can begin by **converting strings to categories**, more specifically pandas categories. The .items() function will treat our DataFrame as a dictionary, with column names being the 'label' variable and the values within these columns being the 'content' variable.

```
# This will turn all of the string value into category values
for label, content in df_tmp.items():
  if pd.api.types.is_string_dtype(content): # returns T/F if column is a string
    df_tmp[label] = content.astype("category").cat.as_ordered()
```

By using the pandas Categories, it has turned all of the strings into the form of numbers, which we can now access and have been given specific codes based on their position in the *.cat.categories* attribute.

```
df_tmp.state.cat.categories # returns an array of ordered values
df_tmp.state.cat.codes # returns numeric code for given array position
```

Before we fill missing values and make new changes, we want to save this data so that we can import the manipulated data and start from the current position if we start a new Jupyter Notebook.

```
# Export current tmp dataframe
df_tmp.to_csv("data/bluebook-for-bulldozers/train_tmp.csv", index=False)

# Import preprocessed data
df_tmp = pd.read_csv("data/bluebook-for-bulldozers/train_tmp.csv", low_memory=False)
```

5.10.3 Filling Missing Values

We will begin by filling the missing **numerical values** first. We can create a loop to find the numerical columns using the pandas categories attributes.

```
# Check for which numeric columns have null values
for label, content in df_tmp.items():
  if pd.api.types.is_numeric_dtype(content):
    if pd.isnull(content).sum():
      print(label) # auctioneerID, MachineHoursCurrentMeter

# Fill numeric rows with the median
for label, content in df_tmp.items():
  if pd.api.types.is_numeric_dtype(content):
    if pd.isnull(content).sum():
      # Add a binary column which tells us if the data was missing or not
      df_tmp[label+"_is_missing"] = pd.isnull(content)
      # Fill missing numeric values with median
      df_tmp[label] = content.fillna(content.median())
```

Note that the *_is_missing* column will have a corresponding value (True or False) to the original column that tells us if the current value was already in the DataFrame or if it was filled from the loop above. We will also use the median instead of the mean value because it is more robust. If we had used the mean, it could be much for sensitive to outliers and not be an accurate measure for our model.

```
# Check to see how many examples were missing
df_tmp.auctioneerID_is_missing.value_counts()
# False: 392562 (values that already existed)
# True: 20136 (missing values filled in from the above loop)
```

Now that we have filled all missing numerical values, we will fill the missing **categorical values**. We can use a similar loop to the one above, but check for *if not numerical* instead. We will add 1 to the codes since it assigns missing values = -1, but if we +1 then all missing values will have a value = 0.

```
# Turn categorical variables into numbers and fill missing
for label, content in df_tmp.items():
  if not pd.api.types.is_numeric_dtype(content):
    # Add binary column to indicate whether sample had missing value
    df_tmp[label+"_is_missing"] = pd.isnull(content)
    # Turn categories into numbers and add +1
    df_tmp[label] = pd.Categorical(content).codes+1

df_tmp.info() # 103 columns (increased from original 53 columns)
# bool(46), float64(3), int16(4), int64(10), int8(40)

df_tmp.isna().sum() # prints 0 for all columns
```

Note that filling the above missing categorical values turned the columns into numerical values since they are now codes that correspond to the string values.

5.10.4 Modeling (Step 5)

This is the first time we are working with a dataset that has hundreds of thousands of observations. Our goal is to minimize the amount of time between tests, but if we test on the entire set we will not be able to do this (we will first time how long it takes to do this). Remember that the .score() method returns R^2 for regression.

```
%%time # It took around 7 mins to go through 412698 observations

model = RandomForestRegressor(n_jobs=-1, random_state=42)
model.fit(df_tmp.drop("SalePrice", axis=1), df_tmp["SalePrice"])
model.score(df_tmp.drop("SalePrice", axis=1), df_tmp["SalePrice"]) # 0.9875468079
```

However, the above score metric is not reliable because we trained it and tested it on the same data sets. In other words, the training set was mixed with the testing set, thus explaining the near 99% score. We must **split the data** into training and validation sets. Remember from step 1 that the training set is up to Dec 31, 2011 while the validation set is Jan 1, 2012 - Apr 30, 2012. A few notes:

- Encode/transform all categorical variables of your data for training/test set at the same time.
- Split your data (into train/test).
- Fill the training set and test set numerical values separately.

```
# Split data into training and validation
df_val = df_tmp[df_tmp.saleYear == 2012]
df_train = df_tmp[df_tmp.saleYear != 2012]
len(df_val), len(df_train) # (11573, 401125)

# Split data into X & y
X_train, y_train = df_train.drop("SalePrice", axis=1), df_train.SalePrice
X_valid, y_valid = df_val.drop("SalePrice", axis=1), df_val.SalePrice
```

The **evaluation metric** for this data set is the RMSLE (Root mean square log error) . We will have to create our own custom evaluation functions to help us with our model. Note that if validation model has a higher score, that means we are over training (we want to see slightly worse metric since it is a smaller set).

```
from sklearn.metrics import mean_squared_log_error, mean_absolute_error, r2_score

def rmsle(y_test, y_preds): # y_test same as y_true (used for train set)
  """
  Caculates root mean squared log error between predictions and true labels.
  """
  return np.sqrt(mean_squared_log_error(y_test, y_preds))

# Create function to evaluate model on a few different levels
def show_scores(model):
  train_preds = model.predict(X_train) # use train data from previous code block
  val_preds = model.predict(X_valid )# use valid data from previous code block
  scores = {"Training MAE": mean_absolute_error(y_train, train_preds),
            "Valid MAE": mean_absolute_error(y_valid, val_preds),
            "Training RMSLE": rmsle(y_train, train_preds),
            "Valid RMSLE": rmsle(y_valid, val_preds),
            "Training R^2": r2_score(y_train, train_preds),
            "Valid R^2": r2_score(y_valid, val_preds)}
  return scores
```

Remember, we want to decrease the time between experiments. When we trained the model on the entire data set, it took about 7 minutes to complete. This is too long, so ideally we would want to train on a subset of the data. This idea is known as **reducing data**.

The model would see the length of the training data (401125) multiplied by the number of estimators (default is 100) which totals to over 40 million different models, hence the 7 minute training time. However, if we reduce the max_samples, then we can take this down to only 1 million different models.

```
# Change max_samples value
model = RandomForestRegressor(n_jobs=-1, random_state=42, max_samples=10000)

%%time
# Cutting down the max number of samples each estimator can improved training time
model.fit(X_train, y_train)
# 16.6s to fit the model to a subset of 10000 samples of data

show_scores(model)
# {'Training MAE': 5561.2988092240585,
#  'Valid MAE': 7177.26365505919,
#  'Training RMSLE': 0.257745378256977,
#  'Valid RMSLE': 0.29362638671089003,
#  'Training R^2': 0.8606658995199189,
#  'Valid R^2': 0.8320374995090507}
```

Now that we have a base evaluation metric for the model, we can use RandomizedSearchCV to **tune the hyperparameters** in order to try and increase these values. Since we have a wide range of parameters to chose from, we want to use RandomizedSearchCV instead of GridSearchCV

```
%%time
from sklearn.model_selection import RandomizedSearchCV

# Different RandomForestRegressor hyperparameters
rf_grid = {"n_estimators": np.arange(10, 100, 10),
           "max_depth": [None, 3, 5, 10],
           "min_samples_split": np.arange(2, 20, 2),
           "min_samples_leaf": np.arange(1, 20, 2),
           "max_features": [0.5, 1, "sqrt", "auto"],
           "max_samples": [10000]}

# Instantiate RandomizedSearchCV model
rs_model = RandomizedSearchCV(RandomForestRegressor(n_jobs=-1, random_state=42),
                              param_distributions=rf_grid, n_iter=2, cv=5,
                              verbose=True)

# Fit the RandomizedSearchCV model
rs_model.fit(X_train, y_train) # 1 min 49 seconds

rs_model.best_params_
# {'n_estimators': 90,
#  'min_samples_split': 16,
#  'min_samples_leaf': 11,
#  'max_samples': 10000,
#  'max_features': 'auto',
#  'max_depth': 10}

# Evaluate the RandomizedSearch model
show_scores(rs_model)
# {'Training MAE': 6633.714300615716,
#  'Valid MAE': 8074.634589086979,
#  'Training RMSLE': 0.2967133662928448,
#  'Valid RMSLE': 0.32142476459239144,
#  'Training R^2': 0.807156908286598,
#  'Valid R^2': 0.7827628271827518}
```

In a model trained with 100 iterations of RandomizedSearchCV, the best parameters were found and the model below will have there parameters found. Note that random_state is similar to np.random.seed(), so it is just used to create reproducible results.

```
%%time

# Most ideal hyperparamters
ideal_model = RandomForestRegressor(n_estimators=40, min_samples_leaf=1,
                                    min_samples_split=14, max_features=0.5,
                                    n_jobs=-1, max_samples=None, random_state=42)

# Fit the ideal model
ideal_model.fit(X_train, y_train) # took around 1 min 14 seconds

# Scores for ideal_model (trained on all the data)
show_scores(ideal_model)
# {'Training MAE': 2952.0893487566345,
#  'Valid MAE': 5959.9235776482565,
#  'Training RMSLE': 0.14457060318530182,
#  'Valid RMSLE': 0.2465443247528795,
#  'Training R^2': 0.9589437998919624,
#  'Valid R^2': 0.8817886038953688}
```

Next we will want to **preprocess the test data** so that it is in the same format as our training dataset. We will create a function to do this for us. Our training data set also has 102 columns, where our current test set only has 51. We must make these the same length in order to use them on our model.

```
# Import the test data
df_test = pd.read_csv("data/bluebook-for-bulldozers/Test.csv", low_memory=False,
                      parse_dates=["saledate"])

def preprocess_data(df):
  """
  Performs transformations on df and returns transformed df.
  """
  df["saleYear"] = df.saledate.dt.year
  df["saleMonth"] = df.saledate.dt.month
  df["saleDay"] = df.saledate.dt.day
  df["saleDayOfWeek"] = df.saledate.dt.dayofweek
  df["saleDayOfYear"] = df.saledate.dt.dayofyear

  df.drop("saledate", axis=1, inplace=True)

  # Fill the numeric rows with median
  for label, content in df.items():
    if pd.api.types.is_numeric_dtype(content):
      if pd.isnull(content).sum():
        # Add a binary column which tells us if the data was missing or not
        df[label+"_is_missing"] = pd.isnull(content)
        # Fill missing numeric values with median
        df[label] = content.fillna(content.median())

    # Fill categorical missing data and turn categories into numbers
    if not pd.api.types.is_numeric_dtype(content):
      df[label+"_is_missing"] = pd.isnull(content)
      # We add +1 to the category code because pd encodes missing categories as -1
      df[label] = pd.Categorical(content).codes+1
  return df

# Process the test data
df_test = preprocess_data(df_test)
```

Currently, if we were to try and make predictions on our test data set, we would get the error "Number of features of the model must match the input. Model n_features is 102 and input n_features is 101". We need to make the sets have the **same number of features**.

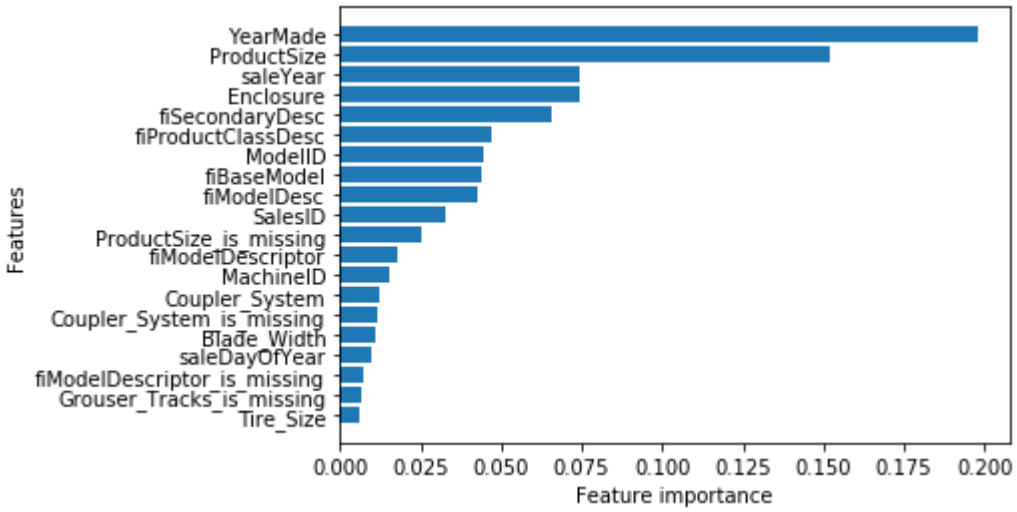
```
# We can find how the columns differ using sets
set(X_train.columns) - set(df_test.columns) # {'auctioneerID_is_missing'}

# Manually adjust df_test to have auctioneerID_is_missing column
df_test["auctioneerID_is_missing"] = False
```

The test data did not have any missing values, hence why the above column was never created. But now that we have created it, the number of features are matching and we can use our model to make price **predictions**. We will want to create a DataFrame and export it as a csv file.

```
# Make predictions on the test data
test_preds = ideal_model.predict(df_test)

# Format predictions into a DataFrame
df_preds = pd.DataFrame()
df_preds["SalesID"] = df_test["SalesID"]
df_preds["SalesPrice"] = test_preds

# Export prediction data
df_preds.to_csv("data/bluebook-for-bulldozers/test_predictions.csv", index=False)
```

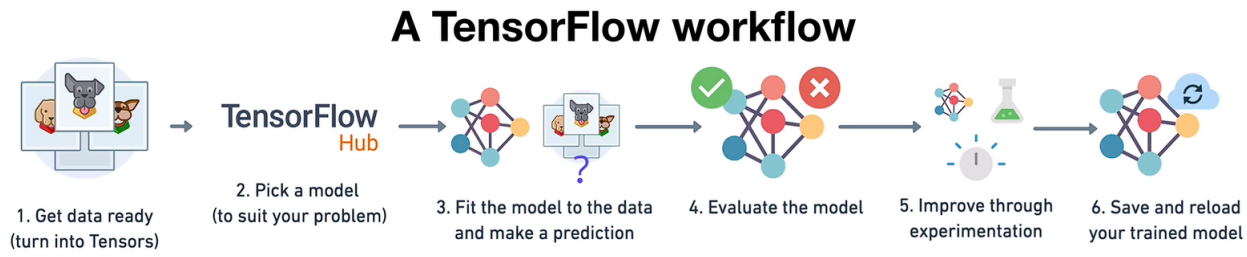
Finally, **feature importance** seeks to figure out which different attributes of the data were most importance when it comes to predicting the *target variable* (SalePrice).

```
# Helper function for plotting feature importance
def plot_features(columns, importances, n=20):
  df = (pd.DataFrame({"features": columns,
                      "feature_importances": importances})
        .sort_values("feature_importances", ascending=False) # sort high to low
        .reset_index(drop=True))

  # Plot the dataframe (only top 20 of the total 102 features)
  fig, ax = plt.subplots()
  ax.barh(df["features"][:n], df["feature_importances"][:20])
  ax.set_ylabel("Features")
  ax.set_xlabel("Feature importance")
  ax.invert_yaxis()

plot_features(X_train.columns, ideal_model.feature_importances_)
```



6 Neural Networks: Deep/Transfer Learning & TensorFlow 2

6.1 Workflow & Setup



1) Get our **unstructured data** ready by turning it into **Tensors**.
2) Pick a model from TensorFlow Hub.
3) Fit the model to find patterns and make predictions.
4) Evaluate the performance of our model.
5) Tune our model to try and improve our evaluation scores.
6) Save and use model to make predictions on custom data.

For this section, we will focus on a **multi-class classification** model (more than two options) where we try and identify the breed of a dog from a given image. We will be using Google Colab for this instead of Jupyter Notebooks. If we upload a .zip file, we can unpack it using the following command in Colab:

```
# Unzip the uploaded data into Google Drive (-d is the destination)
!unzip "drive/My Drive/Dog Vision/dog-breed-identification.zip" -d "drive/My Drive/
        Dog Vision/"
```

Using a GPU
Go to *Runtime* → *Change Runtime Type* → *Hardware Accelerator* → *GPU*.

Step 1: Problem
Identify the breed of a dog given an image of a dog.

Step 2: Data
The data we're using is from Kaggle's dog breed identification competition.

Step 3: Evaluation
The evaluation is a file with prediction probabilities for each dog breed of each test image.

Step 4: Features
- We're dealing with images (unstructured data) so it's probably best we use deep/transfer learning.
- There are 120 breeds of dogs (this means there are 120 different classes).
- There are around 10,000+ images in the training set (these images have labels).
- There are around 10,000+ images in the test set (these images have no labels).

Import TensorFlow 2.x and Hub

```
# Import necessary tools
import tensorflow as tf
import tensorflow_hub as hub
print("TF version:", tf.__version__) # TF version: 2.1.0
print("TF Hub version:", hub.__version__) # TF Hub version: 0.7.0
```

6.2 Step 1: Preparing Data

With all machine learning models, our data has to be in numerical format. So we must start by turning our images into **Tensors** (numerical representations).

```
# Import data and check labels
import pandas as pd
labels_csv = pd.read_csv("drive/My Drive/Dog Vision/labels.csv")
print(labels_csv.describe())
# 10222 count for both columns (id and breed)
# 10222 unique values for id, 120 unique values for breed

# How many images are there of each breed?
labels_csv["breed"].value_counts().plot.bar(figsize=(20, 10))
```

6.2.1 Getting Images and Their Labels

We want to create a list containing the **file path names** for the images we want to use in our model. We can do this from combining the csv file column 'id' and the location of the images within the Google Drive workspace.

```
# Let's manually view an image in our folder
from IPython.display import Image
Image("drive/My Drive/Dog Vision/train/001513dfcb2ffafc82cccf4d8bbaba97.jpg")

# Create pathnames from image ID's (similar to the one above)
filenames = ["drive/My Drive/Dog Vision/train/" + fname + ".jpg" for fname
             in labels_csv["id"]]
```

Now that we have our image file paths in a list, we need to **prepare our labels** by turning them from strings into numbers. We will also want to find the number of unique breeds (similar to classes) that we will be able to classify our dog breeds into.

```
# Convert breed labels into NumPy array
import numpy as np
labels = labels_csv["breed"].to_numpy()
len(labels) # 10222

# Find the unique label values
unique_breeds = np.unique(labels)
len(unique_breeds) # 120

# Turn every label into a boolean array
boolean_labels = [label == unique_breeds for label in labels]
```

Now that we have our file paths and their labels are in numeric format, we want to create our own **validation set** and split the data. We can use train_test_split since the entire csv we are currently working with is only for training, so we can use part of it as a validation set. We will want to use a subset so that we are not training on 10,000 images (time consuming).

```
# Set number of images to use for experimenting
NUM_IMAGES = 1000

# Split them into training and validation of total size NUM_IMAGES
from sklearn.model_selection import train_test_split
X_train, X_val, y_train, y_val = train_test_split(X[:NUM_IMAGES], y[:NUM_IMAGES],
                                                  test_size=0.2, random_state=42)

len(X_train), len(y_train), len(X_val), len(y_val) # (800, 800, 200, 200)
```

6.2.2 Turning Images into Tensors

To preprocess our images into Tensors we're going to write a function which does a few things:

1. Take an image file path as input.
2. Use TensorFlow to read the file and save it to a variable, image.
3. Turn our image (a jpg) into Tensors.
4. Normalize our image (convert color channel values from from 0-255 to 0-1).
5. Resize the image to be a shape of (224, 224).
6. Return the modified image.

Before we write this function, lets see what importing an image into an array looks like. The values within the image array below will be between 0 and 255 (color range for RGB).

```
# Example of how to convert image to NumPy array
from matplotlib.pyplot import imread
image = imread(filenames[42])
image.shape # (257, 350, 3)

IMG_SIZE = 224 # Define image size

# Create a function for preprocessing images
def process_image(image_path, img_size=IMG_SIZE):
  """
  Takes an image file path and turns the image into a Tensor.
  """
  image = tf.io.read_file(image_path) # Read in an image file into a Tensor string
  # Turn the jpeg image into numerical Tensor with 3 colour channels (RGB)
  image = tf.image.decode_jpeg(image, channels=3)
  # Convert the colour channel values from 0-255 to 0-1 values (normalize)
  image = tf.image.convert_image_dtype(image, tf.float32)
  # Resize the image to our desired value (224, 224)
  image = tf.image.resize(image, size=[IMG_SIZE, IMG_SIZE])

  return image
```

6.2.3 Turning Data into Batches

A **batch** is a small portion of our data, typically a size of 32 (default for most deep learning models). We need batches because if we are processing thousands of images at once, it might not all fit into the GPU memory.

In order to use TensorFlow effectively, we need our data in the format of a **Tensor tuples**, which looks like: (image, label). We will want to write a function to do this for us. The image will be in the form of a Tensor from the above process_image function, and our label will be a Boolean array that has already been created.

```
# Create a simple function to return a tuple (image, label)
def get_image_label(image_path, label):
  """
  Takes an image file path name and the assosciated label,
  processes the image and returns a tuple of (image, label).
  """
  image = process_image(image_path)
  return image, label # return the Tensor tuple
```

We now need to write a function that can take our data (X & y) and turn it into batches of size 32.

```
BATCH_SIZE = 32 # Define the batch size, 32 is a good start

# Create a function to turn data into batches
def create_data_batches(X, y=None, batch_size=BATCH_SIZE, valid_data=False,
                        test_data=False):
  """
  Creates batches of data out of image (X) and label (y) pairs.
  Shuffles the data if it's training data but doesn't shuffle if it's validation data
  Also accepts test data as input (no labels).
  """
  if test_data: # If the data is a test dataset, we probably don't have labels
    print("Creating test data batches...")
    data = tf.data.Dataset.from_tensor_slices((tf.constant(X))) # only filepaths
    data_batch = data.map(process_image).batch(BATCH_SIZE)
    return data_batch

  elif valid_data: # If the data is a valid dataset, we don't need to shuffle it
    print("Creating validation data batches...")
    data = tf.data.Dataset.from_tensor_slices((tf.constant(X), # filepaths
                                               tf.constant(y))) # labels
    data_batch = data.map(get_image_label).batch(BATCH_SIZE)
    return data_batch

  else:
    print("Creating training data batches...")
    # Turn filepaths and labels into Tensors
    data = tf.data.Dataset.from_tensor_slices((tf.constant(X), tf.constant(y)))

    # Shuffling pathnames and labels before mapping image processor function is
    # faster than shuffling images
    data = data.shuffle(buffer_size=len(X))

    # Create (image, label) tuples (this also turns the image path into a
    # preprocessed image)
    data = data.map(get_image_label)

    # Turn the training data into batches
    data_batch = data.batch(BATCH_SIZE)
  return data_batch

# Create training and validation data batches
train_data = create_data_batches(X_train, y_train)
val_data = create_data_batches(X_val, y_val, valid_data=True)

# Create a function for viewing images in a data batch
import matplotlib.pyplot as plt
def show_25_images(images, labels):
  """
  Displays a plot of 25 images and their labels from a data batch.
  """
  plt.figure(figsize=(10, 10)) # Setup the figure
  # Loop through 25 (for displaying 25 images)
  for i in range(25):
    ax = plt.subplot(5, 5, i+1) # Create subplots (5 rows, 5 columns, index+1)
    plt.imshow(images[i]) # Display an image
    plt.title(unique_breeds[labels[i].argmax()]) # Add the image label as the title
    plt.axis("off") # Turn the grid lines off

# Now let's visualize the data in a training batch
train_images, train_labels = next(train_data.as_numpy_iterator())
show_25_images(train_images, train_labels)
```

6.3 Step 2: Building a Deep Learning Model

Before we build a model, there are a few things we need to define:
- The input shape (our images shape, in the form of Tensors) to our model.
- The output shape (image labels, in the form of Tensors) of our model.
- The URL of the model we want to use from TensorFlow Hub - click here.

Using a URL for our model is **Transfer Learning** since we are using a model that is already created and we are simply just fitting our data to it. Some resources to find models are: TensorFlow Hub, PyTorch Hub, and Papers with Code. We will access our model through **TensorFlow Hub**.

```
# Setup input shape to the model
INPUT_SHAPE = [None, IMG_SIZE, IMG_SIZE, 3] # batch, height, width, color channels

# Setup output shape of our model
OUTPUT_SHAPE = len(unique_breeds)

# Setup model URL from TensorFlow Hub
MODEL_URL = "https://tfhub.dev/google/imagenet/mobilenet_v2_130_224/classification/4"

INPUT_SHAPE # [None, 224, 224, 3]
```

Now we've got our inputs/outputs and model ready to go. Let's put them together into a **Keras deep learning model**. Knowing this, let's create a function which does the following (click here for steps):
- Takes the input shape, output shape and the model we've chosen as parameters.
- Defines the layers in a Keras model in sequential fashion (do this first, then this, then that).
- Compiles the model (says it should be evaluated and improved).
- Builds the model (tells the model the input shape it'll be getting).
- Returns the model.

```
# Create a function which builds a Keras model
def create_model(input_shape=INPUT_SHAPE, output_shape=OUTPUT_SHAPE, model_url=
  MODEL_URL):
  print("Building model with:", MODEL_URL)

  # Setup the model layers
  model = tf.keras.Sequential([hub.KerasLayer(MODEL_URL), # Layer 1 (input layer)
                               tf.keras.layers.Dense(units=OUTPUT_SHAPE,
                                                     activation="softmax")
                                                     ]) # Layer 2 (output layer)
  # Compile the model
  model.compile(loss=tf.keras.losses.CategoricalCrossentropy(),
                optimizer=tf.keras.optimizers.Adam(),
                metrics=["accuracy"])

  # Build and return the model
  model.build(INPUT_SHAPE) # size of images MobileNetV2 was trained on
  return model

model = create_model()
model.summary()
```

```
Building model with: https://tfhub.dev/google/imagenet/mobilenet_v2_130_224/classification/4
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
keras_layer (KerasLayer)     multiple                  5432713
_________________________________________________________________
dense (Dense)                multiple                  120240
=================================================================
Total params: 5,552,953
Trainable params: 120,240
Non-trainable params: 5,432,713
_________________________________________________________________
```

The **Setup the model layers** step above has the following explanation:

1) We create a Keras model running in sequential fashion.
2) The first layer will be the model URL (MobileNetV2 architecture which is implemented for us since we are using Transfer Learning).
3) Then we output an array of size 120 (output_shape) and use the *softmax* activation to convert the patterns into a range of (0,1) (note that the values will add up to one and the largest value will be our label).

The **Compile the model** step above has the following explanation:

1) Loss is a measure of how well the model is guessing (higher loss means worse prediction)
2) The Adam optimizer is a general optimizer that performs well on most models. It helps guide the loss function to minimize its value and improve its guesses.
3) The metric (accuracy) is how well our model is predicting the correct image label.

A general guide to **choosing activation/loss functions** are:

- Binary Classification : Sigmoid Activation, Binary Crossentropy Loss.
- Multi-class Classification : Softmax Activation, Categorical Crossentropy Loss.

Summary: Our model was trained on the ImageNet dataset, which contains approximately 14 million images. All the patterns that the MobileNetV2 model has learned from these images (approx 5 million patterns as seen the the summary above) can be utilized to help find patterns in our own 120,240 parameter (Dense layer above). Each layer is a model that has found patterns and are layered on top of each other (hence the name Deep Learning).

6.3.1 Creating callbacks

Callbacks are helper functions a model can use during training to do such things as save its progress, check its progress or stop training early if a model stops improving. We'll create two callbacks:

1) *TensorBoard* which helps track our models progress.
2) *Early stopping* which prevents our model from training for too long.

To setup a TensorBoard callback, we need to do 3 things:

1. Load the TensorBoard notebook extension.
2. Create a TensorBoard callback which is able to save logs to a directory and pass it to our model's 'fit()' function.
3. Visualize our models training logs with the '%tensorboard' magic function (after model training).

```
# Load TensorBoard notebook extension
%load_ext tensorboard

import datetime

# Create a function to build a TensorBoard callback
def create_tensorboard_callback():
  # Create a log directory for storing TensorBoard logs
  logdir = os.path.join("drive/My Drive/Dog Vision/logs",
                        # The logs get tracked whenever we run an experiment
                        datetime.datetime.now().strftime("%Y%m%d-%H%M%S"))
  return tf.keras.callbacks.TensorBoard(logdir)
```

Early stopping helps stop our model from overfitting by stopping training if a certain evaluation metric stops improving. Patience is the number of epoch's to wait before stopping if no improvements are made.

```
# Create early stopping callback
early_stopping = tf.keras.callbacks.EarlyStopping(monitor="val_accuracy",
                                                  patience=3)
```

6.4 Step 3: Fit Model and Make Predictions

6.4.1 Training a Deep Neural Network (on a data subset)

Our first model is only going to train on 1000 images, to make sure everything is working. Training on the entire dataset can be time consuming and if any errors occur, it is not efficient to keep retraining on the entire dataset. The **number of epochs** is the chances we give our model to go through the training set and find any patterns.

```
NUM_EPOCHS = 100 #@param {type:"slider", min:10, max:100, step:10}

# Check to make sure we're still running on a GPU
print("GPU", "available." if tf.config.list_physical_devices("GPU") else "not
      available")
```

Let's create a function which trains a model:

1) Create a model using 'create_model()'
2) Setup a TensorBoard callback using 'create_tensorboard_callback()'
3) Call the 'fit()' function on our model passing it the training data, validation data, number of epochs to train for ('NUM_EPOCHS'), and the callbacks we'd like to use
4) Return the model

```
# Build a function to train and return a trained model
def train_model():
  """
  Trains a given model and returns the trained version.
  """
  # Create a model
  model = create_model()

  # Create new TensorBoard session everytime we train a model
  tensorboard = create_tensorboard_callback()

  # Fit the model to the data passing it the callbacks we created
  model.fit(x=train_data,
            epochs=NUM_EPOCHS,
            validation_data=val_data,
            validation_freq=1, # how often we test patterns per epoch
            callbacks=[tensorboard, early_stopping])
  # Return the fitted model
  return model

# Fit the model to the data
model = train_model()
```

```
Building model with: https://tfhub.dev/google/imagenet/mobilenet_v2_130_224/classification/4
Train for 25 steps, validate for 7 steps
Epoch 1/100
25/25 [==============================] - 604s 24s/step - loss: 4.5757 - accuracy: 0.0988 - val_loss: 3.4853 - val_accuracy: 0.2200
Epoch 2/100
25/25 [==============================] - 4s 179ms/step - loss: 1.6390 - accuracy: 0.6900 - val_loss: 2.1415 - val_accuracy: 0.5100
Epoch 3/100
25/25 [==============================] - 4s 175ms/step - loss: 0.5734 - accuracy: 0.9388 - val_loss: 1.6463 - val_accuracy: 0.6050
Epoch 4/100
25/25 [==============================] - 4s 176ms/step - loss: 0.2523 - accuracy: 0.9912 - val_loss: 1.4408 - val_accuracy: 0.6250
Epoch 5/100
25/25 [==============================] - 4s 174ms/step - loss: 0.1475 - accuracy: 0.9987 - val_loss: 1.3662 - val_accuracy: 0.6450
Epoch 6/100
25/25 [==============================] - 4s 174ms/step - loss: 0.1007 - accuracy: 1.0000 - val_loss: 1.3218 - val_accuracy: 0.6350
Epoch 7/100
25/25 [==============================] - 4s 177ms/step - loss: 0.0757 - accuracy: 1.0000 - val_loss: 1.2891 - val_accuracy: 0.6350
Epoch 8/100
25/25 [==============================] - 4s 180ms/step - loss: 0.0602 - accuracy: 1.0000 - val_loss: 1.2607 - val_accuracy: 0.6300
```

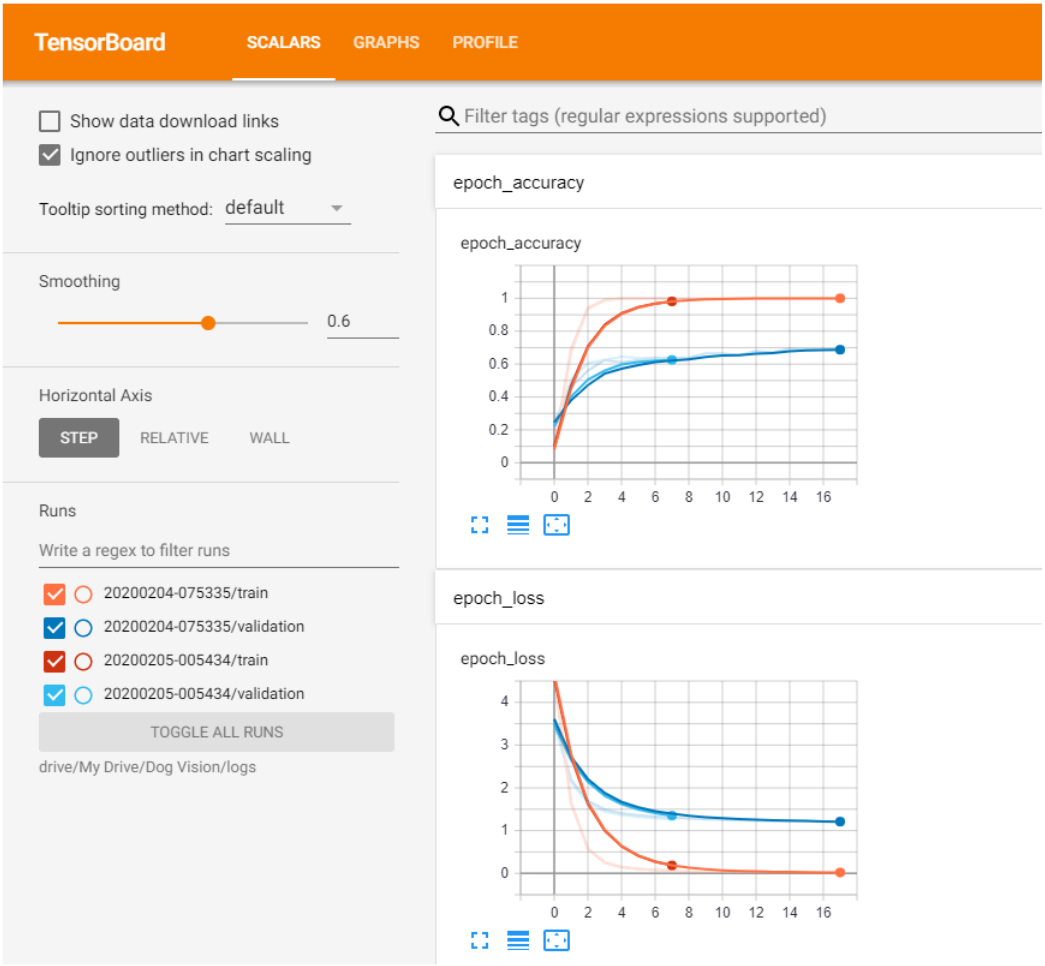
Note that it looks like our model is overfitting because it's performing far better on the training dataset than the validation dataset. Overfitting to begin with is a good thing! It means our model is learning.

6.4.2 Evaluating Performance with TensorBoard

The TensorBoard magic function (%tensorboard) will access the logs directory we created earlier and visualize its contents.

```
%tensorboard --logdir drive/My\ Drive/Dog\ Vision/logs # \ used for space in name
```

We can tell the model is overfitting because the red line (for the training data) is learning the patterns far too well. In the future, we would want to stop this by using more data. Note that the red line is behind the orange, ending at epoch 7.



6.4.3 Making and Evaluating Predictions

```
# Make predictions on the validation data (not used to train on)
predictions = model.predict(val_data, verbose=1) # verbose shows us progress bar
len(y_val) # 200
predictions.shape # (200, 120) meaning 200 arrays with 120 values in each

index = 42
print(f"Max value (probability of prediction): {np.max(predictions[index])}")
print(f"Sum: {np.sum(predictions[index])}") # sum to 1 since softmax
print(f"Max index: {np.argmax(predictions[index])}")
print(f"Predicted label: {unique_breeds[np.argmax(predictions[index])]}")
```

```
Max value (probability of prediction): 0.7575560808181763
Sum: 0.9999999403953552
Max index: 113
Predicted label: walker_hound
```

Note that above, each sub-array (with 120 values) in the predictions array will sum to approximately 1 for all of the values combined. The position of the largest value within the sub-array (value with the highest probability) will correspond to our label in the unique_breeds data set. The higher this value, the more confident that the model is about its guess.

6.5 Step 4: Evaluating a Model

6.5.1 Transform Predictions to Text

Having the the above functionality is great but we want to be able to do it at scale. It would be even better if we could see the image the prediction is being made on. Also, **prediction probabilities** are also known as *confidence levels*.

Now since our validation data is still in a batch dataset, we'll have to unbatch it to make predictions on the validation images and then compare those predictions to the validation labels (truth labels).

```
# Turn prediction probabilities into their respective label (easier to understand)
def get_pred_label(prediction_probabilities):
  """
  Turns an array of prediction probabilities into a label.
  """
  return unique_breeds[np.argmax(prediction_probabilities)]

# Create a function to unbatch a batch dataset
def unbatchify(data):
  """
  Takes a batched dataset of (image, label) Tensors and reutrns separate arrays
  of images and labels.
  """
  images = []
  labels = []
  # Loop through unbatched data
  for image, label in data.unbatch().as_numpy_iterator():
    images.append(image)
    labels.append(unique_breeds[np.argmax(label)])
  return images, labels

# Unbatchify the validation data
val_images, val_labels = unbatchify(val_data)

# Get label for unbatched validation set
get_pred_label(val_labels[0]) # 'cairn'
```

6.5.2 Visualize Model Predictions

Now we've got ways to get Prediction labels, Validation labels (truth labels), and Validation images. Next, let's make a function to make these all a bit more visual. We'll create a function which:

- Takes an array of prediction probabilities, truth labels, and images and an integer.
- Convert the prediction probabilities to a predicted label.
- Plot the predicted label, its predicted probability, the truth label and target image on a single plot.