

# Machine Learning in Python

## Contents

<b>1</b>	<b>Six Step Machine Learning Framework</b>	<b>1</b>
1.1	Step 1: Problem Definition	1
1.2	Step 2: Data	1
1.3	Step 3: Evaluation	2
1.4	Step 4: Features	2
1.5	Step 5: Modeling	2
<b>2</b>	<b>Pandas: Data Analysis</b>	<b>3</b>
2.1	Series, Data Frames, and CSVs	3
2.2	Describing Data	3
2.3	Selecting/Viewing Data	4
2.4	Manipulating Data	4
<b>3</b>	<b>NumPy</b>	<b>5</b>
3.1	Arrays and Attributes	5
3.2	Array Functions	6
3.2.1	Sorting Arrays	7
3.2.2	Turn Images into NumPy Arrays	7
<b>4</b>	<b>Matplotlib: Plotting and Visualizing</b>	<b>8</b>
4.1	Types of Plots	8
4.1.1	Subplots	9
4.2	Plotting from Pandas DataFrames	9
4.3	Customizing Plots	10
<b>5</b>	<b>Scikit-learn: Machine Learning Models</b>	<b>11</b>
5.1	Workflow	11
5.2	Step 1: Getting Data Ready	12
5.3	Step 2: Choosing the Right Model	14
5.3.1	Regression models	15
5.3.2	Classification models	15
5.4	Step 3: Making Predictions with our Model	16
5.5	Step 4: Evaluating a Model	16
5.5.1	Evaluating a Classification Model	17
5.5.2	Evaluating a Regression Model	19
5.5.3	Cross Validation and Scoring parameter	19
5.5.4	Metric Functions with scikit-learn	20
5.6	Step 5: Improving a Model	21
5.6.1	Tuning Hyperparameters Manually	21
5.6.2	Tuning Hyperparameters with RandomizedSearchCV	22
5.6.3	Tuning Hyperparameters with GridSearchCV	22
5.7	Step 6: Saving and Loading a Model	23
5.8	Putting It All Together	24
5.9	Milestone Project 1 (Classification)	25
5.9.1	Data Exploration (EDA)	26
5.9.2	Modeling (Step 5)	28
5.9.3	Tuning Hyperparameters	28
5.9.4	Evaluating the Model	30
5.9.5	Find Most Important Features	32

5.10	Milestone Project 2 (Regression)	33
5.10.1	Exploring the Data (EDA)	33
5.10.2	Converting Strings to Categories	34
5.10.3	Filling Missing Values	35
5.10.4	Modeling (Step 5)	36
<b>6</b>	<b>Neural Networks: Deep/Transfer Learning &amp; TensorFlow 2</b>	<b>40</b>
6.1	Workflow & Setup	40
6.2	Step 1: Preparing Data	41
6.2.1	Getting Images and Their Labels	41
6.2.2	Turning Images into Tensors	42
6.2.3	Turning Data into Batches	42
6.3	Step 2: Building a Deep Learning Model	44
6.3.1	Creating callbacks	45
6.4	Step 3: Fit Model and Make Predictions	46
6.4.1	Training a Deep Neural Network (on a data subset)	46
6.4.2	Evaluating Performance with TensorBoard	47
6.4.3	Making and Evaluating Predictions	47
6.5	Step 4: Evaluating a Model	48
6.5.1	Transform Predictions to Text	48
6.5.2	Visualize Model Predictions	48
6.6	Step 5: Improving our Model (on full dataset)	50
6.7	Step 6: Saving and Loading a Trained Model	52
6.8	Making Predictions on Test Data	52
6.8.1	Preparing Test Predictions for Kaggle	53
6.8.2	Predicting on Custom Images	54

# 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 to. 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) that 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 exist 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.

```
1 import pandas as pd
2
3 cars = pd.Series(['BMW', 'Toyota', 'Honda'])
4 colors = pd.Series(['Red', 'Blue', 'White'])
5
6 car_data = pd.DataFrame({'Car Make': cars, 'Color': colors})
7
8 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).

```
1 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 use slicing when accessing rows. However, `.loc` will give us up to and including the last specified row while `.iloc` will give us all up to but *not* including the last specified row.

```
1 car_sales.loc[:3] # gives us rows 0, 1, 2, 3
2 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.

```
1 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 variable column.

```
1 pd.crosstab(car_sales['Make'], car_sales['Doors'])
2
3 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 let us 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.

```
1 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.

```
1 car_sales_missing = pd.read_csv('car_sales_missing_data.csv')
2
3 car_sales_missing['Odometer'].fillna(car_sales_missing['Odometer'].mean(),
4                                     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.

```
1 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.

```
1 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).

```
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.

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

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

```
1 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.

```
1 import numpy as np
2
3 a1 = np.array([1, 2, 3])
4 a1.shape # returns (3,)
5
6 a2 = np.array([[1, 2, 3],
7               [4, 5, 6]])
8 a2.ndim # returns 2
9
10 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.

```
1 ones = np.ones((2,3)) # create a 2x3 array of 1's
2
3 zeros = np.zeros((2,3)) # create a 2x3 array of 0's
4
5 range_arr = np.arange(0, 10, 2) # create a 1D array from 0 to 10 (inc by 2)
6
7 random_arr = np.random.randint(0, 10, size=(3,5))
8 # create 3x5 array of random ints between 0 and 10
9
10 random_arr_2 = np.random.random((5,3)) # create a 5x3 array of values between 0 and 1
11
12 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.

```
1 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.

```
1 large_arr = np.random.random(1000000)
2
3 %timeit sum(large_arr) # approx 19.9 ms per loop (526 times slower than np method)
4 %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).

```
1 high_var = np.array([1, 100, 200, 300, 4000, 5000])
2 low_var = np.array([2, 4, 6, 8, 10])
3
4 np.var(high_var), np.var(low_var) # (4296133.47, 8.0)
5 np.std(high_var), np.std(low_var) # (2072.712, 2.828)
6
7 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.

```
1 a2.shape # (2, 3)
2 a2_reshape = a2.reshape(2, 3, 1) # change into a 3D matrix
3
4 a2_transpose = a2.T
5 a2_transpose.shape # (3, 2)
```

We can perform different types of multiplication on NumPy arrays.

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

$$\text{Dot product: } \begin{bmatrix} A & B & C \\ D & E & F \\ G & H & I \end{bmatrix}_{3 \times 3} * \begin{bmatrix} J & K \\ L & M \\ N & O \end{bmatrix}_{3 \times 2} = \begin{bmatrix} A * J + B * L + C * N & A * K + B * M + C * O \\ D * J + E * L + F * N & D * K + E * M + F * O \\ G * J + H * L + I * N & G * K + H * M + I * O \end{bmatrix}_{3 \times 2}$$

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

```
1 np.random.seed(0)
2
3 mat1 = np.random.randint(10, size=(5,3)) # range 0-10 of 5x3 array
4 mat2 = np.random.randint(10, size=(5,3)) # range 0-10 of 5x3 array
5
6 np.multiply(mat1, mat2) # element wise multiplication
7
8 mat1.shape, mat2.T.shape # ((5, 3), (3, 5))
9
10 mat3 = np.dot(mat1, mat2.T)
11 mat3.shape # (5, 5)
```



## Dot product example

```
1  np.random.seed(0)
2
3  sales_amount = np.random.randint(20, size=(5,3)) # 5x3 matrix of sales
4
5  weekly_sales = pd.DataFrame(sales_amounts,
6                              index = ['Mon', 'Tues', 'Wed', 'Thrs', 'Fri', 'Sat', 'Sun'],
7                              columns = ['Almond', 'Peanut', 'Cashew'])
8
9  prices = np.array([10, 8, 12])
10 prices.shape # (3,) but we need (1,3)
11
12 butter_prices = pd.DataFrame(prices.reshape(1,3),
13                              index = ['Price'],
14                              columns = ['Almond', 'Peanut', 'Cashew'])
15
16 butter_prices.shape, weekly_sales.shape # ((1, 3), (5, 3))
17 daily_sales = butter_prices.dot(weekly_sales.T)
18 # creates a 1x5 data frame with Price as row and days as columns
19
20 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 in 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.

```
1  rand_arr = np.random.randint(10, size=(3,5))
2
3  np.argmin(a1) # index for minimum value (0)
4  np.argmax(a1) # index for maximum value (2)
5
6  np.argmax(rand_array, axis=0) # return the index pos for the max value in each column
7  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.

```
1  import matplotlib.image import imread
2
3  panda = imread('images/panda.png')
4  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.

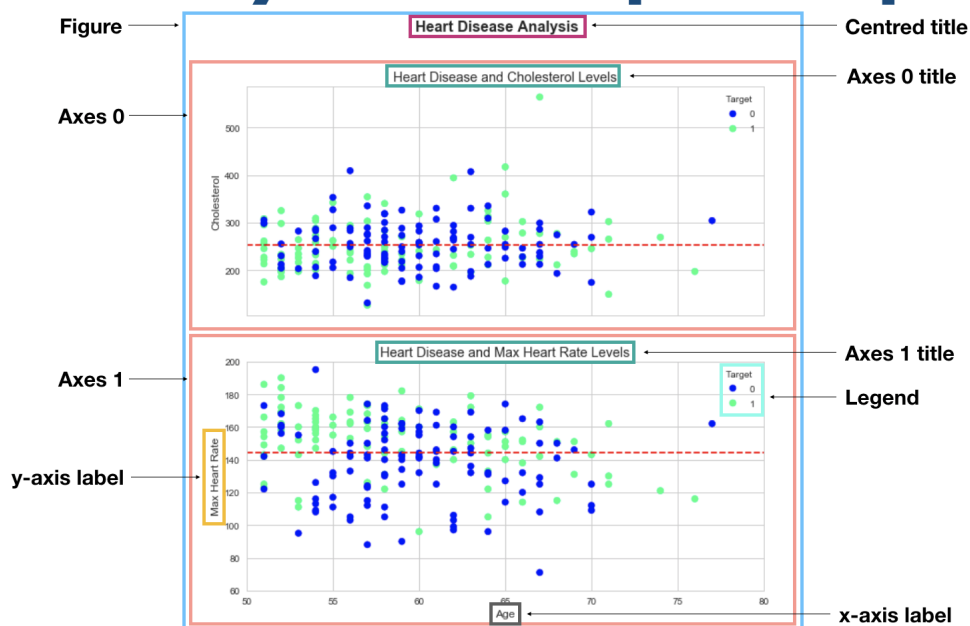
## 4 Matplotlib: Plotting and Visualizing

### 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:

```
1 import matplotlib.pyplot as plt
2
3 # 1. Prepare data
4 x = [1, 2, 3, 4]
5 y = [11, 22, 33, 44]
6
7 # 2. Setup plot (create our objects)
8 fig, ax = plt.subplots(figsize=(10, 10)) # (width, height)
9
10 # 3. Plot data
11 ax.plot(x, y)
12
13 # 4. Customize plot
14 ax.set(title='Sample Plot',
15        xlabel = 'x-axis',
16        ylabel = 'y-axis')
17
18 # 5. Save and show your figure
19 fig.savefig('sample-plot.png')
```

## Anatomy of a Matplotlib plot



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

```
1 x = np.linspace(0, 10, 100) # 100 evenly spaced numbers between 0 and 10
2
3 fig, ax = plt.subplots()
4
5 ax.plot(x, x**2) # create a line graph
6 ax.scatter(x, np.exp(x)) # create a scatter plot
7 ax.bar(['Almonds', 'Peanuts'], [10, 12]) # create a bar chart
8 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.

```
1 x = np.random.randn(1000)
2
3 # Create a 10x5 figure with 4 subplots (2 in each row/column)
4 fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(nrows = 2,
5                                             ncols = 2,
6                                             figsize = (10,5))
7 # access subplots: ax1.plot(), ax2.hist(), etc.
8
9 # Create a 10x5 figure with 4 subplots (2 in each row/column)
10 fig, ax = plt.subplots(nrows = 2,
11                       ncols = 2,
12                       figsize = (10,5))
13 # 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.

```
1 car_sales = pd.read_csv('car-sales.csv')
2
3 car_sales['Price'] = car_sales['Price'].str.replace('[\$,\\.]', '') # remove symbols
4 car_sales['Price'] = car_sales['Price'].str[:-2] # remove extra zeros
5 car_sales['Price'] = car_sales['Price'].astype(int) # cast as an integer
6
7 # create a sale date column for each row in our data frame
8 car_sales['Sale Date'] = pd.date_range('1/1/2020', periods = len(car_sales))
9
10 # create total sales column by summing all prices
11 car_sales['Total Sales'] = car_sales['Price'].cumsum()
12
13 car_sales.plot(x='Odometer (KM)', y='Price', kind='scatter')
14 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.

```
1 heart_disease = pd.read_csv('heart-disease.csv')
2
3 over_50 = heart_disease[heart_disease['age'] > 50]
4
5 # create a scatter plot of age vs. chol, colored by the target values (OO Method)
6 fig, ax = plt.subplots(figsize=(10, 6))
7 scatter = ax.scatter(x=over_50['age'], y=over_50['chol'], c=over_50['target'])
8
9 ax.set(title='Heart Disease and Cholesterol Levels', xlabel='Age',
10       ylabel='Cholesterol')
11 ax.legend(*scatter.legend_elements(), title='Target') # legend elements from 'c='
12
13 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).

```

1  fig, (ax0, ax1) = plt.subplots(nrows=2, ncols=1, figsize=(10,10))
2
3  # Plot and customize axis 0 graph
4  scatter = ax0.scatter(x=over_50['age'], y=over_50['chol'], c=over_50['target'])
5  ax0.set(title='Heart Disease and Cholesterol Levels', xlabel='Age',
6         ylabel='Cholesterol')
7  ax0.legend(*scatter.legend_elements(), title='Target')
8  ax0.axhline(over_50['chol'].mean(), linestyle='--') # mean line
9
10 # Plot and customize axis 1 graph
11 scatter = ax1.scatter(x=over_50['age'], y=over_50['thalach'], c=over_50['target'])
12 ax1.set(title='Heart Disease and Max Heart Rate', xlabel='Age',
13        ylabel='Max Heart Rate')
14 ax1.legend(*scatter.legend_elements(), title='Target')
15 ax1.axhline(over_50['thalach'].mean(), linestyle='--') # mean line
16
17 fig.suptitle('Heart Disease Analysis', 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`.

```

1  plt.style.use('seaborn-whitegrid') # change the style of our plot
2
3  fig, (ax0, ax1) = plt.subplots(nrows=2, ncols=1, figsize=(10,10))
4
5  # Plot and customize axis 0 graph
6  scatter = ax0.scatter(x=over_50['age'], y=over_50['chol'], c=over_50['target'],
7                        cmap='winter') # this changes the color scheme
8  ax0.set(title='Heart Disease and Cholesterol Levels', xlabel='Age',
9         ylabel='Cholesterol')
10 ax0.set_xlim([50, 80]) # change the x-axis range limits
11 ax0.set_ylim([60, 200]) # change the y-axis range limits
12 ax0.legend(*scatter.legend_elements(), title='Target')
13 ax0.axhline(over_50['chol'].mean(), linestyle='--') # mean line
14
15 # Plot and customize axis 1 graph
16 scatter = ax1.scatter(x=over_50['age'], y=over_50['thalach'], c=over_50['target'],
17                       cmap='winter') # this changes the color scheme
18 ax1.set(title='Heart Disease and Max Heart Rate', xlabel='Age',
19        ylabel='Max Heart Rate')
20 ax1.set_xlim([50, 80]) # change the x-axis range limits
21 ax1.set_ylim([60, 200]) # change the y-axis range limits
22 ax1.legend(*scatter.legend_elements(), title='Target')
23 ax1.axhline(over_50['thalach'].mean(), linestyle='--') # mean line
24
25 fig.suptitle('Heart Disease Analysis', fontsize=16, fontweight='bold')
26
27 fig.savefig('Heart-disease-analysis.png') # saves to current directory

```

## 5 Scikit-learn: Machine Learning 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.

```
1 import pandas as pd
2 import numpy as np
3
4 # Step 1: getting data ready
5 heart_disease = pd.read_csv('data/heart-disease.csv')
6 X = heart_disease.drop('target', axis=1) # create features matrix
7 y = heart_disease['target'] # create labels matrix
8
9 # Step 2: Chose the model and hyperparameters
10 from sklearn.ensemble import RandomForestClassifier
11 clf = RandomForestClassifier(n_estimators = 100)
12 clf.get_params() # show the parameter values
13
14 #Step 3: Fit the model to the training data
15 from sklearn.model_selection import train_test_split
16 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2)
17 clf.fit(X_train, y_train) # find patterns in training data
18 y_preds = clf.predict(X_test)
19
20 # Step 4: Evaluate the model on training and test data
21 clf.score(X_train, y_train) # 1.0
22 clf.score(X_test, y_test) # 0.754
23
24 from sklearn.metrics import classification_report, confusion_matrix, accuracy_score
25 classification_report(y_test, y_preds) # compare test labels to pred labels
26 confusion_matrix(y_test, y_preds)
27 accuracy_score(y_test, y_preds) # 0.754
28
29 # Step 5: Improve the model
30 np.random.seed(42)
31 for i in range(10, 100, 10):
32     print(f'Trying model with {i} estimators...')
33     clf = RandomForestClassifier(n_estimators = i).fit(X_train, y_train)
34     print(f'Model accuracy on test set: {clf.score(X_test, y_test) * 100:.2f}%')
35     print("") # after running loop, we see 20 estimators gets us to 83% accuracy
36
37 # Step 6: Save model and load it
38 import pickle
39
40 pickle.dump(clf, open('random_forst_model_1.pkl', 'wb')) # saves to a file in dir
41
42 loaded_model = pickle.load(open('random_forst_model_1.pkl', 'rb'))
43 loaded_model.score(X_test, y_test)
```

## 5.2 Step 1: Getting Data Ready

There are three main things we must do 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**:

```
1 heart_disease = pd.read_csv('heart-disease.csv')
2
3 # Split into features and labels
4 X = heart_disease.drop('target', axis=1)
5 y = heart_disease['target']
6
7 # Split into training and test sets
8 from sklearn.model_selection import train_test_split
9 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 with the `pd.get_dummies()` method, which does the same thing as `OneHotEncoder`.

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

```
1 np.random.seed(42)
2 car_sales = pd.read_csv('car-sales-extended.csv')
3
4 X = car_sales.drop('Price', axis=1)
5 y = car_sales['Price']
6
7 # Convert categorical to numerical data
8 from sklearn.preprocessing import OneHotEncoder
9 from sklearn.compose import ColumnTransformer
10
11 # Option 1: One Hot Encoder method
12 categorical_features = ['Make', 'Colour', 'Doors']
13 one_hot = OneHotEncoder()
14 transformer = ColumnTransformer([('one_hot', one_hot, categorical_features)],
15                                 remainder='passthrough')
16 transformed_X = transformer.fit_transform(X)
17
18 # Option 2: Pandas get_dummies() method
19 dummies = pd.get_dummies(car_sales[['Make', 'Colour', 'Doors']])
20
21 # Fit the model with the new data
22 X_train, X_test, y_train, y_test = train_test_split(transformed_X, y,
23                                                       test_size = 0.2)
24
25 from sklearn.ensemble import RandomForestRegressor # predicts numbers
26 model = RandomForestRegressor()
27 model.fit(X_train, y_train)
28 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.

```
1 car_sales_missing = pd.read_csv('data/car-sales-extended-missing-data.csv')
2
3 car_sales_missing.isna().sum() # around 50 values missing in each column
4
5 ##### Fill in missing values using Pandas #####
6 car_sales_missing['Make'].fillna('missing', inplace=True)
7 car_sales_missing['Colour'].fillna('missing', inplace=True)
8 car_sales_missing['Odometer (KM)'].fillna(car_sales_missing['Odometer (KM)'].mean(),
9                                           inplace=True)
10 car_sales_missing['Doors'].fillna(4, inplace=True)
11 car_sales_missing.dropna(inplace=True) # drop missing values in Price column
12
13 # Split data and convert to numerical
14 X = car_sales_missing.drop('Price', axis=1)
15 y = car_sales_missing['Price']
16
17 categorical_features = ['Make', 'Colour', 'Doors']
18 one_hot = OneHotEncoder()
19 transformer = ColumnTransformer([('one_hot', one_hot, categorical_features)],
20                                remainder='passthrough')
21 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).

```
1 car_sales_missing = pd.read_csv('data/car-sales-extended-missing-data.csv')
2
3 car_sales_missing.dropna(subset=['Price'], inplace=True) # drop rows with no labels
4 X = car_sales_missing.drop('Price', axis=1)
5 y = car_sales_missing['Price']
6
7 # Split into train/test data before filling missing values
8 np.random.seed(42)
9 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
10
11 ##### Fill missing values using scikit-learn #####
12 from sklearn.impute import SimpleImputer
13 from sklearn.compose import ColumnTransformer
14
15 # Fill categorical values with 'missing' & numerical values with mean
16 cat_imputer = SimpleImputer(strategy="constant", fill_value="missing") # categorical
17 door_imputer = SimpleImputer(strategy="constant", fill_value=4) # door column
18 num_imputer = SimpleImputer(strategy="mean") # numerical
19
20 cat_features = ["Make", "Colour"]
21 door_feature = ["Doors"]
22 num_features = ["Odometer (KM)"]
23
24 # Create an imputer (something that fills missing data)
25 imputer = ColumnTransformer([
26     ("cat_imputer", cat_imputer, cat_features), # (name, imputer, features)
27     ("door_imputer", door_imputer, door_feature),
28     ("num_imputer", num_imputer, num_features) ])
```



```

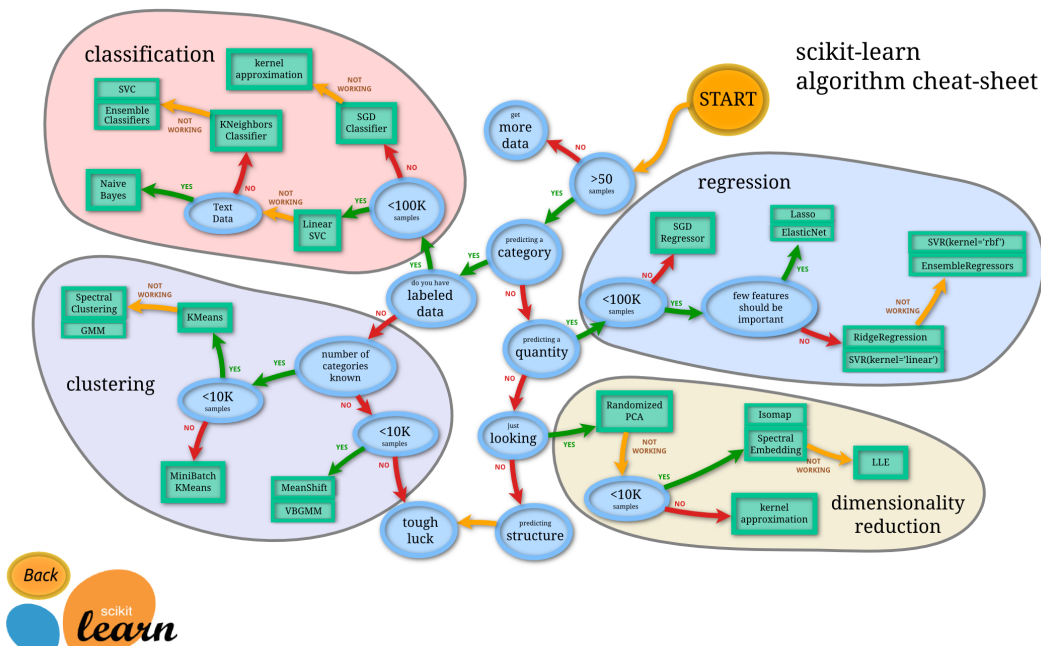
29
30 # Fill train and test values separately
31 filled_X_train = imputer.fit_transform(X_train)
32 filled_X_test = imputer.transform(X_test)
33
34 # Get our transformed data array's back into DataFrame's
35 car_sales_filled_train = pd.DataFrame(filled_X_train,
36                                       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*.

```
1 from sklearn.datasets import load_boston
2 from sklearn.linear_model import Ridge
3 np.random.seed(42)
4
5 boston = load_boston() # loads in as a dictionary
6 boston_df = pd.DataFrame(boston['data'], columns=boston['feature_names'])
7 boston_df['target'] = pd.Series(boston['target'])
8 len(boston_df) # 506 samples
9
10 X = boston_df.drop('target', axis=1)
11 y = boston_df['target']
12
13 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
14
15 model = Ridge()
16 model.fit(X_train, y_train)
17 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.

```
1 from sklearn.ensemble import RandomForestRegressor
2 np.random.seed(42)
3
4 rf = RandomForestRegressor(n_estimators = 100)
5 rf.fit(X_train, y_train)
6 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).

```
1 from sklearn.svm import LinearSVC
2 np.random.seed(42)
3
4 heart_disease = pd.read_csv('data/heart-disease.csv')
5
6 X = heart_disease.drop('target', axis=1)
7 y = heart_disease['target']
8
9 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
10
11 clf = LinearSVC(max_iter=10000)
12 clf.fit(X_train, y_train)
13 clf.score(X_test, y_test) # 0.4754
```

This is a binary classification problem, and our model is 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.

```

1 from sklearn.ensemble import RandomForestClassifier
2 np.random.seed(42)
3
4 clf = RandomForestClassifier(n_estimators = 100)
5 clf.fit(X_train, y_train)
6 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.

```

1 # Compare predictions to truth labels to evaluate the model
2 y_preds = clf.predict(X_test)
3 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.

```

1 clf.predict_proba(X_test) # first value in array is [0.89, 0.11]
2 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).

```

1 X = boston_df.drop('target', axis=1)
2 y = boston_df['target']
3 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
4
5 model = RandomForestRegressor(n_estimators = 100).fit(X_train, y_train)
6 y_preds = model.predict(X_test)
7
8 from sklearn.metrics import mean_absolute_error
9 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.

```

1  from sklearn.ensemble import RandomForestClassifier
2  np.random.seed(42)
3
4  X = heart_disease.drop('target', axis=1)
5  y = heart_disease['target']
6  X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
7
8  clf = RandomForestClassifier(n_estimators = 100).fit(X_train, y_train)
9  clf.score(X_test, y_test) # 0.852459
10

```

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).

```

1  from sklearn.model_selection import cross_val_score
2  np.random.seed(42)
3
4  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:

```

1  from sklearn.model_selection import cross_val_score
2  from sklearn.ensemble import RandomForestClassifier
3  np.random.seed(42)
4
5  X = heart_disease.drop('target', axis=1)
6  y = heart_disease['target']
7
8  clf = RandomForestClassifier(n_estimators = 100)
9  c_v_score = cross_val_score(clf, X, y, cv=5)
10 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.

```

1  from sklearn.metrics import roc_curve
2
3  X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
4
5  clf = RandomForestClassifier(n_estimators = 100).fit(X_train, y_train)
6  y_probs = clf.predict_proba(X_test)
7
8  y_probs_positive = y_probs[:, 1] # only keep columns with 1's probability

```

```

1 fpr, tpr, threshold = roc_curve(y_test, y_probs_positive)
2
3 def plot_roc_curve(fpr, tpr):
4     plt.plot(fpr, tpr, color='orange', label='ROC') # roc curve
5     plt.plot([0,1], [0,1], color='darkblue', linestyle='--', label='Guessing') # base
6     plt.xlabel('False Positive Rate (FPR)')
7     plt.ylabel('True Positive Rate (TPR)')
8     plt.title('Receiver Operating Characteristic (ROC) Curve')
9     plt.legend()
10    plt.show()
11
12    plot_roc_curve(fpr, tpr)
13
14    from sklearn.metrics import roc_auc_score
15    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.

```

1 from sklearn.metrics import confusion_matrix
2
3 y_preds = clf.predict(X_test)
4 conf_mat = confusion_matrix(y_test, y_preds)
5 # create a table which shows true pos/neg and false pos/neg results
6 pd.crosstab(y_test, y_preds, rownames=['Actual Label'], colnames=['Predicted Label'])
7
8 import seaborn as sns
9
10 def plot_conf_mat(conf_mat):
11     fig, ax = plt.subplots(figsize=(3,3))
12     ax = sns.heatmap(conf_mat, annot=True, cbar=False)
13     plt.xlabel('True Label')
14     plt.ylabel('Predicted Label')
15     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.

```

1 from sklearn.metrics import classification_report
2
3 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.

```
1 import sklearn.ensemble import RandomForestRegressor
2 np.random.seed(42)
3
4 X = boston_df.drop('target', axis=1)
5 y = boston_df['target']
6 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
7
8 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.

```
1 from sklearn.metrics import r2_score
2
3 mode.score(X_test, y_test) # 0.87387
4
5 # create an array of y_test mean values to compare of predictions
6 y_test_mean = np.full(len(y_test), y_test.mean())
7
8 r2_score(y_test, y_test_mean) # 0.0
9 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).

```
1 from sklearn.metrics import mean_absolute_error
2
3 y_preds = model.predict(X_test)
4 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.

```
1 from sklearn.metrics import mean_squared_error
2
3 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 far 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.

```

1  from sklearn.model_selection import cross_val_score
2  from sklearn.ensemble import RandomForestClassifier
3  from sklearn.ensemble import RandomForestRegressor
4  np.random.seed(42)
5
6  # Classification model scoring
7  X = heart_disease.drop('target', axis=1)
8  y = heart_disease['target']
9  clf = RandomForestClassifier(n_estimators = 100)
10
11 cv_acc = cross_val_score(clf, X, y, cv=5)
12 cv_precision = cross_val_score(clf, X, y, cv=5, scoring='precision')
13 cv_recall = cross_val_score(clf, X, y, cv=5, scoring='recall')
14 cv_f1 = cross_val_score(clf, X, y, cv=5, scoring='f1')
15
16 # Regression model scoring
17 X = boston_df.drop('target', axis=1)
18 y = boston_df['target']
19 model = RandomForestRegressor(n_estimators = 100)
20
21 cv_r2 = cross_val_score(model, X, y, cv=5)
22 cv_mae = cross_val_score(model, X, y, cv=5, scoring='neg_mean_absolute_error')
23 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:

```

1  from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
2  from sklearn.ensemble import RandomForestClassifier
3  from sklearn.model_selection import train_test_split
4  np.random.seed(42)
5
6  X = heart_disease.drop('target', axis=1)
7  y = heart_disease['target']
8  X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
9  clf = RandomForestClassifier(n_estimators = 100).fit(X_train, y_train)
10
11 y_preds = clf.predict(X_test)
12 accuracy_score(y_test, y_preds) # 0.8525
13 precision_score(y_test, y_preds) # 0.8484
14 recall_score(y_test, y_preds) # 0.875
15 f1_score(y_test, y_preds) # 0.8615

```

Next, lets look at regression evaluation functions:

```

1  from sklearn.metrics import r2_score, mean_absolute_error, mean_squared_error
2  from sklearn.ensemble import RandomForestRegressor
3  from sklearn.model_selection import train_test_split
4  np.random.seed(42)
5
6  X = boston_df.drop('target', axis=1)
7  y = boston_df['target']
8  X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
9  model = RandomForestRegressor(n_estimators = 100).fit(X_train, y_train)
10
11 y_preds = model.predict(X_test)
12 r2_score(y_test, y_preds) # 0.8739
13 mean_absolute_error(y_test, y_preds) # 2.122
14 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.

```
1 from sklearn.ensemble import RandomForestClassifier
2
3 clf = RandomForestClassifier()
4 clf.get_params()
```

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

### 5.6.1 Tuning Hyperparameters Manually

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

```
1 def evaluate_preds(y_true, y_preds):
2     accuracy = accuracy_score(y_true, y_preds)
3     precision = precision_score(y_true, y_preds)
4     recall = recall_score(y_true, y_preds)
5     f1 = f1_score(y_true, y_preds)
6     metric_dict = {'accuracy': round(accuracy, 2),
7                   'precision': round(precision, 2),
8                   'recall': round(recall, 2),
9                   'f1': round(f1, 2) }
10    print('Acc: {}'.format(accuracy * 100:0.2f))
11    print('Precision: {}'.format(precision:0.2f))
12    print('Recall: {}'.format(recall:.2f))
13    print('F1 Score: {}'.format(f1:0.2f))
14    return metric_dict
```

Next let's split the data into training, validation, and test split (by hand):

```
1 from sklearn.ensemble import RandomForestClassifier
2 np.random.seed(42)
3
4 heart_disease_shuffled = heart_disease.sample(frac=1)
5 X = heart_disease_shuffled.drop('target', axis=1)
6 y = heart_disease_shuffled['target']
7
8 train_split = round(0.7 * len(heart_disease_shuffled)) # 70% of data
9 valid_split = round(train_split + 0.15 * len(heart_disease_shuffled)) # 15% of data
10 X_train, y_train = X[:train_split], y[:train_split]
11 X_valid, y_valid = X[train_split:valid_split], y[train_split:valid_split]
12 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:

```
1  clf = RandomForestClassifier()
2  clf.fit(X_train, y_train)
3  y_preds = clf.predict(X_valid) # baseline predictions
4  baseline_metrics = evaluate_preds(y_valid, y_preds) # 80%, 0.77, 0.92, 0.84
5
6  # Create a second classifier with different hyperparameters & make predictions
7  clf_2 = RandomForestClassifier(n_estimators=100)
8  clf_2.fit(X_train, y_train)
9  y_preds_2 = clf_2.predict(X_valid)
10 clf_2_metrics = evaluate_preds(y_valid, y_preds_2) # 82.2%, 0.84, 0.84, 0.84
11 # 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).

```
1  from sklearn.model_selection import RandomizedSearchCV
2  np.random.seed(42)
3
4  grid = {'n_estimators': [10, 100, 200, 500, 1000, 1200],
5         'max_depth': [None, 5, 10, 20, 30],
6         'max_features': ['auto', 'sqrt'],
7         'min_samples_split': [2, 4, 6],
8         'min_samples_leaf': [1, 2, 4]}
9
10 X = heart_disease_shuffled.drop('target', axis=1)
11 y = heart_disease_shuffled['target']
12 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
13 clf = RandomForestClassifier(n_jobs=1) # n_jobs is dedicated CPU power
14
15 # Setup RandomizedSearchCV
16 rs_clf = RandomizedSearchCV(estimator=clf,
17                             param_distributions=grid,
18                             n_iter=10, # number of models to try
19                             cv=5, # creates 5 different validation sets
20                             verbose=2)
21
22 # Fit the RandomizedSearchCV version of clf
23 rs_clf.fit(X_train, y_train)
24 rs_clf.best_params_ # shows which combination got best results
25
26 # Make predictions with best hyperparameters & evaluate
27 rs_y_preds = rs_clf.predict(X_test)
28 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).



```

1  from sklearn.model_selection import GridSearchCV, train_test_split
2  np.random.seed(42)
3
4  grid_2 = {'n_estimators': [100, 200, 500],
5           'max_depth': [None],
6           'max_features': ['auto', 'sqrt'],
7           'min_samples_split': [6],
8           'min_samples_leaf': [1, 2]}
9
10 # Use the data splits and original clf model from previous code block
11 # Setup GridSearchCV
12 gs_clf = RandomizedSearchCV(estimator=clf,
13                             param_grid=grid_2,
14                             cv=5,
15                             verbose=2)
16
17 # Fit the GridSearchCV version of clf
18 gs_clf.fit(X_train, y_train)
19 gs_clf.best_params_
20
21 gs_y_preds = gs_clf.predict(X_test)
22 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.

```

1  compare_metrics = pd.DataFrame({'baseline': baseline_metrics,
2                                'clf_2': clf_2_metrics,
3                                'random search': rs_metrics,
4                                'grid search': gs_metrics})
5
6  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.

```

1  import pickle
2
3  # Save model in our current working directory using Pickle
4  pickle.dump(gs_clf, open('gs_random_random_forest_model_1.pkl', 'wb'))
5
6  # Load saved model
7  loaded_pickle_model = pickle.load(open('gs_random_random_forest_model_1.pkl', 'rb'))
8
9  # Make predictions using model
10 pickle_y_preds = loaded_pickle_model.predict(X_test)
11 evaluate_preds(y_test, pickle_y_preds)

```

```

1  from joblib import dump, load
2
3  dump(gs_clf, filename='gs_random_forest_model_1.joblib') # save a joblib model
4
5  loaded_job_model = load(filename='gs_random_forest_model_1.joblib') # load in model
6
7  # Make and evaluate joblib predictions
8  joblib_y_preds = loaded_job_model.predict(X_test)
9  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.

```
1  # Modules for getting data ready
2  import pandas as pd
3  from sklearn.compose import ColumnTransformer
4  from sklearn.pipeline import Pipeline
5  from sklearn.impute import SimpleImputer
6  from sklearn.preprocessing import OneHotEncoder
7
8  # Modelling tools
9  from sklearn.ensemble import RandomForestRegressor
10 from sklearn.model_selection import train_test_split, GridSearchCV
11
12 # Setup random seed
13 import numpy as np
14 np.random.seed(42)
15
16 # Import data and drop rows with missing labels
17 data = pd.read_csv('data/car-sales-extended-missing-data.csv')
18 data.dropna(subset=['Price'], inplace=True)
19
20 # Define different features and transformer pipeline
21 categorical_features = ['Make', 'Color']
22 categorical_transformer = Pipeline(steps=[
23     ('imputer', SimpleImputer(strategy='constant', fill_value='missing')),
24     ('onehot', OneHotEncoder(handle_unknown='ignore'))]) # turn data into numbers
25
26 door_feature = ['Doors']
27 door_transformer = Pipeline(steps=[
28     ('imputer', SimpleImputer(strategy='constant', fill_value='4'))])
29
30 numeric_features = ['Odometer (KM)']
31 numeric_transformer = Pipeline(steps=[
32     ('imputer', SimpleImputer(strategy='mean'))])
33
34 # Setup preprocessing steps (fill missing values and convert to numeric)
35 preprocessor = ColumnTransformer(
36     transformer=[
37         ('cat', categorical_transformer, categorical_features),
38         ('door', door_transformer, door_feature),
39         ('num', numeric_transformer, numeric_features)]
40
41 # Create a preprocessing and modelling pipeline
42 model = Pipeline(steps=[('preprocessor', preprocessor),
43                          ('model', RandomForestRegressor()) ])
44
45 # Split data
46 X = data.drop('Price', axis=1)
47 y = data['Price']
48 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
49
50 # Fit and score the model
51 model.fit(X_train, y_train)
52 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.

```
1 # Use GridSearchCV with our regression Pipeline
2 from sklearn.model_selection import GridSearchCV
3
4 pipe_grid = {
5     'preprocessor__num__imputer__strategy': ['mean', 'median'],
6     'model__n_estimators': [100, 1000],
7     'model__max_depth': [None, 5],
8     'model__max_features': ['auto'], # 'model' step, change max_features of model
9     'model__min_samples_split': [2, 4]
10 } # 16 different combinations of parameters
11
12 gs_model = GridSearchCV(model, pipe_grid, cv=5, verbose=2) # 16*5 = 80 models
13 gs_model.fit(X_train, y_train)
14
15 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

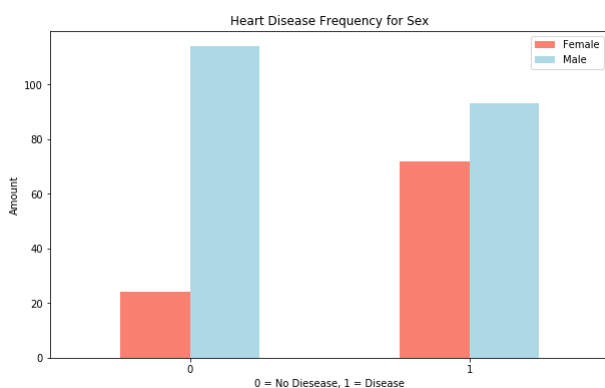
```
1 # Regular EDA (exploratory data analysis) and plotting libraries
2 import numpy as np
3 import pandas as pd
4 import matplotlib.pyplot as plt
5 import seaborn as sns
6
7 # Models and Evaluations from Scikit-Learn
8 from sklearn.linear_model import LogisticRegression
9 from sklearn.neighbors import KNeighborsClassifier
10 from sklearn.ensemble import RandomForestClassifier
11
12 from sklearn.model_selection import train_test_split, cross_val_score
13 from sklearn.model_selection import RandomizedSearchCV, GridSearchCV
14 from sklearn.metrics import confusion_matrix, classification_report
15 from sklearn.metrics import precision_score, recall_score, f1_score
16 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 expert 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?

```
1 df = pd.read_csv("../data/heart-disease.csv") # data folder from previous dir
2 df.shape # (303, 14)
3 df["target"].value_counts() # 1 (165), 0 (138). Even distribution
4 df.info() # show data types, number of rows/cols, etc.
5 df.isna().sum() # find any missing values in each column
6 df.describe() # get numerical analysis for each column
7
8 # Compare target column to the sex column
9 df.sex.value_count() # 1 (207 males), 0 (96 females)
10 pd.crosstab(df.target, df.sex).plot(kind="bar", figsize=(10, 6),
11                                     color=["salmon", "lightblue"])
12 plt.title("Heart Disease Frequency for Sex")
13 plt.xlabel("0 = No Disease, 1 = Disease")
14 plt.ylabel("Amount")
15 plt.legend(["Female", "Male"]);
16 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.

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

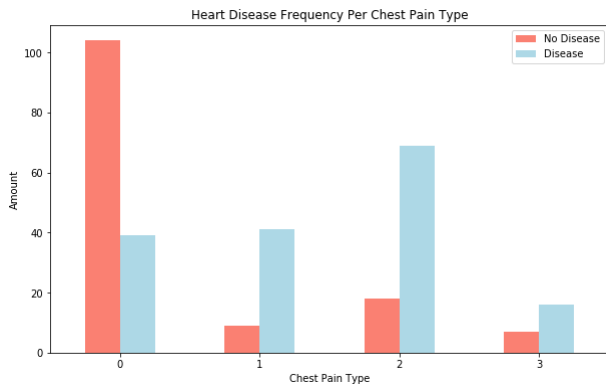


All we can see is a downward trend for both target values. As age increases, the maximum heart rate decreases in the individual. There is no real stand out pattern here, but it is a good analysis for understanding the data.

```

1 # Does chest pain (cp) correlate to having heart disease?
2 ### note that chest pain can have 4 different values (0, 1, 2, 3) see UCI for details
3
4 pd.crosstab(df.cp, df.target).plot(kind='bar', figsize=(10,6),
5                                     color=['salmon', 'lightblue'])
6
7 plt.title("Heart Disease Frequency Per Chest Pain Type")
8 plt.xlabel("Chest Pain Type")
9 plt.ylabel("Amount")
10 plt.legend(["No Disease", "Disease"])
11 plt.xticks(rotation=0);

```



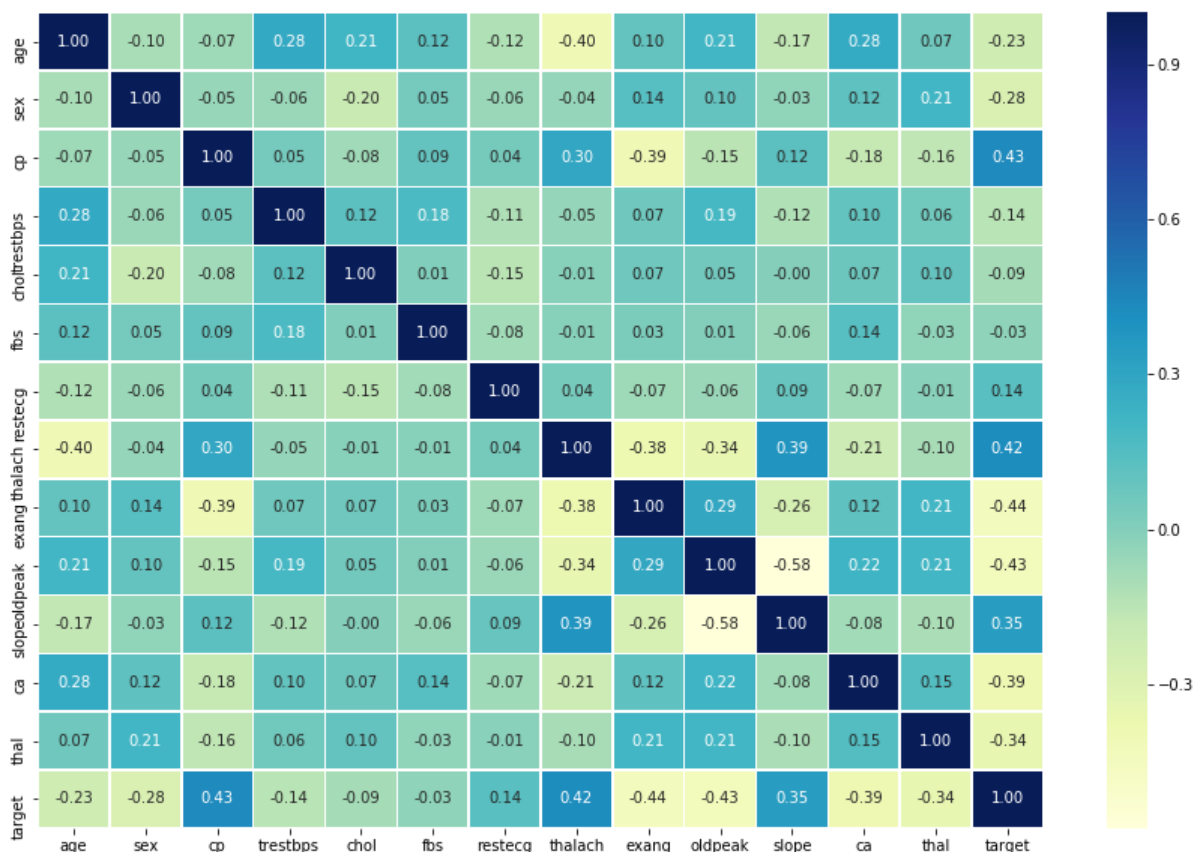
An interesting find is that for heart disease 1 and 2 (which are both suppose to be not related to the heart), we see that they both have very high numbers for the patients who have this chest pain and have a target value of 1.

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.

```

1 corr_matrix = df.corr()
2 fig, ax = plt.subplots(figsize=(15, 10))
3 ax = sns.heatmap(corr_matrix, annot=True,
4                 linewidths=0.5, fmt=".2f", cmap="YlGnBu");
5 bottom, top = ax.get_ylim() # get current limit values
6 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.

```
1  np.random.seed(42) # reproducible results
2
3  X = df.drop('target', axis=1)
4  y = df['target']
5  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.

```
1  models = {"Logistic Regression": LogisticRegression(),
2           "KNN": KNeighborsClassifier(),
3           "Random Forest": RandomForestClassifier()}
4
5  def fit_and_score(models, X_train, X_test, y_train, y_test):
6      """
7      Fits and evaluates given machine learning models.
8      models : a dict of different Scikit-Learn machine learning models
9      """
10     np.random.seed(42) # Set random seed
11     model_scores = {} # Make a dictionary to keep model scores
12     for name, model in models.items():
13         model.fit(X_train, y_train) # Fit the model to the data
14         model_scores[name] = model.score(X_test, y_test) # Evaluate append score
15     return model_scores
16
17 model_scores = fit_and_score(models, X_train, X_test, y_train, y_test)
18
19 {'Logistic Regression': 0.8852459016393442,
20  'KNN': 0.6885245901639344,
21  'Random Forest': 0.8360655737704918}
22 """
```

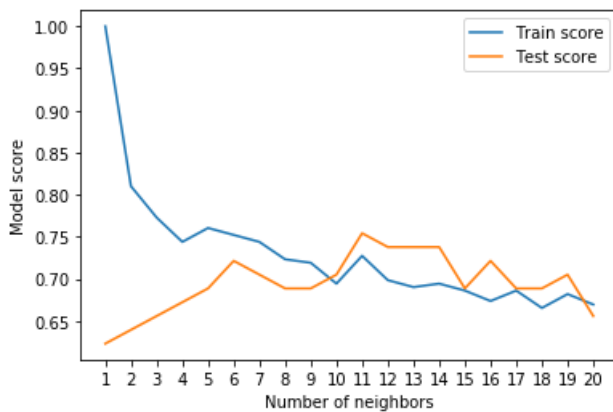
### 5.9.3 Tuning Hyperparameters

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

```
1  train_scores = []
2  test_scores = []
3
4  neighbors = range(1, 21) # Create a list of different values for n_neighbors
5  knn = KNeighborsClassifier() # Setup KNN instance
6
7  for i in neighbors: # Loop through different n_neighbors
8      knn.set_params(n_neighbors=i) # Set n_neighbors value
9      knn.fit(X_train, y_train) # Fit the algorithm
10     train_scores.append(knn.score(X_train, y_train)) # Update the training scores list
11     test_scores.append(knn.score(X_test, y_test)) # Update the test scores list
12
13 plt.plot(neighbors, train_scores, label="Train score")
14 plt.plot(neighbors, test_scores, label="Test score")
15 plt.xticks(np.arange(1, 21, 1))
16 plt.xlabel("Number of neighbors")
17 plt.ylabel("Model score")
18 plt.legend()
```

```
19 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`.

```
1 # Create a hyperparameter grid for LogisticRegression model
2 log_reg_grid = {"C": np.logspace(-4, 4, 20),
3                 "solver": ["liblinear"]}
4
5 # Create a hyperparameter grid for RandomForestClassifier model
6 rf_grid = {"n_estimators": np.arange(10, 1000, 50),
7            "max_depth": [None, 3, 5, 10],
8            "min_samples_split": np.arange(2, 20, 2),
9            "min_samples_leaf": np.arange(1, 20, 2)}
10
11 ##### Tune LogisticRegression #####
12 np.random.seed(42)
13
14 # Setup random hyperparameter search for LogisticRegression
15 rs_log_reg = RandomizedSearchCV(LogisticRegression(), # model to use
16                                param_distributions=log_reg_grid, # hyper grid
17                                cv=5, # 5 different data set splits
18                                n_iter=20, # 20 different models
19                                verbose=True) # 100 different fits
20
21 rs_log_reg.fit(X_train, y_train) # Fit random hyperparameter search model
22
23 rs_log_reg.best_params_ # {'solver': 'liblinear', 'C': 0.23357214690901212}
24 rs_log_reg.score(X_test, y_test) # 0.8852 (no increase in score)
25
26 ##### Tune RandomForestClassifier #####
27 np.random.seed(42)
28
29 # Setup random hyperparameter search for RandomForestClassifier
30 rs_rf = RandomizedSearchCV(RandomForestClassifier(), # model to use
31                             param_distributions=rf_grid, # hyper grid
32                             cv=5, # 5 different data set splits
33                             n_iter=20, # 20 different models
34                             verbose=True) # 100 different fits
35
36 rs_rf.fit(X_train, y_train) # Fit random hyperparameter search model
37
38 rs_rf.best_params_ # {'n_estimators': 210, 'min_samples_split': 4,
39                     # 'min_samples_leaf': 19, 'max_depth': 3}
40 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).

```

1 # Different hyperparameters for our LogisticRegression model
2 log_reg_grid = {"C": np.logspace(-4, 4, 30),
3                 "solver": ["liblinear"]}
4
5 # Setup grid hyperparameter search for LogisticRegression
6 gs_log_reg = GridSearchCV(LogisticRegression(),
7                             param_grid=log_reg_grid, # 30 different combinations
8                             cv=5, # 5 different data splits
9                             verbose=True) # 150 total fits
10
11 gs_log_reg.fit(X_train, y_train) # Fit grid hyperparameter search model
12
13 gs_log_reg.best_params_ # {'C': 0.20433597178569418, 'solver': 'liblinear'}
14 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.

```

1 y_preds = gs_log_reg.predict(X_test) # Make predictions with tuned model
2
3 # Plot ROC curve and calculate and calculate AUC metric
4 plot_roc_curve(gs_log_reg, X_test, y_test)

```

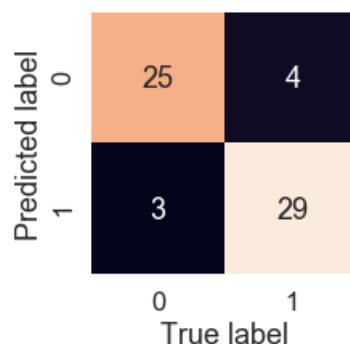
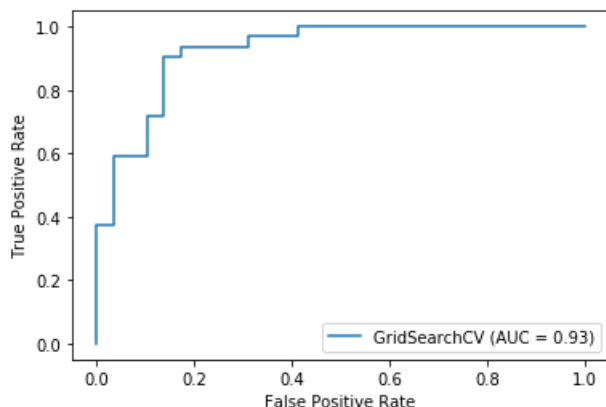
Next we want to see a **confusion matrix**, but we can use a heatmap to make it more easily readable.

```

1 sns.set(font_scale=1.5)
2
3 def plot_conf_mat(y_test, y_preds):
4     fig, ax = plt.subplots(figsize=(3, 3))
5     ax = sns.heatmap(confusion_matrix(y_test, y_preds), annot=True, cbar=False)
6     plt.xlabel("True label")
7     plt.ylabel("Predicted label")
8
9     bottom, top = ax.get_ylim()
10    ax.set_ylim(bottom + 0.5, top - 0.5)
11
12    plot_conf_mat(y_test, y_preds)

```

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





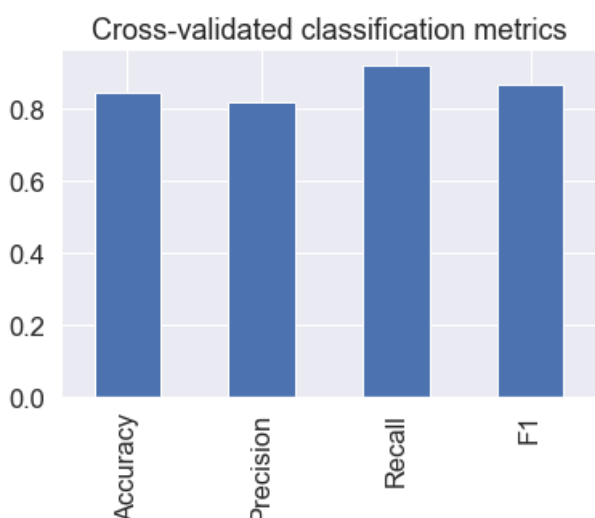
Now we will look at a **classification report**, which gives us precision, recall, and f1 score.

- *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).

```
1 classification_report(y_test, y_preds)
2 #      precision    recall  f1-score
3 #  0      0.89        0.86    0.88
4 #  1      0.88        0.91    0.89
5 # 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).

```
1 # Create a new classifier with best parameters
2 gs_log_reg.best_params_ # {'C': 0.20433597178569418, 'solver': 'liblinear'}
3 clf = LogisticRegression(C=0.20433597178569418, solver="liblinear")
4
5 # Cross-validated accuracy
6 cv_acc = cross_val_score(clf, X, y, cv=5, scoring="accuracy")
7 np.mean(cv_acc) # 0.8446994535519124
8
9 # Cross-validated precision
10 cv_precision = cross_val_score(clf, X, y, cv=5, scoring="precision")
11 np.mean(cv_precision) # 0.8207936507936507
12
13 # Cross-validated recall
14 cv_recall = cross_val_score(clf, X, y, cv=5, scoring="recall")
15 np.mean(cv_recall) # 0.9212121212121213
16
17 # Cross-validated f1-score
18 cv_f1 = cross_val_score(clf, X, y, cv=5, scoring="f1")
19 np.mean(cv_f1) # 0.8673007976269721
20
21 # Visualize cross-validated metrics
22 cv_metrics = pd.DataFrame({"Accuracy": cv_acc, "Precision": cv_precision,
23                           "Recall": cv_recall, "F1": cv_f1}, index=[0])
24
25 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).

```
1 # Fit an instance of LogisticRegression (with the best hyperparameters from earlier)
2 clf = LogisticRegression(C=0.20433597178569418, solver="liblinear")
3 clf.fit(X_train, y_train)
4
5 clf.coef_ # only used for LogisticRegression
6 # [[ 0.00316727, -0.86044582,  0.66067073, -0.01156993, -0.00166374,
7 #    0.04386131,  0.31275787,  0.02459361, -0.60413038, -0.56862852,
8 #    0.45051617, -0.63609863, -0.67663375]]
9
10 # Match coef's of features to columns from data frame (into an dict)
11 feature_dict = dict(zip(df.columns, list(clf.coef_[0])))
12
13 # Visualize feature importance
14 feature_df = pd.DataFrame(feature_dict, index=[0])
15 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.

```
1 pd.crosstab(df["sex"], df["target"])
2 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.

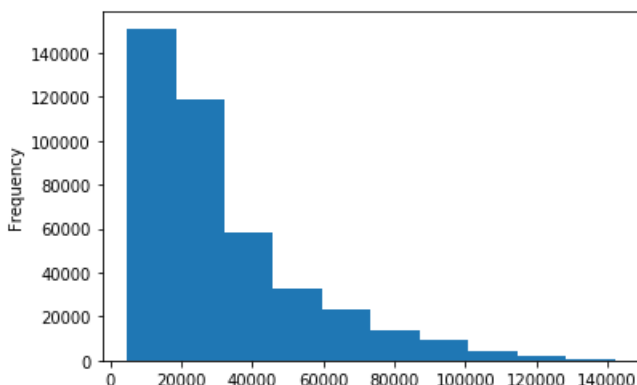
### Step 4: Features

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.

```
1 # Import Training and Validation Sets
2 df = pd.read_csv('data/bluebook-for-bulldozers/TrainAndValid.csv', low_memory=False)
3
4 df.info()
5 # 412,698 rows and 53 columns
6 # dtypes: float64 (3), int64 (5), object (45)
7 # memory usage: 166.9+ MB (very large file)
8
9 df.SalePrice.plot.hist()
```



Plotting the distribution of our target variable 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**.

```
1 # Import data again but this time parse dates
2 df = pd.read_csv("data/bluebook-for-bulldozers/TrainAndValid.csv",
3                 low_memory=False, parse_dates=["saledate"])
4
5 df.saledate.dtype # '<M8[ns]\' (same as datetime64[ns]) which is a datetime object
6 # 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.

```
1 # Sort DataFrame in date order
2 df.sort_values(by=["saledate"], inplace=True, ascending=True)
3
4 # Make a copy of the original DataFrame to perform edits on
5 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.

```
1 # Create new columns from attributes of the saledate column
2 df_tmp["saleYear"] = df_tmp.saledate.dt.year
3 df_tmp["saleMonth"] = df_tmp.saledate.dt.month
4 df_tmp["saleDay"] = df_tmp.saledate.dt.day
5 df_tmp["saleDayOfWeek"] = df_tmp.saledate.dt.dayofweek
6 df_tmp["saleDayOfYear"] = df_tmp.saledate.dt.dayofyear
7
8 # Now we've enriched our DataFrame with date time features, we can remove 'saledate'
9 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 on 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.

```
1 # This will turn all of the string value into category values
2 for label, content in df_tmp.items():
3     if pd.api.types.is_string_dtype(content): # returns T/F if column is a string
4         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.

```
1 df_tmp.state.cat.categories # returns an array of ordered values
2 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.

```
1 # Export current tmp dataframe
2 df_tmp.to_csv("data/bluebook-for-bulldozers/train_tmp.csv", index=False)
3
4 # Import preprocessed data
5 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.

```
1 # Check for which numeric columns have null values
2 for label, content in df_tmp.items():
3     if pd.api.types.is_numeric_dtype(content):
4         if pd.isnull(content).sum():
5             print(label) # auctioneerID, MachineHoursCurrentMeter
6
7 # Fill numeric rows with the median
8 for label, content in df_tmp.items():
9     if pd.api.types.is_numeric_dtype(content):
10        if pd.isnull(content).sum():
11            # Add a binary column which tells us if the data was missing or not
12            df_tmp[label+"_is_missing"] = pd.isnull(content)
13            # Fill missing numeric values with median
14            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.

```
1 # Check to see how many examples were missing
2 df_tmp.auctioneerID_is_missing.value_counts()
3 # False: 392562 (values that already existed)
4 # 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.

```
1 # Turn categorical variables into numbers and fill missing
2 for label, content in df_tmp.items():
3     if not pd.api.types.is_numeric_dtype(content):
4         # Add binary column to indicate whether sample had missing value
5         df_tmp[label+"_is_missing"] = pd.isnull(content)
6         # Turn categories into numbers and add +1
7         df_tmp[label] = pd.Categorical(content).codes+1
8
9 df_tmp.info() # 103 columns (increased from original 53 columns)
10 # bool(46), float64(3), int16(4), int64(10), int8(40)
11
12 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.

```
1  %%time # It took around 7 mins to go through 412698 observations
2
3  model = RandomForestRegressor(n_jobs=-1, random_state=42)
4  model.fit(df_tmp.drop("SalePrice", axis=1), df_tmp["SalePrice"])
5  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.

```
1  # Split data into training and validation
2  df_val = df_tmp[df_tmp.saleYear == 2012]
3  df_train = df_tmp[df_tmp.saleYear != 2012]
4  len(df_val), len(df_train) # (11573, 401125)
5
6  # Split data into X & y
7  X_train, y_train = df_train.drop("SalePrice", axis=1), df_train.SalePrice
8  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).

```
1  from sklearn.metrics import mean_squared_log_error, mean_absolute_error, r2_score
2
3  def rmsle(y_test, y_preds): # y_test same as y_true (used for train set)
4      """
5      Caculates root mean squared log error between predictions and true labels.
6      """
7      return np.sqrt(mean_squared_log_error(y_test, y_preds))
8
9  # Create function to evaluate model on a few different levels
10 def show_scores(model):
11     train_preds = model.predict(X_train) # use train data from previous code block
12     val_preds = model.predict(X_valid) # use valid data from previous code block
13     scores = {"Training MAE": mean_absolute_error(y_train, train_preds),
14              "Valid MAE": mean_absolute_error(y_valid, val_preds),
15              "Training RMSLE": rmsle(y_train, train_preds),
16              "Valid RMSLE": rmsle(y_valid, val_preds),
17              "Training R^2": r2_score(y_train, train_preds),
18              "Valid R^2": r2_score(y_valid, val_preds)}
19     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.

```
1 # Change max_samples value
2 model = RandomForestRegressor(n_jobs=-1, random_state=42, max_samples=10000)
3
4 %%time
5 # Cutting down the max number of samples each estimator can improved training time
6 model.fit(X_train, y_train)
7 # 16.6s to fit the model to a subset of 10000 samples of data
8
9 show_scores(model)
10 # {'Training MAE': 5561.2988092240585,
11 #   'Valid MAE': 7177.26365505919,
12 #   'Training RMSLE': 0.257745378256977,
13 #   'Valid RMSLE': 0.29362638671089003,
14 #   'Training R^2': 0.8606658995199189,
15 #   '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

```
1 %%time
2 from sklearn.model_selection import RandomizedSearchCV
3
4 # Different RandomForestRegressor hyperparameters
5 rf_grid = {"n_estimators": np.arange(10, 100, 10),
6            "max_depth": [None, 3, 5, 10],
7            "min_samples_split": np.arange(2, 20, 2),
8            "min_samples_leaf": np.arange(1, 20, 2),
9            "max_features": [0.5, 1, "sqrt", "auto"],
10           "max_samples": [10000]}
11
12 # Instantiate RandomizedSearchCV model
13 rs_model = RandomizedSearchCV(RandomForestRegressor(n_jobs=-1, random_state=42),
14                               param_distributions=rf_grid, n_iter=2, cv=5,
15                               verbose=True)
16
17 # Fit the RandomizedSearchCV model
18 rs_model.fit(X_train, y_train) # 1 min 49 seconds
19
20 rs_model.best_params_
21 # {'n_estimators': 90,
22 #   'min_samples_split': 16,
23 #   'min_samples_leaf': 11,
24 #   'max_samples': 10000,
25 #   'max_features': 'auto',
26 #   'max_depth': 10}
27
28 # Evaluate the RandomizedSearch model
29 show_scores(rs_model)
30 # {'Training MAE': 6633.714300615716,
31 #   'Valid MAE': 8074.634589086979,
32 #   'Training RMSLE': 0.2967133662928448,
33 #   'Valid RMSLE': 0.32142476459239144,
34 #   'Training R^2': 0.807156908286598,
35 #   '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.

```

1  %%time
2
3  # Most ideal hyperparamters
4  ideal_model = RandomForestRegressor(n_estimators=40, min_samples_leaf=1,
5                                     min_samples_split=14, max_features=0.5,
6                                     n_jobs=-1, max_samples=None, random_state=42)
7
8  # Fit the ideal model
9  ideal_model.fit(X_train, y_train) # took around 1 min 14 seconds
10
11 # Scores for ideal_model (trained on all the data)
12 show_scores(ideal_model)
13 # {'Training MAE': 2952.0893487566345,
14 #  'Valid MAE': 5959.9235776482565,
15 #  'Training RMSLE': 0.14457060318530182,
16 #  'Valid RMSLE': 0.2465443247528795,
17 #  'Training R^2': 0.9589437998919624,
18 #  '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.

```

1  # Import the test data
2  df_test = pd.read_csv("data/bluebook-for-bulldozers/Test.csv", low_memory=False,
3                        parse_dates=["saledate"])
4
5  def preprocess_data(df):
6      """
7      Performs transformations on df and returns transformed df.
8      """
9      df["saleYear"] = df.saledate.dt.year
10     df["saleMonth"] = df.saledate.dt.month
11     df["saleDay"] = df.saledate.dt.day
12     df["saleDayOfWeek"] = df.saledate.dt.dayofweek
13     df["saleDayOfYear"] = df.saledate.dt.dayofyear
14
15     df.drop("saledate", axis=1, inplace=True)
16
17     # Fill the numeric rows with median
18     for label, content in df.items():
19         if pd.api.types.is_numeric_dtype(content):
20             if pd.isnull(content).sum():
21                 # Add a binary column which tells us if the data was missing or not
22                 df[label+"_is_missing"] = pd.isnull(content)
23                 # Fill missing numeric values with median
24                 df[label] = content.fillna(content.median())
25
26     # Fill categorical missing data and turn categories into numbers
27     if not pd.api.types.is_numeric_dtype(content):
28         df[label+"_is_missing"] = pd.isnull(content)
29         # We add +1 to the category code because pd encodes missing categories as -1
30         df[label] = pd.Categorical(content).codes+1
31     return df
32
33 # Process the test data
34 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**.

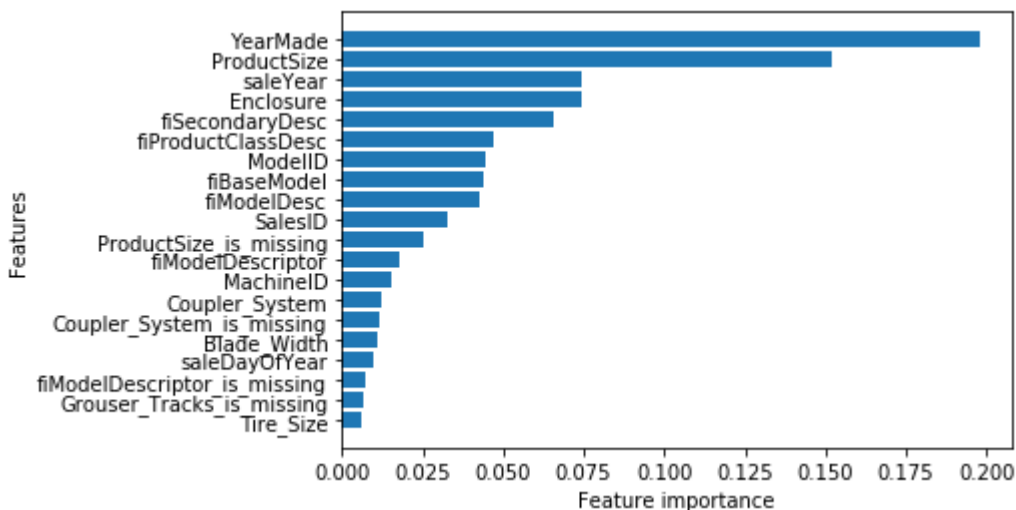
```
1 # We can find how the columns differ using sets
2 set(X_train.columns) - set(df_test.columns) # {'auctioneerID_is_missing'}
3
4 # Manually adjust df_test to have auctioneerID_is_missing column
5 df_test["auctioneerID_is_missing"] = False
```

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.

```
1 # Make predictions on the test data
2 test_preds = ideal_model.predict(df_test)
3
4 # Format predictions into a DataFrame
5 df_preds = pd.DataFrame()
6 df_preds["SalesID"] = df_test["SalesID"]
7 df_preds["SalesPrice"] = test_preds
8
9 # Export prediction data
10 df_preds.to_csv("data/bluebook-for-bulldozers/test_predictions.csv", index=False)
```

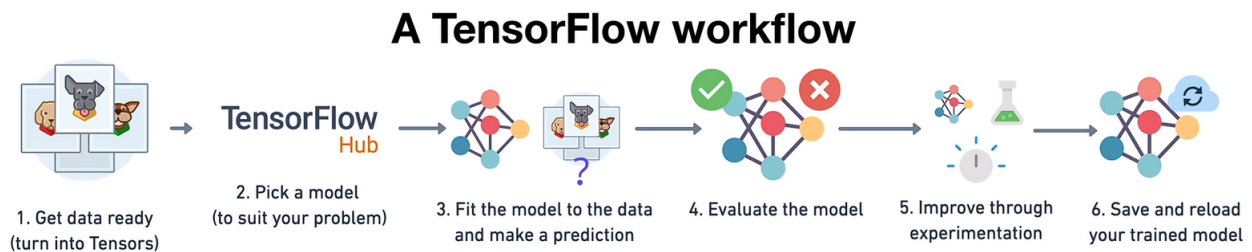
Finally, **feature importance** seeks to figure out which different attributes of the data were most importance when it comes to predicting the *target variable* (SalePrice).

```
1 # Helper function for plotting feature importance
2 def plot_features(columns, importances, n=20):
3     df = (pd.DataFrame({"features": columns,
4                         "feature_importances": importances})
5           .sort_values("feature_importances", ascending=False) # sort high to low
6           .reset_index(drop=True))
7
8     # Plot the dataframe (only top 20 of the total 102 features)
9     fig, ax = plt.subplots()
10    ax.barh(df["features"][:n], df["feature_importances"][:n])
11    ax.set_ylabel("Features")
12    ax.set_xlabel("Feature importance")
13    ax.invert_yaxis()
14
15    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:

```
1 # Unzip the uploaded data into Google Drive (-d is the destination)
2 !unzip "drive/My Drive/Dog Vision/dog-breed-identification.zip" -d "drive/My Drive/
3     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

```
1 # Import necessary tools
2 import tensorflow as tf
3 import tensorflow_hub as hub
4 print("TF version:", tf.__version__) # TF version: 2.1.0
5 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).

```
1 # Import data and check labels
2 import pandas as pd
3 labels_csv = pd.read_csv("drive/My Drive/Dog Vision/labels.csv")
4 print(labels_csv.describe())
5 # 10222 count for both columns (id and breed)
6 # 10222 unique values for id, 120 unique values for breed
7
8 # How many images are there of each breed?
9 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.

```
1 # Let's manually view an image in our folder
2 from IPython.display import Image
3 Image("drive/My Drive/Dog Vision/train/001513dfcb2ffafc82cccf4d8bbaba97.jpg")
4
5 # Create pathnames from image ID's (similar to the one above)
6 filenames = ["drive/My Drive/Dog Vision/train/" + fname + ".jpg" for fname
7              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.

```
1 # Convert breed labels into NumPy array
2 import numpy as np
3 labels = labels_csv["breed"].to_numpy()
4 len(labels) # 10222
5
6 # Find the unique label values
7 unique_breeds = np.unique(labels)
8 len(unique_breeds) # 120
9
10 # Turn every label into a boolean array
11 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).

```
1 # Set number of images to use for experimenting
2 NUM_IMAGES = 1000
3
4 # Split them into training and validation of total size NUM_IMAGES
5 from sklearn.model_selection import train_test_split
6 X_train, X_val, y_train, y_val = train_test_split(X[:NUM_IMAGES], y[:NUM_IMAGES],
7                                                  test_size=0.2, random_state=42)
8
9 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).

```
1  # Example of how to convert image to NumPy array
2  from matplotlib.pyplot import imread
3  image = imread(filenamees[42])
4  image.shape # (257, 350, 3)
5
6  IMG_SIZE = 224 # Define image size
7
8  # Create a function for preprocessing images
9  def process_image(image_path, img_size=IMG_SIZE):
10     """
11     Takes an image file path and turns the image into a Tensor.
12     """
13     image = tf.io.read_file(image_path) # Read in an image file into a Tensor string
14     # Turn the jpeg image into numerical Tensor with 3 colour channels (RGB)
15     image = tf.image.decode_jpeg(image, channels=3)
16     # Convert the colour channel values from 0-255 to 0-1 values (normalize)
17     image = tf.image.convert_image_dtype(image, tf.float32)
18     # Resize the image to our desired value (224, 224)
19     image = tf.image.resize(image, size=[IMG_SIZE, IMG_SIZE])
20
21     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.

```
1  # Create a simple function to return a tuple (image, label)
2  def get_image_label(image_path, label):
3      """
4      Takes an image file path name and the associated label,
5      processes the image and returns a tuple of (image, label).
6      """
7      image = process_image(image_path)
8      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.

```

1 BATCH_SIZE = 32 # Define the batch size, 32 is a good start
2
3 # Create a function to turn data into batches
4 def create_data_batches(X, y=None, batch_size=BATCH_SIZE, valid_data=False,
5                         test_data=False):
6     """
7     Creates batches of data out of image (X) and label (y) pairs.
8     Shuffles the data if it's training data but doesn't shuffle if it's validation data
9     Also accepts test data as input (no labels).
10    """
11    if test_data: # If the data is a test dataset, we probably don't have labels
12        print("Creating test data batches...")
13        data = tf.data.Dataset.from_tensor_slices((tf.constant(X))) # only filepaths
14        data_batch = data.map(process_image).batch(BATCH_SIZE)
15        return data_batch
16
17    elif valid_data: # If the data is a valid dataset, we don't need to shuffle it
18        print("Creating validation data batches...")
19        data = tf.data.Dataset.from_tensor_slices((tf.constant(X), # filepaths
20                                                  tf.constant(y))) # labels
21        data_batch = data.map(get_image_label).batch(BATCH_SIZE)
22        return data_batch
23
24    else:
25        print("Creating training data batches...")
26        # Turn filepaths and labels into Tensors
27        data = tf.data.Dataset.from_tensor_slices((tf.constant(X), tf.constant(y)))
28
29        # Shuffling pathnames and labels before mapping image processor function is
30        # faster than shuffling images
31        data = data.shuffle(buffer_size=len(X))
32
33        # Create (image, label) tuples (this also turns the image path into a
34        # preprocessed image)
35        data = data.map(get_image_label)
36
37        # Turn the training data into batches
38        data_batch = data.batch(BATCH_SIZE)
39        return data_batch
40
41 # Create training and validation data batches
42 train_data = create_data_batches(X_train, y_train)
43 val_data = create_data_batches(X_val, y_val, valid_data=True)
44
45 # Create a function for viewing images in a data batch
46 import matplotlib.pyplot as plt
47 def show_25_images(images, labels):
48     """
49     Displays a plot of 25 images and their labels from a data batch.
50     """
51     plt.figure(figsize=(10, 10)) # Setup the figure
52     # Loop through 25 (for displaying 25 images)
53     for i in range(25):
54         ax = plt.subplot(5, 5, i+1) # Create subplots (5 rows, 5 columns, index+1)
55         plt.imshow(images[i]) # Display an image
56         plt.title(unique_breeds[labels[i].argmax()]) # Add the image label as the title
57         plt.axis("off") # Turn the grid lines off
58
59 # Now let's visualize the data in a training batch
60 train_images, train_labels = next(train_data.as_numpy_iterator())
61 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**.

```
1 # Setup input shape to the model
2 INPUT_SHAPE = [None, IMG_SIZE, IMG_SIZE, 3] # batch, height, width, color channels
3
4 # Setup output shape of our model
5 OUTPUT_SHAPE = len(unique_breeds)
6
7 # Setup model URL from TensorFlow Hub
8 MODEL_URL = "https://tfhub.dev/google/imagenet/mobilenet_v2_130_224/classification/4"
9
10 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.

```
1 # Create a function which builds a Keras model
2 def create_model(input_shape=INPUT_SHAPE, output_shape=OUTPUT_SHAPE, model_url=
  MODEL_URL):
3     print("Building model with:", MODEL_URL)
4
5     # Setup the model layers
6     model = tf.keras.Sequential([hub.KerasLayer(MODEL_URL), # Layer 1 (input layer)
7                                 tf.keras.layers.Dense(units=OUTPUT_SHAPE,
8                                                         activation="softmax")
9                                 ]) # Layer 2 (output layer)
10
11     # Compile the model
12     model.compile(loss=tf.keras.losses.CategoricalCrossentropy(),
13                  optimizer=tf.keras.optimizers.Adam(),
14                  metrics=["accuracy"])
15
16     # Build and return the model
17     model.build(INPUT_SHAPE) # size of images MobileNetV2 was trained on
18     return model
19
20 model = create_model()
21 model.summary()
```

Building model with: [https://tfhub.dev/google/imagenet/mobilenet\\_v2\\_130\\_224/classification/4](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).

```
1 # Load TensorBoard notebook extension
2 %load_ext tensorboard
3
4 import datetime
5
6 # Create a function to build a TensorBoard callback
7 def create_tensorboard_callback():
8     # Create a log directory for storing TensorBoard logs
9     logdir = os.path.join("drive/My Drive/Dog Vision/logs",
10                           # The logs get tracked whenever we run an experiment
11                           datetime.datetime.now().strftime("%Y%m%d-%H%M%S"))
12     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.

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

```
1 NUM_EPOCHS = 100 #@param {type:"slider", min:10, max:100, step:10}
2
3 # Check to make sure we're still running on a GPU
4 print("GPU", "available." if tf.config.list_physical_devices("GPU") else "not
5         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

```
1 # Build a function to train and return a trained model
2 def train_model():
3     """
4     Trains a given model and returns the trained version.
5     """
6     # Create a model
7     model = create_model()
8
9     # Create new TensorBoard session everytime we train a model
10    tensorboard = create_tensorboard_callback()
11
12    # Fit the model to the data passing it the callbacks we created
13    model.fit(x=train_data,
14             epochs=NUM_EPOCHS,
15             validation_data=val_data,
16             validation_freq=1, # how often we test patterns per epoch
17             callbacks=[tensorboard, early_stopping])
18
19    # Return the fitted model
20    return model
21
22 # Fit the model to the data
23 model = train_model()
```

Building model with: [https://tfhub.dev/google/imagenet/mobilenet\\_v2\\_130\\_224/classification/4](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.

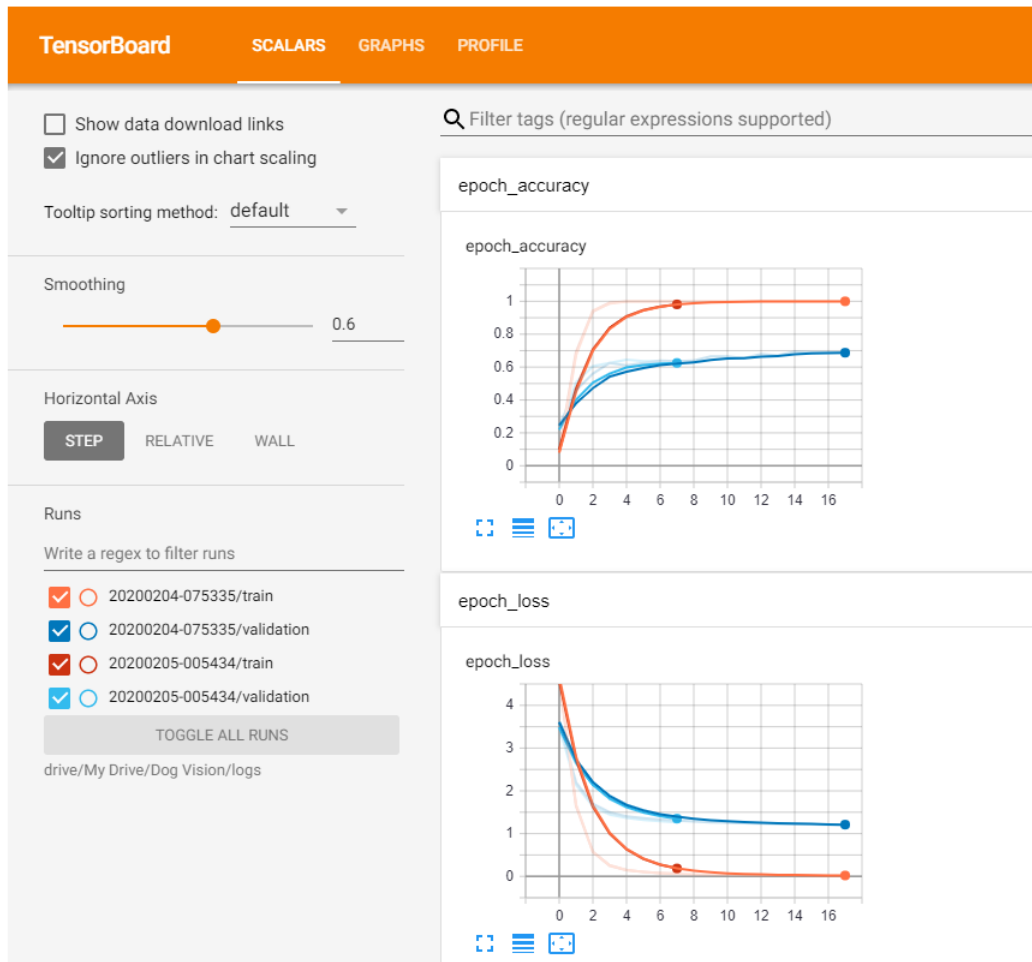


### 6.4.2 Evaluating Performance with TensorBoard

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

```
1 %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

```
1 # Make predictions on the validation data (not used to train on)
2 predictions = model.predict(val_data, verbose=1) # verbose shows us progress bar
3 len(y_val) # 200
4 predictions.shape # (200, 120) meaning 200 arrays with 120 values in each
5
6 index = 42
7 print(f"Max value (probability of prediction): {np.max(predictions[index])}")
8 print(f"Sum: {np.sum(predictions[index])}") # sum to 1 since softmax
9 print(f"Max index: {np.argmax(predictions[index])}")
10 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).

```
1 # Turn prediction probabilities into their respective label (easier to understand)
2 def get_pred_label(prediction_probabilities):
3     """
4     Turns an array of prediction probabilities into a label.
5     """
6     return unique_breeds[np.argmax(prediction_probabilities)]
7
8 # Create a function to unbatch a batch dataset
9 def unbatchify(data):
10     """
11     Takes a batched dataset of (image, label) Tensors and reutrns separate arrays
12     of images and labels.
13     """
14     images = []
15     labels = []
16     # Loop through unbatched data
17     for image, label in data.unbatch().as_numpy_iterator():
18         images.append(image)
19         labels.append(unique_breeds[np.argmax(label)])
20     return images, labels
21
22 # Unbatchify the validation data
23 val_images, val_labels = unbatchify(val_data)
24
25 # Get label for unbatched validation set
26 get_pred_label(val_labels[0]) # 'cairn'
```

### 6.5.2 Visualize Model Predictions

Now we've got a function that can get us:

- Prediction labels
- Validation labels (truth labels)
- 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.

The below function will output the predicted label, confidence level, truth label, and image. If it predicted correctly, the title will be green but if it predicted wrong, then the title with be red.

```

1 def plot_pred(prediction_probabilities, labels, images, n=1):
2     """
3     View the prediction, ground truth and image for sample n
4     """
5     pred_prob, true_label, image = prediction_probabilities[n], labels[n], images[n]
6
7     # Get the pred label
8     pred_label = get_pred_label(pred_prob)
9
10    # Plot image & remove ticks
11    plt.imshow(image)
12    plt.xticks([])
13    plt.yticks([])
14
15    # Change the color of the title depending on if the prediction is right or wrong
16    if pred_label == true_label:
17        color = "green"
18    else:
19        color = "red"
20
21    # Change plot title to be predicted, probability of prediction and truth label
22    plt.title("{} {:.20f}% {}".format(pred_label, np.max(pred_prob)*100,
23                                     true_label), color=color)

```

Now we've got one function to visualize our models top prediction, let's make another to view our models top 10 predictions. This function will:

- Take an input of prediction probabilities array and a ground truth array and an integer.
- Find the prediction using 'get\_pred\_label()'.
- Find the top 10: Prediction probabilities indexes , Prediction probabilities values, Prediction labels.
- Plot the top 10 prediction probability values and labels, coloring the true label green.

```

1 def plot_pred_conf(prediction_probabilities, labels, n=1):
2     """
3     Plot the top 10 highest prediction confidences and the truth label for sample n.
4     """
5     pred_prob, true_label = prediction_probabilities[n], labels[n]
6
7     # Get the predicted label
8     pred_label = get_pred_label(pred_prob)
9
10    # Find the top 10 prediction confidence indexes
11    top_10_pred_indexes = pred_prob.argsort()[-10:][::-1] # [::-1] descending order
12    # Find the top 10 prediction confidence values
13    top_10_pred_values = pred_prob[top_10_pred_indexes]
14    # Find the top 10 prediction labels
15    top_10_pred_labels = unique_breeds[top_10_pred_indexes]
16
17    # Setup plot
18    top_plot = plt.bar(np.arange(len(top_10_pred_labels)),
19                      top_10_pred_values,
20                      color="grey")
21    plt.xticks(np.arange(len(top_10_pred_labels)),
22              labels=top_10_pred_labels,
23              rotation="vertical")
24
25    # Change color of true label
26    if np.isin(true_label, top_10_pred_labels):
27        top_plot[np.argmax(top_10_pred_labels == true_label)].set_color("green")
28    else:
29        pass

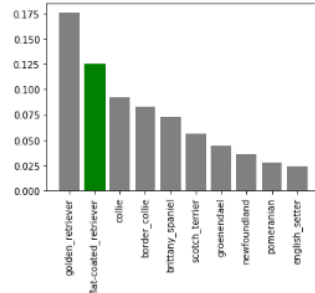
```

```

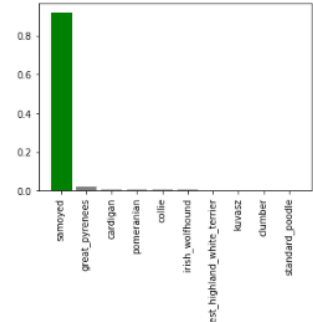
1 # Let's check out a few predictions and their different values
2 i_multiplier = 20
3 num_rows = 3
4 num_cols = 2
5 num_images = num_rows*num_cols
6 plt.figure(figsize=(10*num_cols, 5*num_rows))
7 for i in range(num_images):
8     plt.subplot(num_rows, 2*num_cols, 2*i+1)
9     plot_pred(prediction_probabilities=predictions,
10              labels=val_labels,
11              images=val_images,
12              n=i+i_multiplier)
13     plt.subplot(num_rows, 2*num_cols, 2*i+2)
14     plot_pred_conf(prediction_probabilities=predictions,
15                   labels=val_labels,
16                   n=i+i_multiplier)
17 plt.tight_layout(h_pad=1.0)
18 plt.show() # be sure to call outside loop (dont show each time)

```

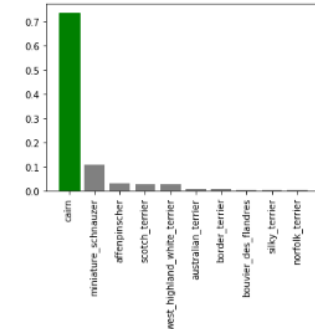
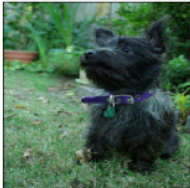
golden\_retriever 18% flat-coated\_retriever



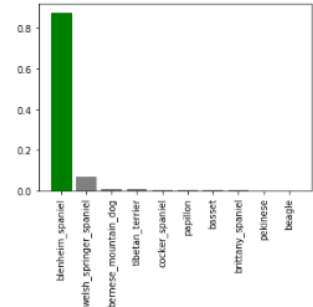
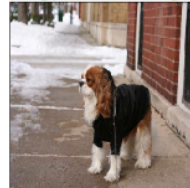
samoyed 92% samoyed



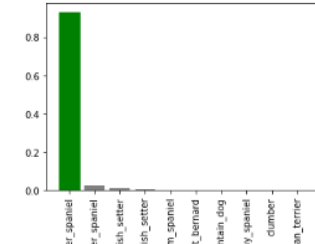
cairn 73% cairn



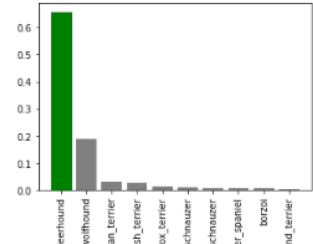
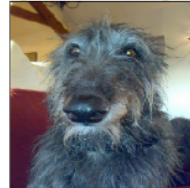
blenheim\_spaniel 87% blenheim\_spaniel



welsh\_springer\_spaniel 93% welsh\_springer\_spaniel



scottish\_deerhound 66% scottish\_deerhound



## 6.6 Step 5: Improving our Model (on full dataset)

Previously, we split our data into a subset of 1000 images to train on. Our `X_train` has 800 images and our `X_val` has 200 images. Now, we want to train a model on the full 10,000 images rather than just 800.

```

1 # Create a data batch with the full data set
2 full_data = create_data_batches(X, y)
3 full_data # <BatchDataset shapes: ((None, 224, 224, 3), (None, 120)),
4           # types: (tf.float32, tf.bool)>

```

```

1 # Create a model for full model
2 full_model = create_model()
3
4 # Create full model callbacks
5 full_model_tensorboard = create_tensorboard_callback()
6 # No val set when training on all the data, so we can't monitor validation accuracy
7 full_model_early_stopping = tf.keras.callbacks.EarlyStopping(monitor="accuracy",
8                                                                patience=3)

```

Running the cell below will take a little while (maybe up to 30 minutes for the first epoch) because the GPU we're using in the runtime has to load all of the images into memory (around 10,000 images). The model stopped at epoch 17, where the accuracy plateaued at around 99.87% accuracy for 3 epochs.

```

1 # Fit the full model to the full data
2 full_model.fit(x=full_data, epochs=NUM_EPOCHS,
3               callbacks=[full_model_tensorboard, full_model_early_stopping])

```

Train for 320 steps

```

Epoch 1/100
320/320 [=====] - 4730s 15s/step - loss: 1.3120 - accuracy: 0.6787
Epoch 2/100
320/320 [=====] - 54s 168ms/step - loss: 0.4011 - accuracy: 0.8795
Epoch 3/100
320/320 [=====] - 52s 164ms/step - loss: 0.2411 - accuracy: 0.9337
Epoch 4/100
320/320 [=====] - 54s 168ms/step - loss: 0.1565 - accuracy: 0.9635
Epoch 5/100
320/320 [=====] - 50s 155ms/step - loss: 0.1082 - accuracy: 0.9782
Epoch 6/100
320/320 [=====] - 47s 146ms/step - loss: 0.0783 - accuracy: 0.9857
Epoch 7/100
320/320 [=====] - 47s 146ms/step - loss: 0.0589 - accuracy: 0.9922
Epoch 8/100
320/320 [=====] - 47s 146ms/step - loss: 0.0457 - accuracy: 0.9942
Epoch 9/100
320/320 [=====] - 47s 146ms/step - loss: 0.0366 - accuracy: 0.9968
Epoch 10/100
320/320 [=====] - 47s 146ms/step - loss: 0.0304 - accuracy: 0.9969
Epoch 11/100
320/320 [=====] - 46s 145ms/step - loss: 0.0274 - accuracy: 0.9978
Epoch 12/100
320/320 [=====] - 47s 147ms/step - loss: 0.0234 - accuracy: 0.9977
Epoch 13/100
320/320 [=====] - 48s 149ms/step - loss: 0.0194 - accuracy: 0.9984
Epoch 14/100
320/320 [=====] - 49s 152ms/step - loss: 0.0177 - accuracy: 0.9988
Epoch 15/100
320/320 [=====] - 49s 153ms/step - loss: 0.0157 - accuracy: 0.9987
Epoch 16/100
320/320 [=====] - 49s 153ms/step - loss: 0.0152 - accuracy: 0.9987
Epoch 17/100
320/320 [=====] - 49s 154ms/step - loss: 0.0135 - accuracy: 0.9987
<tensorflow.python.keras.callbacks.History at 0x7fdaa6bf5128>

```

The above models first epoch took around 80 minutes to complete, where each epoch following was less than a minute. It is good practice to have a 'save model' call after you finish fitting the model so that it saves it if the model successfully trains.

We should be a little skeptic of the above accuracy because we cannot validate it with another set. Instead we can use the test data to make predictions and submit them to Kaggle. We will do this in a future section.

## 6.7 Step 6: Saving and Loading a Trained Model

```
1 # Create a function to save a model
2 def save_model(model, suffix=None):
3     """
4     Saves a given model in a models directory and appends a suffix (string).
5     """
6     # Create a model directory pathname with current time
7     model_dir = os.path.join("drive/My Drive/Dog Vision/models",
8                               datetime.datetime.now().strftime("%Y%m%d-%H%M%S"))
9     model_path = model_dir + "-" + suffix + ".h5" # save format of model
10    print(f"Saving model to: {model_path}...")
11    model.save(model_path)
12    return model_path
13
14 # Create a function to load a trained model
15 def load_model(model_path):
16     """
17     Loads a saved model from a specified path.
18     """
19    print(f"Loading saved model from: {model_path}")
20    model = tf.keras.models.load_model(model_path,
21                                         custom_objects={"KerasLayer": hub.KerasLayer})
22    return model
```

Now we've got functions to save and load a trained model, let's make sure they work.

```
1 # Save our model trained on full data (10,000 images)
2 save_model(full_model, suffix="full-image-set-mobilenetv2-Adam")

Saving model to: drive/My Drive/Dog Vision/models/20200205-07041580886291-full-image-set-mobilenetv2-Adam.h5...
'drive/My Drive/Dog Vision/models/20200205-07041580886291-full-image-set-mobilenetv2-Adam.h5'

1 # Load in the full model
2 loaded_full_model = load_model('drive/My Drive/Dog Vision/models/20200205-
3                               07041580886291-full-image-set-mobilenetv2-
4                               Adam.h5')
```

```
Loading saved model from: drive/My Drive/Dog Vision/models/20200205-07041580886291-full-image-set-mobilenetv2-Adam.h5
WARNING:tensorflow:Sequential models without an 'input_shape' passed to the first layer cannot reload their optimizer state. As a result, your model is starting with a freshly initialized optimizer.
WARNING:tensorflow:Sequential models without an 'input_shape' passed to the first layer cannot reload their optimizer state. As a result, your model is starting with a freshly initialized optimizer.
```

## 6.8 Making Predictions on Test Data

Since our model has been trained on images in the form of Tensor batches, to make predictions on the test data, we'll have to get it into the same format. Luckily we created 'create\_data\_batches()' earlier which can take a list of filenames as input and convert them into Tensor batches. To make predictions on the test data, we'll:

- 1) Get the test image filenames.
- 2) Convert the filenames into test data batches using 'create\_data\_batches()' and setting the 'test\_data' parameter to 'True' (since the test data doesn't have labels).
- 3) Make a predictions array by passing the test batches to the 'predict()' method called on our model.

```
1 # Load test image filenames
2 test_path = "drive/My Drive/Dog Vision/test/"
3 test_filenames = [test_path + fname for fname in os.listdir(test_path)]
4 len(test_filenames) # 10357
5
6 # Create test data batch
7 test_data = create_data_batches(test_filenames, test_data=True)
8 test_data # <BatchDataset shapes: (None, 224, 224, 3), types: tf.float32>
```

Calling `predict()` on our full model and passing it the test data batch will take a long time to run (about one hour). This is because we have to process approximately 10,000+ images and get our model to find patterns in those images and generate predictions based on what its learned in the training dataset.

```

1  # Make predictions on test data batch using the loaded full model
2  test_predictions = loaded_full_model.predict(test_data, verbose=1)
3
4  # Save predictions (NumPy array) to csv file (for access later)
5  np.savetxt("drive/My Drive/Dog Vision/preds_array.csv",
6             test_predictions, delimiter=",")
7
8  # Load predictions (NumPy array) from csv file
9  test_predictions = np.loadtxt("drive/My Drive/Dog Vision/preds_array.csv",
10                               delimiter=",")
11
12 test_predictions.shape # (10357, 120)

```

### 6.8.1 Preparing Test Predictions for Kaggle

Looking at the [Kaggle Sample submission](#), we find that it wants our models prediction probability outputs in a DataFrame with an ID and a column for each dog breed. To get the data in this format, we'll:

- Create a pandas DataFrame with an ID column as well as a column for each dog breed.
- Add data to the ID column by extracting the test image ID's from their file-paths.
- Add data (the prediction probabilities) to each of the dog breed columns.
- Export the DataFrame as a CSV to submit it to Kaggle.

```

1  # Create a pandas DataFrame with empty columns
2  preds_df = pd.DataFrame(columns=["id"] + list(unique_breeds))
3
4  # Append test image ID's to predictions DataFrame
5  test_ids = [os.path.splitext(path)[0] for path in os.listdir(test_path)]
6  preds_df["id"] = test_ids
7
8  # Add the prediction probabilities to each dog breed column
9  preds_df[list(unique_breeds)] = test_predictions
10
11 # Save our predictions dataframe to CSV for submission to Kaggle
12 preds_df.to_csv("drive/My Drive/Dog Vision/full_model_predictions_
13                 submission_1_mobilenetV2.csv", index=False)

```

	id	affenpinscher	afghan_hound	african_hunting_dog	airedale	american_staffordshire_terrier	appenzeller	aust
0	ea5ae600ca3996528b6548eaceac767cf	1.61196e-09	3.44086e-12	2.32834e-11	1.40799e-12	5.98033e-09	2.14534e-11	
1	eb61a828e939eb633c1811f283d15647	3.17894e-10	3.20088e-14	1.85375e-10	1.95466e-13	8.68222e-07	3.36228e-07	
2	efb4e5dc776a639a140b022cf0835eb5	4.27301e-09	1.8414e-13	1.11785e-09	1.10664e-10	7.22501e-11	1.16924e-05	
3	f16f47570b2ef2c21c54eb1478518e8f	5.22772e-14	3.37155e-14	1.63818e-11	5.28355e-12	7.16004e-08	1.8326e-13	
4	eb3849def6fa245bcaae9ab9c64f024f	9.61375e-11	1.30428e-08	1.24693e-08	1.47886e-07	6.65842e-09	6.37447e-12	

5 rows × 121 columns



## 6.8.2 Predicting on Custom Images

To make predictions on custom images, we'll:

- 1) Get the file-paths of our own images.
- 2) Turn the file-paths into data batches using 'create\_data\_batches()'. And since our custom images won't have labels, we set the 'test\_data' parameter to 'True'.
- 3) Pass the custom image data batch to our model's 'predict()' method.
- 4) Convert the prediction output probabilities to predictions labels.
- 5) Compare the predicted labels to the custom images.

```
1 # Get custom image filepaths
2 custom_path = "drive/My Drive/Dog Vision/my-dog-photos/"
3 custom_image_paths = [custom_path + fname for fname in os.listdir(custom_path)]
4
5 # Turn custom images into batch datasets (test_data=True since no labels)
6 custom_data = create_data_batches(custom_image_paths, test_data=True)
7 custom_data # <BatchDataset shapes: (None, 224, 224, 3), types: tf.float32>
8
9 # Make predictions on the custom data
10 custom_preds = loaded_full_model.predict(custom_data)
11 custom_preds.shape # (3, 120)
12
13 # Get custom image prediction labels
14 custom_pred_labels = [get_pred_label(custom_preds[i]) for i in
15                       range(len(custom_preds))]
16 custom_pred_labels # ['lakeland-terrier', 'golden-retriever', 'labrador-retriever']
17
18 # Get custom images (our unbatchify() function won't work since there aren't labels)
19 custom_images = []
20 # Loop through unbatched data
21 for image in custom_data.unbatch().as_numpy_iterator():
22     custom_images.append(image)
23
24 # Check custom image predictions
25 plt.figure(figsize=(10, 10))
26 for i, image in enumerate(custom_images): # saves count to i, image to image
27     plt.subplot(1, 3, i+1)
28     plt.xticks([])
29     plt.yticks([])
30     plt.title(custom_pred_labels[i])
31     plt.imshow(image)
```



For the above images, the first prediction is wrong (is it the baby picture of the last picture). The second and third guesses are both correct. Some ways to improve the model include:

- Trying a new model from TensorFlow Hub that would perform better (including pretrained models).
- Take the training images and manipulate (crop, resize) or distort them (flip, rotate) to create even more training data for the model to learn from. Check out the [TensorFlow images documentation](#) for a whole bunch of functions you can use on images.
- Use what the model already knows and fine-tune this knowledge to our own dataset (pictures of dogs). This would mean all of the patterns within the model would be updated to be more specific to pictures of dogs rather than general images. ([click here for more resources](#)).