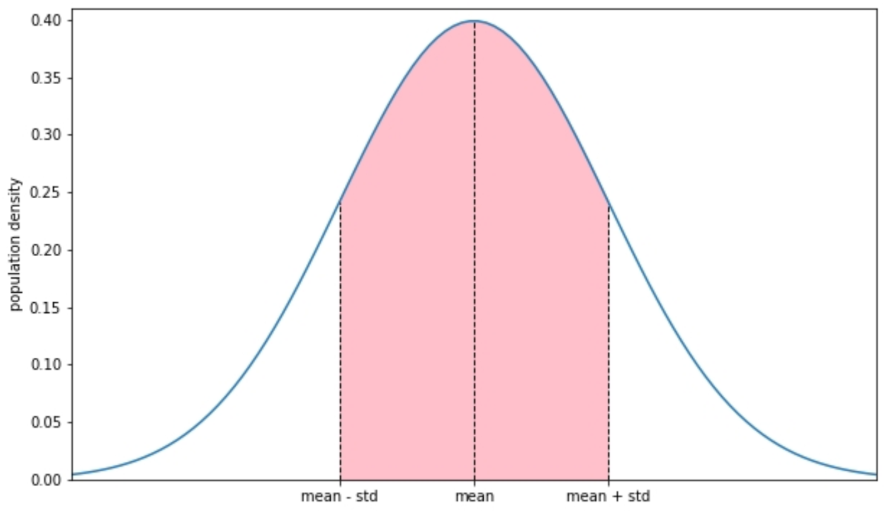
Overview of Machine Learning:

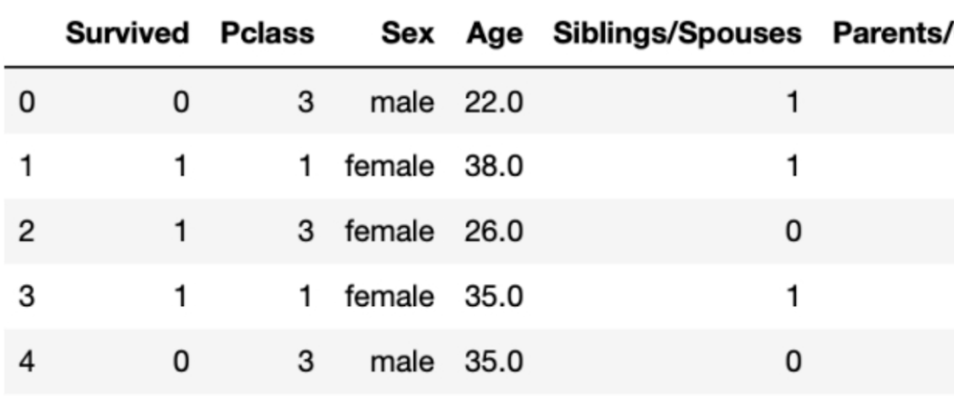
The Basics:

- Welcome to Machine Learning:
    - The main language that the course is going to use is Python
- Machine Learning Overview:
    - ML is a way to take data and turn it into insight
    - Use computer power to analyze examples from the past to build a model that can predict the result for new example
    - ML examples range from recommended for Netflix, Amazon choosing pricing for goods, chatbot, detecting spam, image recognition, etc
    - Course Basics:
        - Most common language used Python since it is very approachable and very powerful
        - Helpful packages:
            - Pandas:
                - Used for reading data and doing data manipulations
            - Numpy:
                - Used for computation of numerical data
            - Matplotlib:
                - Used for graphing data
            - Scikit-learn:
                - Used for machine learning model
        - Covers both theory and practice of machine learning techniques and applications
- What the Course contains:
    - There are two types of learning:
        - Supervised:
            - Takes place when we have a known target based on past data
            - What the course will mainly focus on
            - Ex:
                - Predicint price a house will sell by using previous data from the past
        - Unsupervised:
            - Takes place when there isn't a known past answer
            - Ex:
                - Determining the topics discussed in a restaurant review
    - Regression:
        - Predicting a numerical value:
        - Ex:
            - Predicting the price a house will sell for
    - Classification:
        - Predicting what class something belongs to
        - Mainly be focusing on this topic

        - Ex:
            - Predicting if a borrower will default on their loan
    - Topics like logistic regression, decision trees, random forests, and neural networks will be explored
- Statistics Review:
    - Goes over basic concepts like mean, median, etc
    - Mean & median -> averages
    - Ex:
        - 15, 16, 18, 19, 22, 24, 29, 30, 34
        - We have 9 values, so 25% of the data would be approximately 2 data points. So the 3rd datapoint is greater than 25% of the data. Thus, the 25th percentile is 18 (the 3rd datapoint)
        - Similarly, 75% of the data is approximately 6 data points. So the 7th datapoint is greater than 75% of the data. Thus, the 75th percentile is 29 (the 7th datapoint)
        - The full range of our data is between 15 and 34. The 25th and 75th percentiles tell us that half our data is between 18 and 29. This helps us gain understanding of how the data is distributed
    - Standard deviation and variance are measures of how dispersed or spread out the data is
    - Process for variance and std:
        - To get the variance you use this formula:
            - Take the mean and subtract that from all numbers
            - After that step you square all those numbers and add them together
            - Take the new total and divide it by the total number of values to get the variances
            - To get the std, you just square root the variance
    - Normal distribution graph:
        - 

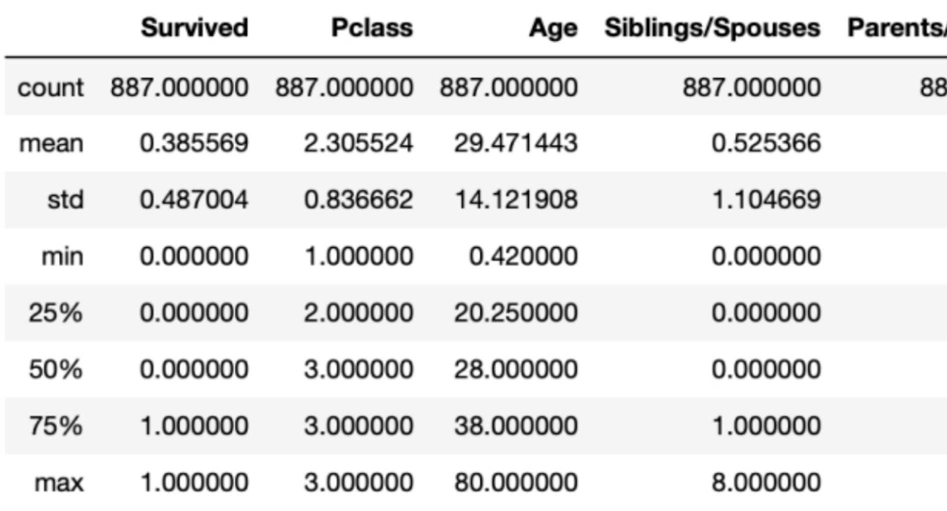
    - Statistics with Python:

        - Use the numpy package in Python to do various calculations for mean, median, percentile, std, and variance
        - Numpy is used for fast and easy math operations to be used on arrays
        - To get this package do this:
            - Import numpy as np
        - Ex of how numpy is used:
            - import numpy as np
            - data = [15, 16, 18, 19, 22, 24, 29, 30, 34]
            - print("mean:", np.mean(data))
            - print("median:", np.median(data))
            - print("50th percentile (median):", np.percentile(data, 50))
            - print("25th percentile:", np.percentile(data, 25))
            - print("75th percentile:", np.percentile(data, 75))
            - print("standard deviation:", np.std(data))
            - print("variance:", np.var(data))
- What is Pandas:
    - Pandas is a Python module that reads and manipulates data
    - You can take data and put it into readable table
    - Table of data -> DataFrame -> Pandas data object
    - Read in your Data:
        - Import pandas -> import pandas as pd
        - Using the dataset from the Titanic passengers and this data is stored in a CSV
        - Read_csv takes csv formats and converts it to Pandas DataFrame:
            - Ex:
                - df = pd.read_csv('titanic.csv")
                - Df is not a Pandas DataFrame right now
        - The head method will return the first 5 rows in a DataFrame:
            - Ex:
                - print(df.head())
        - Example code of using read_csv() and head():
            - import pandas as pd
            - df = pd.read_csv('path of titanic.csv file')
            - It will print out something like this:



	Survived	Pclass	Sex	Age	Siblings/Spouses	Parents/
0	0	3	male	22.0	1	
1	1	1	female	38.0	1	
2	1	3	female	26.0	0	
3	1	1	female	35.0	1	
4	0	3	male	35.0	0	

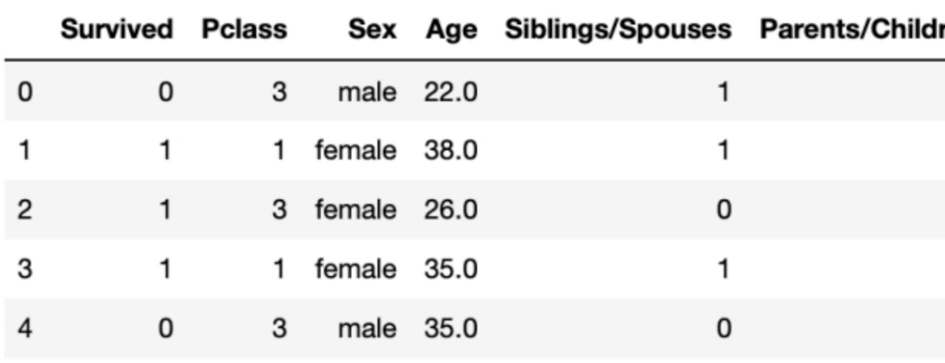
- Summarize the Data:
  - The describe method will return a table of statistics about the columns
  - print(df.describe())
  - Example code of how describe will print out:
    - import pandas as pd
    - pd.options.display.max_columns = 6
    - df = pd.read_csv(‘path of titanic.csv file’)
    - print(df.describe())
    - Result will look something like this:



	Survived	Pclass	Age	Siblings/Spouses	Parents/
count	887.000000	887.000000	887.000000	887.000000	88
mean	0.385569	2.305524	29.471443	0.525366	
std	0.487004	0.836662	14.121908	1.104669	
min	0.000000	1.000000	0.420000	0.000000	
25%	0.000000	2.000000	20.250000	0.000000	
50%	0.000000	3.000000	28.000000	0.000000	
75%	1.000000	3.000000	38.000000	1.000000	
max	1.000000	3.000000	80.000000	8.000000	

- Manipulating Data with Pandas:
  - Selecting a Single Column:
    - Use square brackets and column name to get a single column
      - Ex:
        - col = df[‘Fare’]
        - print(col)
    - The result is a Pandas Series (a 1d array that can hold any data type)
    - Ex of getting Single Column:

        - import pandas as pd
        - df = pd.read_csv('https://sololearn.com/uploads/files/titanic.csv')
        - col = df['Fare']
        - print(col)
- Selecting Multiple Columns:
    - Take original DataFrame and create a smaller DataFrame
    - The example will focus on age, sex, survived
    - Use double [[ ]] for mutiple columns
    - Ex:
        - import pandas as pd
        - df = pd.read_csv('https://sololearn.com/uploads/files/titanic.csv')
        - small_df = df[['Age', 'Sex', 'Survived']]
        - print(small_df.head())
- Creating a Column:
    - This exmple will have data about the sex of the passengaer as boolean value
    - Ex:
        - Create a Pandas Series that will be Trues and Falases
        - df['Sex'] = = 'male'
        - Create a column with this result by doing this
        - Df['male'] = df['Sex'] == 'male'
        - Whole thing:
        - import pandas as pd
        - df = pd.read_csv('https://sololearn.com/uploads/files/titanic.csv')
        - df['male'] = df['Sex'] == 'male'
        - print(df.head())
        - Result:



	Survived	Pclass	Sex	Age	Siblings/Spouses	Parents/Childr
0	0	3	male	22.0	1	
1	1	1	female	38.0	1	
2	1	3	female	26.0	0	
3	1	1	female	35.0	1	
4	0	3	male	35.0	0	

            -
- Numpy Basics:
    - Numpy arrays are mainly used for convert DataFrame into a format that easier for computations
- Converting from Pandas Series to a Numpy Array:
    - First you need a Pandas Series, it will look something like this:
        - df['Fare']
    - Convert Pandas Series -> Numpy Array:

                - df[‘Fare’].values
            - Example code:
                - import pandas as pd
                - df = pd.read_csv(“Path for titanic.csv”)
                - print(df[‘Fare’].values
            - For converting 2d DataFrame to 2d Numpy Array:
                - It the same thing as converting pandas series to numpy array, but you have multiple title and have two square brackets
            - Ex:
                - import pandas as pd
                - df = pd.read_csv('https://sololearn.com/uploads/files/titanic.csv')
                - print(df[['Pclass', 'Fare', 'Age']].values)
        - Numpy Shape Attribute:
            - Numpy shape determines the size of a numpy array (.shape)
            - Ex:
                - import pandas as pd
                - df = pd.read_csv('https://sololearn.com/uploads/files/titanic.csv')
                - arr = df[['Pclass', 'Fare', 'Age']].values
                - print(arr.shape)
            - Can also use shape on pandas DataFrame and use it to find number of rows and number columns present in a dataset
- More with Numpy Array:
    - Select from a Numpy Array:
        - Assume we have this arr = df[[‘Pclass’, ‘Fare’, ‘Age’]].values
        - To get a single element from a numpy array with this:
            - Arr[0,1] -> first item is the row # and the second is the column #
            - For this case it prints out the second column of the first row
        - Can also just select a single row by doing this:
            - print(arr[0]) -> print the whole row of first passenger
        - Also you can pick a single column by doing something like this:
            - print(arr[:,2]) -> this will print Age Column content
        - Ex:
            - import pandas as pd
            - df = pd.read_csv('https://sololearn.com/uploads/files/titanic.csv')
            - arr = df[['Pclass', 'Fare', 'Age']].values
            - print(arr[0, 1])
            - print(arr[0])
            - print(arr[:,2])
    - Mask:
        - Boolean array that tells us which values from the array we are need
        - Ex:
            - import pandas as pd
            -

            - df = pd.read_csv('https://sololearn.com/uploads/files/titanic.csv')
            - # take first 10 values for simplicity
            - arr = df[['Pclass', 'Fare', 'Age']].values[:10]
            -
            - mask = arr[:, 2] < 18
            - # this part of the code set up condition to make sure that children are only and there boolean values are stored in mask
            - print(arr[mask])
            - print(arr[arr[:, 2] < 18])
            - #this is alternative way of using mask by just throwing the logic into the arr
    - Summing and Counting:
        - Ex:
            - import pandas as pd
            -
            - df = pd.read_csv('https://sololearn.com/uploads/files/titanic.csv')
            - arr = df[['Pclass', 'Fare', 'Age']].values
            - mask = arr[:, 2] < 18
            -
            - print(mask.sum())
            - print((arr[:, 2] < 18).sum())
        - In this example the mask.sum() sums up the array and that the same thing as counting the number of true values present in the set
    - Plotting Basics:
        - Matplotlib is used for plot data and to get it you need to do this:
            - import matplotlib.pyploy as plt
        - To plot our data, its recommended to do this:
            - plt.scatter(df['Age'], df['Fare'])
            - The first argument is x axis and second argument is y axis
        - Label:
            - plt.xlabel('Age')
            - plt.ylabel('Fare')
        - Color Coding:
            - plt.scatter(df['Age'], df['Fare'], c=df['Pclass'])
            - The c parameter will give it a pandas series
            - The code will allow for every single one of those categories to have different colors
        - A scatter plot is used to show all the values from your data on a graph. In order to get a visual representation of our data, we have to limit our data to two features.
    - Line:
        - Plot function can draw lines for the data points present
        - Ex:

        - plt.plot([0,80],[85,5])
        - Two values present are coordinates values (x values first and then y values)

<u>Classification</u>:

- What is Classification:
    - ML is made up of supervised and unsupervised learning
    - Supervised Learning means that we will have labeled historical data that we will use to inform our model. We call the label or thing we're trying to predict, the target. So in supervised learning, there is a known target for the historical data, and for unsupervised learning there is no known target.
    - Within supervised learning, there is Classification and Regression. Classification problems are where the target is a categorical value (often True or False, but can be multiple categories). Regression problems are where the target is a numerical value.
    - For example, predicting housing prices is a regression problem. It's supervised, since we have historical data of the sales of houses in the past. It's regression, because the housing price is a numerical value. Predicting if someone will default on their loan is a classification problem. Again, it's supervised, since we have the historical data of whether past loanees defaulted, and it's a classification problem because we are trying to predict if the loan is in one of two categories (default or not).
    - Logistic Regression, while it has regression in its name is an algorithm for solving classification problems, not regression problems.
- Classification Terminology:
    - Target is item that you are trying to predicting using machine learning
    - Feature/predictor is information about said class that we are trying to predict
    - A ML model will be used to help us predict the targets and features
- A Linear Model for Classification:
    - Equation for a line is 0 = ax+by+c
    - The coefficients of the line control where the line goes
    - The x and y correspond to to x and y axis points
    - Using a equation for a line based on two points can help us make prediction of a data point would fit the trend of said line
    - In the Titanic example, depending on what side of the line a point lied it might be the difference between survive or death
    - Logistic Regression is the best method for mathematically finding the best line possible
- Logistic Regression Model:
    - Each datapoint in a model will have a value from 0 to 1
    - This is the probability of that event occurring
    - In the titanic example, it the probability of the person surviving
    - Use the sigmoid function give accurate prediction/probability that event occurred

- Formula:
  - $$\frac{1}{1+e^{-(ax+by+c)}}$$
- To determine if the line that is created is good, there is a need for predictions are correct
- To reward good prediction and penalize bad prediction is possible by the likelihood equation
- Equation:
  - $$\text{likelihood} = \begin{cases} p & \text{if passenger survived} \\ 1-p & \text{if passenger didn't survive} \end{cases}$$
- Let's look at a couple possibilities:
  - If the predicted probability p is 0.25 and the passenger didn't survive, we get a score of 0.75 (good).
  - If the predicted probability p is 0.25 and the passenger survived, we get a score of 0.25 (bad).
  - We multiply all the individual scores for each datapoint together to get a score for our line. Thus we can compare different lines to determine the best one.
- Get the total score by multiplying the four score together
- The value is always going to be really small since it is the likelihood that our model predicts everything perfectly. A perfect model would have a predicted probability of 1 for all positive cases and 0 for all negative cases.
- The likelihood is how we score and compare possible choices of a best fit line.

- Building a Logistic Regression Model with Sklearn:
  - Scikit-learn is a Python module that use many machine learning algorithms
  - Its one of the best documented Python module out right now and there are tons of sample code on scikit-learn.org
  - With the Titanic example:
    - First want to creat boolean column for Sex:
      - df['male'] = df['Sex'] == 'male'
      -
    - Store a numpy array with all the features and take only the columns were are interested in and convert it to numpy array:
      - X = df[['Pclass', 'male', 'Age', 'Siblings/Spouses', 'Parents/Children', 'Fare']].values
    - Take target survuved column and store it in variable y:

        - y = df['Survived'].values
    - Example code:
        - import pandas as pd
        - df = pd.read_csv('https://sololearn.com/uploads/files/titanic.csv')
        - df['male'] = df['Sex'] == 'male'
        - X = df[['Pclass', 'male', 'Age', 'Siblings/Spouses', 'Parents/Children', 'Fare']].values
        - y = df['Survived'].values
        - print(X)
        - print(y)
- Standard practice for x to be 2d array while y is 1d array
- Importing the Logistic Regression model:
    - from sklearn.linear_model import LogisticRegression
- Sklearn model are Python class:
    - model = LogisticRegression()
- Taking example columns of fare, age, and survived we need to covert those Dataframe and series into numpy array:
    - X = df[['Fare', 'Age']].values
    - y = df['Survived'].values
- Fit method is used to build a model (first argument is 2d numpy array and the y is 1d numpy array)
    - model.fit(X,y)
- Choosing a line of best fit (look at the coefficients):
    - print(model.coef_,model.intercept_)
- It recommended to have as many features so for the ititnatic example throw all the features in:
    - X = df[['Pclass', 'male', 'Age', 'Siblings/Spouses', 'Parents/Children', 'Fare']].values
    - y = df['Survived'].values
    - model = LogisticRegression()
    - model.fit(X, y)
- Predict method make prediction when given 2d array:
    - model.predict(X)
    - The first person is [3,True,22.0,1,0,7.25] so if we do the following:
        - print(model.predict([[3, True, 22.0, 1, 0, 7.25]]))
        - This will give us the probability that individual lived based on the previous data
        - It produced # [0] so this means that they died
- Model predicting first 5 rows of data:
    - print(model.predict(X[:5]))
    - # [0 1 1 1 0]
    - print(y[:5])
    - # [0 1 1 1 0]

    - To check the accuracy of prediction compare it with the real data (y)
- Whole Code Example:
    - import pandas as pd
    - from sklearn.linear_model import LogisticRegression
    - df = pd.read_csv('https://sololearn.com/uploads/files/titanic.csv')
    - df['male'] = df['Sex'] == 'male'
    - X = df[['Pclass', 'male', 'Age', 'Siblings/Spouses', 'Parents/Children', 'Fare']].values
    - y = df['Survived'].values
    - model = LogisticRegression()
    - model.fit(X, y)
    - print(model.predict([[3, True, 22.0, 1, 0, 7.25]]))
    - print(model.predict(X[:5]))
    - print(y[:5])
- Score the Model:
    - Counting the number of datapoints it predicts correctly is called accuracy score
    - Creating an array that has the predicted y values:
        - y_pred = model.predict(y)
    - Create array of boolean values of whether or not our model predicted each passenger correctly
        - y == y_pred
    - Use numpy sum method to get the amount of points that were accurate:
        - print((y == y_pred).sum())
        - # 714
    - 714/887 were accuracy and that's around 80.4%
    - To get the percentage of this we need total passenger and this is done by the shape() function
        - y.shape[0]
    - Accuracy score:
        - print((y == y_pred).sum() / y.shape[0])
        - # 0.8038331454340474
    - Easier way of doing this is by using:
        - print(model.score(X, y))
        - # 0.8038331454340474
    - Whole Code:
        - import pandas as pd
        - from sklearn.linear_model import LogisticRegression
        -
        - df = pd.read_csv('https://sololearn.com/uploads/files/titanic.csv')
        - df['male'] = df['Sex'] == 'male'
        - X = df[['Pclass', 'male', 'Age', 'Siblings/Spouses', 'Parents/Children', 'Fare']].values

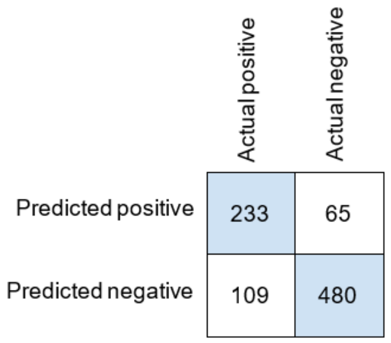
        - y = df['Survived'].values
        -
        - model = LogisticRegression()
        - model.fit(X, y)
        -
        - y_pred = model.predict(X)
        - print((y == y_pred).sum())
        - print((y == y_pred).sum() / y.shape[0])
        - print(model.score(X, y))
- Introducing the Breast Cancer Dataset (Example of whole process):
    - Loading the dataset and taking peak at the data and how it formatted:
        - from sklearn.datasets import load_breast_cancer
        - cancer_data = load_breast_cancer()
    - The cancer_data object is similar Python dictionary
    - We can see the avviabile keys with key methods:
        - print(cancer_data.keys())
    - Looking at DESCR, this give detailed description of the dataset:
        - print(cancer_data['DESCR'])
        -
    - Whole code:
        - import pandas as pd
        - from sklearn.datasets import load_breast_cancer
        -
        - cancer_data = load_breast_cancer()
        - print(cancer_data.keys())
        - print(cancer_data['DESCR'])
    - 30 features, 569 datapoints and target is either malignant or benign
    - 10 measurements with multiple values for mean, std, and worst value
    - Feature engineering is the processing of figuring out what additional feature to calculate
    - We need to first get the gesture data and store them
    - Numpy array of the data:
        - Cancer_data['data']
    - Use shape to see that an array of 569 and 30 columns is present:
        - cancer_data['data'].shape
    - Put this into a Pandas DataFrame with all our features data:
        - cancer_data['feature_names']
        - df = pd.DataFrame(cancer_data['data'], columns=cancer_data['feature_names'])
        - print(df.head()) #only displays the first 5 datapoints
    - We need the target in a 1 d numpy array of 1's and 0's:
        - cancer_data['target']
    - Shape of target array:

        - cancer_data[‘target’].shape
    - Need to know if 1 or 0 is benign or malignant:
        - cancer_data['target_names']
    - This give the array [‘malignant’ ‘bnign’] which tells us that 0 means malignant and 1 means benign
        - df['target'] = cancer_data['target']
        - df.head()
    - Whole Code for this part:
        - import pandas as pd
        - from sklearn.datasets import load_breast_cancer
        -
        - cancer_data = load_breast_cancer()
        -
        - df = pd.DataFrame(cancer_data['data'], columns=cancer_data['feature_names'])
        - df['target'] = cancer_data['target']
        - print(df.head())
    - Build our feature matrix X and target array y using Logistic Regression model:
        - X = df[cancer_data.feature_name].values
        - y = df[‘target’].values
    - Create a Logistic Regression object and fit method to build model:
        - model = LogisticRegression()
        - model.fit(X, y)
    - Convergence Warning may appear and this means model needs more time to find the optimal solution
    - One option is increase number of iterations or switch different solver
    - The solver is the algorithm that model uses to find the equation of the line
    - Look at this for varies solvers (https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html)
    - Change the solver algorithm:
        - model = LogisticRegression(solver='liblinear')
        - model.fit(X, y)
    - Model predicts the first datapoint in our dataset and predict method takes 2d array so datapoint need to be put in:
        - model.predict([X[0]])
    - Performance of the whole dataset:
        - model.score(X, y)
    - The code for the whole project:
        - import pandas as pd
        - from sklearn.datasets import load_breast_cancer

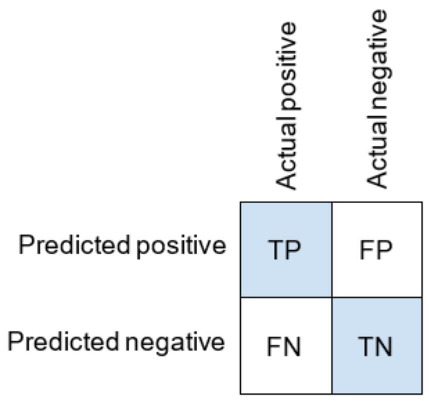
            - from sklearn.linear_model import LogisticRegression
            -
            - cancer_data = load_breast_cancer()
            - df = pd.DataFrame(cancer_data['data'], columns=cancer_data['feature_names'])
            - df['target'] = cancer_data['target']
            -
            - X = df[cancer_data.feature_names].values
            - y = df['target'].values
            -
            - model = LogisticRegression(solver='liblinear')
            - model.fit(X, y)
            - print("prediction for datapoint 0:", model.predict([X[0]]))
            - print(model.score(X, y))
        - The tools present can build a model for any classification dataset

Model Evaluation:

- Accuracy:
    - Percent of prediction that are correct
    - It is not always the best way to measure since it only good when classes are evenly split, but it is overall misleading if we have not even split classes
- Confusion Matrix:
    - Can see the important values in dataset
    - The confusion matrix fully describes how a model performs on a dataset, though is difficult to use to compare models.
    - Shows four values:
        - Datapoints we predicted positive that are actually positive
        - Datapoints we predicted positive that are actually negative
        - Datapoints we predicted negative that are actually positive
        - Datapoints we predicted negative that are actually negative
    - Ex:
        - In our Titanic dataset, we have 887 passengers, 342 survived (positive) and 545 didn't survive (negative). The model we built in the previous module has the following confusion matrix.
        -



	Actual positive	Actual negative
Predicted positive	233	65
Predicted negative	109	480

        - The blue shaded squares are the counts of the predictions that we got correct. So of the 342 passengers that survived, we predicted 233 or them correctly (and 109 of them incorrectly). Of the 545 passengers that didn't survive, we predicted 480 correctly (and 65 incorrectly).
        - We can use the confusion matrix to compute the accuracy. As a reminder, the accuracy is the number of datapoints predicted correctly divided by the total number of datapoints.
        - (233+480)/(233+65+109+480) = 713/887 = 80.38%
    - A true positive (TP) is a datapoint we predicted positively that we were correct about.
    - A true negative (TN) is a datapoint we predicted negatively that we were correct about.
    - A false positive (FP) is a datapoint we predicted positively that we were incorrect about.
    - A false negative (FN) is a datapoint we predicted negatively that we were incorrect about.
    - The terms can be a little hard to keep track of. The way to remember is that the second word is what our prediction is (positive or negative) and the first word is whether that prediction was correct (true or false).
    - Image of what TP,TN,FP, and FN look like:
        - 

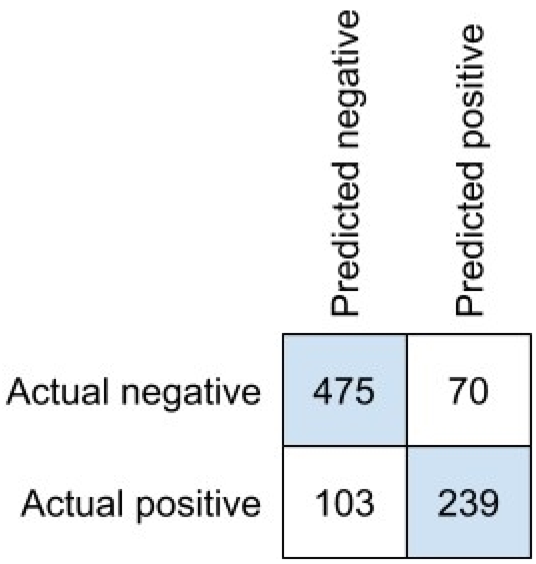
	Actual positive	Actual negative
Predicted positive	TP	FP
Predicted negative	FN	TN

- Precision and Recall:
    - Precision refers to the percentage of positive results which are relevant while recall is the percentage of positive cases correctly classified
    - Formula for precision:
        - $$\text{precision} = \frac{\#\text{ positives predicted correctly}}{\#\text{ positive predictions}} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$
    - Formula for recall:
        - $$\text{recall} = \frac{\#\text{ positives predicted correctly}}{\#\text{ positive cases}} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

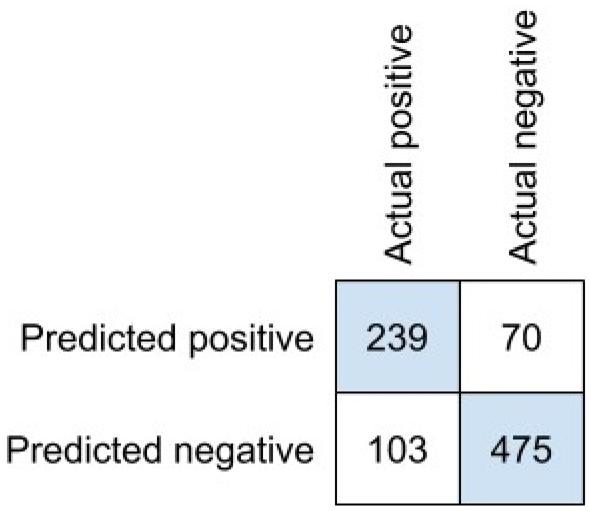
- Trade offs:
    - The higher the false positive, the lower the precision
    - For example, let’s say we’re building a model to predict if a credit card charge is fraudulent. The positive cases for our model are fraudulent charges and the negative cases are legitimate charges.
    - Let’s consider two scenarios:
    - 1. If we predict the charge is fraudulent, we’ll reject the charge.
    - 2. If we predict the charge is fraudulent, we’ll call the customer to confirm the charge.
    - In case 1, it’s a huge inconvenience for the customer when the model predicts fraud incorrectly (a false positive). In case 2, a false positive is a minor inconvenience for the customer.
    - The higher the false positives, the lower the precision. Because of the high cost to false positives in the first case, it would be worth having a low recall in order to have a very high precision. In case 2, you would want more of a balance between precision and recall.
    - There’s no hard and fast rule on what values of precision and recall you’re shooting for. It always depends on the dataset and the application.
- F1 Score:
    - Is an average of the precision and recall so that we have a single score for our model
    - Formula:

$$F1 = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

- Calculating Metrics in Scikit-learn:
    - Has function for accuracy, precision, recall, and F1 score
    - Example of the previous code with new Scikit-learn applications:
        - from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
        - df = pd.read_csv('https://sololearn.com/uploads/files/titanic.csv')
        - df['male'] = df['Sex'] == 'male'
        - X = df[['Pclass', 'male', 'Age', 'Siblings/Spouses', 'Parents/Children', 'Fare']].values
        - y = df['Survived'].values
        - model = LogisticRegression()
        - model.fit(X, y)
        - y_pred = model.predict(X)
        - print("accuracy:", accuracy_score(y, y_pred))
        - print("precision:", precision_score(y, y_pred))

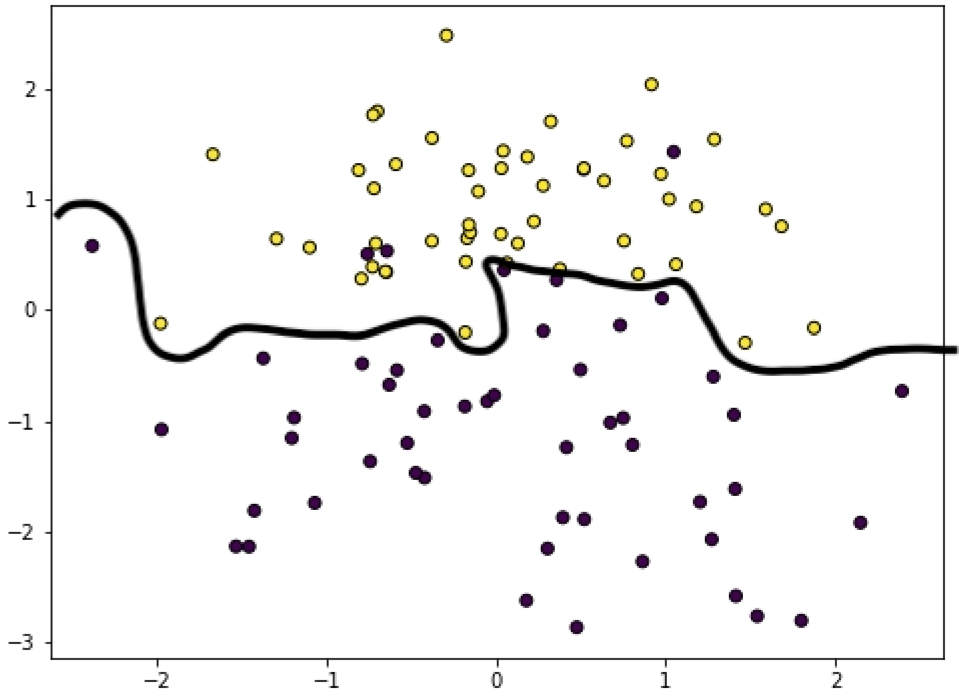
        - print("recall:", recall_score(y, y_pred))
        - print("f1 score:", f1_score(y, y_pred))
    - These functions take two 1-d numpy arrays: the true values of the target and the predicted values of the start
    - Confusion matrix function are used to get four values such as true positives, false positives, false negatives, and true negatirvs
    - Ex:
        - from sklearn.metrics import confusion_matrix
        - print(confusion_matrix(y, y_pred))
        - # Output:
        - # [[475 70]
        - # [103 239]]
    - Scikit-learn reverses the confusion matrix to show the negative conunts first
    - This is how Scikit-learn does it such be labeled:
        - 

	Predicted negative	Predicted positive
Actual negative	475	70
Actual positive	103	239

    - This is how confusion matrix is normally done:
        - 

	Actual positive	Actual negative
Predicted positive	239	70
Predicted negative	103	475

        - Since negative target values correspond to 0 and positive to 1, scikit-learn has ordered them in this order. Make sure you double check that you are interpreting the values correctly!
- Training and Testing:
    - Overfitting:
        - This occurs when we perform well on the data the model has already seen, but we don't perform well on the new data
        - Visual of overfitting:

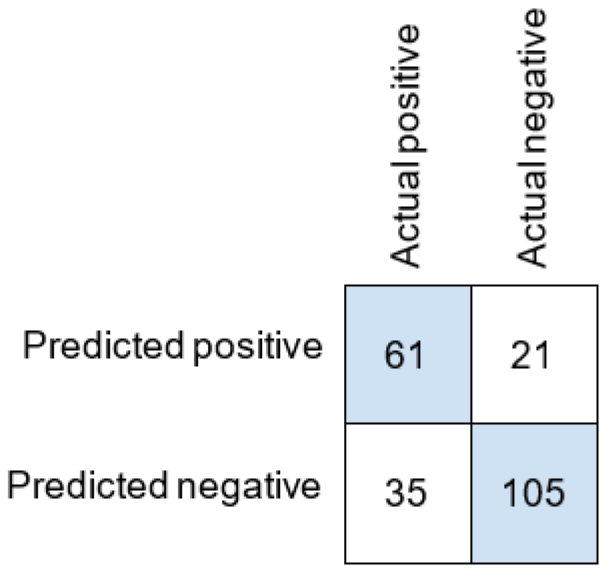


-
        - The line is too close to trying to get ever single datapoint on the correct side of the line, but it is missing the essence of the data
        - More features in a dataset, the more prone it is to overfitting
    - Training Set and Test Set:
        - To help prevent overfitting and other problems, we have to split our data into a training and test set
        - The training set is used for building the model while the test set is used for evaluating the models
        - The standard breakdown is to put 70-80% of the data for training set and 20-30% for the test set
    - Training and Testing in Sklearn:
        - Assuming that we have 2d numpy array for X (features) and 1d numpy array of the target use the train_test_split function to set it up

    - By default the training set is 75% of the data and the test set is the remaining 25% of the data.
    - Example of using train_test_split operation:
        - from sklearn.model_selection import train_test_split
        - X_train, X_test, y_train, y_test = train_test_split(X, y)
        - #using the shape attribute to see the sizes of our datasets
        - print("whole dataset:", X.shape, y.shape)
        - print("training set:", X_train.shape, y_train.shape)
        - print("test set:", X_test.shape, y_test.shape)
    - Example code of the whole train_tesst_split operation for whole dataset:
        - import pandas as pd
        - from sklearn.linear_model import LogisticRegression
        - from sklearn.model_selection import train_test_split
        -
        - df = pd.read_csv('https://sololearn.com/uploads/files/titanic.csv')
        - df['male'] = df['Sex'] == 'male'
        - X = df[['Pclass', 'male', 'Age', 'Siblings/Spouses', 'Parents/Children', 'Fare']].values
        - y = df['Survived'].values
        -
        - X_train, X_test, y_train, y_test = train_test_split(X, y)
        -
        - print("whole dataset:", X.shape, y.shape)
        - print("training set:", X_train.shape, y_train.shape)
        - print("test set:", X_test.shape, y_test.shape)
- We can change the size of our training set by using the train_size parameter. E.g. train_test_split(X, y, train_size=0.6) would put 60% of the data in the training set and 40% in the test set.
- Building a Scikit-learn Model Using a Training Set:
    - How to build a model using the training set:
        - model = LogisticRegression()
        - model.fit(X_train, y_train)
        - #evaluate the model using the test set
        - print(model.score(X_test, y_test))
    - All the metrics from before should only be calculated oin the test set:
        - y_pred = model.predict(X_test)
        - print("accuracy:", accuracy_score(y_test, y_pred))
        - print("precision:", precision_score(y_test, y_pred))
        - print("recall:", recall_score(y_test, y_pred))
        - print("f1 score:", f1_score(y_test, y_pred))
    - Note that the split is random each time so the scores for accuracy, precision, recall and f1 are going to be different
- Using a Random State:

            - Without random states the values above will change all the time
            - Example of changing values for accuracy, precision, recall, and f1:
                - from sklearn.model_selection import train_test_split
                -
                - X = [[1, 1], [2, 2], [3, 3], [4, 4]]
                - y = [0, 0, 1, 1]
                -
                - X_train, X_test, y_train, y_test = train_test_split(X, y)
                - print('X_train', X_train)
                - print('X_test', X_test)
            - To get the same split every time, we can use the random_state attribute and give it any number that we want:
            - Example code of Random State:
                - from sklearn.model_selection import train_test_split
                -
                - X = [[1, 1], [2, 2], [3, 3], [4, 4]]
                - y = [0, 0, 1, 1]
                -
                - X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=27)
                - print('X_train', X_train)
                - print('X_test', X_test)
            - Another word for random state is seed
- Foundation for the ROC Curve:
    - Logistics Regression Threshold:
        - Making the threshold higher would cause fewer positive predictions to occur and more likely to be correct
        - Precision would be higher and recall lower and vice versa
        - An ROC (Receiver operating characteristics) curve is graphs all the possible models and their performance
    - Sensitivity and Specificity:
        - Sensitivity is another term for recall which is the true positive rate
        - Formula for sensitivity:
            - $$\text{sensitivity} = \text{recall} = \frac{\#\text{ positives predicted correctly}}{\#\text{ positive cases}} = \frac{\text{TP}}{\text{TP}+\text{FN}}$$
        - Specificity is the true negative rate
        - Formula for specificity:
            - $$\text{specificity} = \frac{\#\text{ negatives predicted correctly}}{\#\text{ negative cases}} = \frac{\text{TN}}{\text{TN}+\text{FP}}$$
        - Example of this:

        - We’ve done a train test split on our Titanic dataset and gotten the following confusion matrix. We have 96 positive cases and 126 negative cases in our test set.

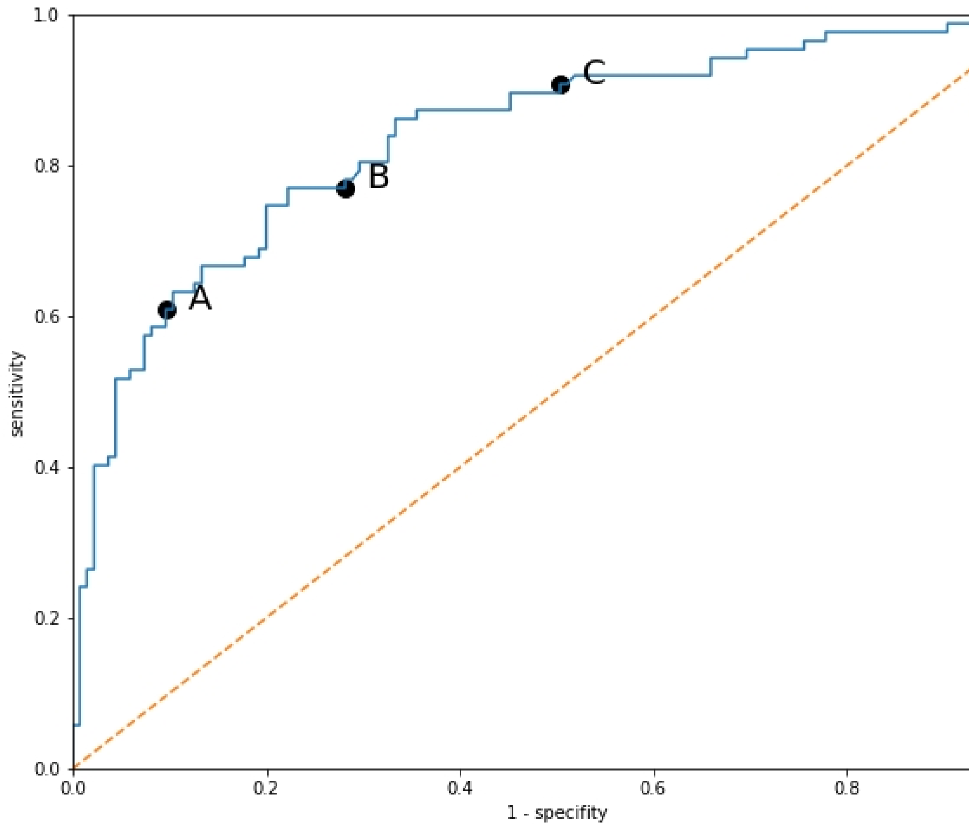


	Actual positive	Actual negative
Predicted positive	61	21
Predicted negative	35	105

        -
        - Sensitivity = 61/96 = 0.6354
        - Specificity = 105/126 = 0.8333
    - We want to make sure that these two values are as higher as possible, but increase one cause the other to decrease
    - A nice balance has to be found over else it will not work
    - While we generally look at precision and recall values, for graphing the standard is to use the sensitivity and specificity. It is possible to build a precision-recall curve, but this isn’t commonly done.
- Sensitivity and Specificity in Scikit-learn
    - Scikit learn does not have functions for sensitivity and specificity, but you can figure it out:
    - Example of figuring out the sensitivity:
        - from sklearn.metrics import recall_score
        - sensitivity_score = recall_score
        - print(sensitivity_score(y_test, y_pred))
        - # 0.6829268292682927
    - We need to import the precision and recall fscore from scikit:
        - from sklearn.metrics import precision_recall_fscore_support
        - print(precision_recall_fscore_support(y, y_pred))
    - Writing a function to get the specificity:
        - def specificity_score(y_true, y_pred):
        -   p, r, f, s = precision_recall_fscore_support(y_true, y_pred)
        -   return r[0]
        - print(specificity_score(y_test, y_pred))

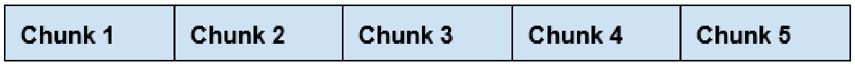
      - # 0.9214285714285714
    - Whole code to run a random state in a train test split in which you get the sensitivity and specificity:
      - import pandas as pd
      - from sklearn.linear_model import LogisticRegression
      - from sklearn.model_selection import train_test_split
      - from sklearn.metrics import recall_score, precision_recall_fscore_support
      -
      - sensitivity_score = recall_score
      - def specificity_score(y_true, y_pred):
      -   p, r, f, s = precision_recall_fscore_support(y_true, y_pred)
      -   return r[0]
      -
      - df = pd.read_csv('https://sololearn.com/uploads/files/titanic.csv')
      - df['male'] = df['Sex'] == 'male'
      - X = df[['Pclass', 'male', 'Age', 'Siblings/Spouses', 'Parents/Children', 'Fare']].values
      - y = df['Survived'].values
      -
      - X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=5)
      -
      - model = LogisticRegression()
      - model.fit(X_train, y_train)
      - y_pred = model.predict(X_test)
      -
      - print("sensitivity:", sensitivity_score(y_test, y_pred))
      - print("specificity:", specificity_score(y_test, y_pred))
  - Adjusting the Logistics Regression Threshold in Sklearn:
    - The predict_prob will tell you value from 0-1 of a datapoint
    - Ex: (model.predict_proba(X_test)
    - The result is a numpy array with two values:
      - Odds of datapoint being in 0
      - Odds of datapoint being in 1
    - If you only need the second column you can use this code:
      - model.predict_proba(X_test)[:, 1]
    - Setting up a threshold of 0.75 and comparing it with all of our data points:
      - y_pred = model.predict_proba(X_test)[:, 1] > 0.75
    - The following code will do look at all the datapoints and make sure anything with 0.75 will pass:
      - import pandas as pd

                - from sklearn.linear_model import LogisticRegression
                - from sklearn.metrics import precision_score, recall_score
                - from sklearn.model_selection import train_test_split
                - df = pd.read_csv('https://sololearn.com/uploads/files/titanic.csv')
                - df['male'] = df['Sex'] == 'male'
                - X = df[['Pclass', 'male', 'Age', 'Siblings/Spouses', 'Parents/Children', 'Fare']].values
                - y = df['Survived'].values
                - X_train, X_test, y_train, y_test = train_test_split(X, y)
                - model = LogisticRegression()
                - model.fit(X_train, y_train)
                - print("predict proba:")
                - print(model.predict_proba(X_test))
                - y_pred = model.predict_proba(X_test)[:, 1] > 0.75
                - print("precision:", precision_score(y_test, y_pred))
                - print("recall:", recall_score(y_test, y_pred))
        - 0.5 is the originally Logistics Regression model and anything lower or higher is a different model that is produced
- ROC Curve:
    - How to Build an ROC Curve:
        - Graphs specificity and sensitivity
        - Plotting a ROC curve using matplotlib:
            - model = LogisticRegression()
            - model.fit(X_train, y_train)
            - y_pred_proba = model.predict_proba(X_test)
            - fpr, tpr, thresholds = roc_curve(y_test, y_pred_proba[:,1])
            -
            - plt.plot(fpr, tpr)
            - plt.plot([0, 1], [0, 1], linestyle='--')
            - plt.xlim([0.0, 1.0])
            - plt.ylim([0.0, 1.0])
            - plt.xlabel('1 - specificity')
            - plt.ylabel('sensitivity')
            - plt.show()
        - This only assuming that the dataset has been split into training and test set
    - ROC Curve Interpretation:
        - ROC Curve shows the performance of many model rather than one model
        - Example of this process:

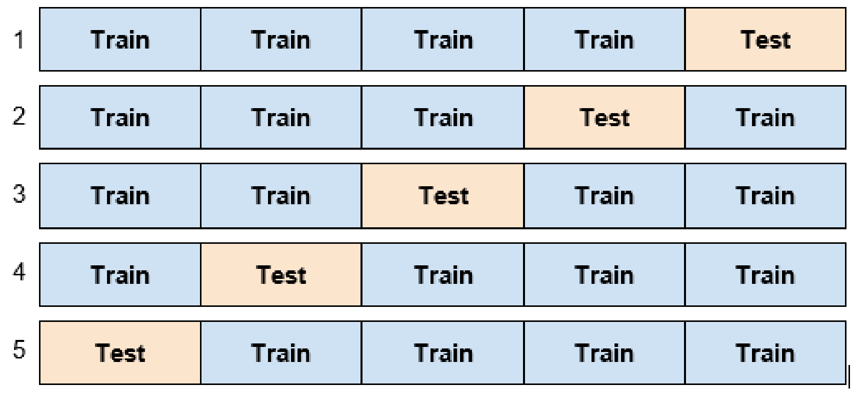


-
    - Each point A, B & C refers to a model with a different threshold
    - Model A has a sensitivity of 0.6 and a specificity of 0.9 (recall that the graph is showing 1-specificity).
    - Model B has a sensitivity of 0.8 and a specificity of 0.7.
    - Model C has a sensitivity of 0.9 and a specificity of 0.5.
    -
    - How to choose between these models will depend on the specifics of our situation.
    - The closer the curve gets to the upper left corner, the better the performance. The line should never fall below the diagonal line as that would mean it performs worse than a random model.
- Picking a Model from the ROC Curve:
    - If we are in a situation where it's more important that all of our positive predictions are correct than that we catch all the positive cases (meaning that we predict most of the negative cases correctly), we should choose the model with higher specificity (model A).
    -

        - If we are in a situation where it's important that we catch as many of the positive cases as possible, we should choose the model with the higher sensitivity (model C).
        -
        - If we want a balance between sensitivity and specificity, we should choose model B.
        - It can be tricky keeping track of all these terms. Even experts have to look them up again to ensure they are interpreting the values correctly.
    - Area Under the Curve:
        - AUC gives us sense of how well the LogisticRegression Model performs
        - AUC calculation using scikit-learn:
            - (roc_auc_score(y_test, y_pred_proba[:,1])
        - Example of using roc_auc_score for the Titanic Dataset:
            - import pandas as pd
            - from sklearn.linear_model import LogisticRegression
            - from sklearn.model_selection import train_test_split
            - from sklearn.metrics import roc_auc_score
            -
            - df = pd.read_csv('https://sololearn.com/uploads/files/titanic.csv')
            - df['male'] = df['Sex'] == 'male'
            - X = df[['Pclass', 'male', 'Age', 'Siblings/Spouses', 'Parents/Children', 'Fare']].values
            - y = df['Survived'].values
            -
            - X_train, X_test, y_train, y_test = train_test_split(X, y)
            -
            - model1 = LogisticRegression()
            - model1.fit(X_train, y_train)
            - y_pred_proba1 = model1.predict_proba(X_test)
            - print("model 1 AUC score:", roc_auc_score(y_test, y_pred_proba1[:, 1]))
            -
            - model2 = LogisticRegression()
            - model2.fit(X_train[:, 0:2], y_train)
            - y_pred_proba2 = model2.predict_proba(X_test[:, 0:2])
            - print("model 1 AUC score:", roc_auc_score(y_test, y_pred_proba2[:, 1]))
        - It's important to note that this metric tells us how well in general a Logistic Regression model performs on our data. As an ROC curve shows the performance of multiple models, the AUC is not measuring the performance of a single model.
- K-fold Cross Validation:
    - Concerns with Training and Test Set:

        - Instead of doing a single train/test split, we’ll split our data into a training set and test set multiple times.
        - As we can see, all the values in the training set are never used to evaluate. It would be unfair to build the model with the training set and then evaluate with the training set, but we are not getting as full a picture of the model performance as possible.
        - Here’s the code if you want to try running yourself and seeing the varying values of the metrics:
            - import pandas as pd
            - from sklearn.linear_model import LogisticRegression
            - from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
            - from sklearn.model_selection import train_test_split
            - import numpy as np
            -
            - df = pd.read_csv('https://sololearn.com/uploads/files/titanic.csv')
            - df['male'] = df['Sex'] == 'male'
            - X = df[['Pclass', 'male', 'Age', 'Siblings/Spouses', 'Parents/Children', 'Fare']].values
            - y = df['Survived'].values
            -
            - X_train, X_test, y_train, y_test = train_test_split(X, y)
            -
            - # building the model
            - model = LogisticRegression()
            - model.fit(X_train, y_train)
            -
            - # evaluating the model
            - y_pred = model.predict(X_test)
            - print(" accuracy: {0:.5f}".format(accuracy_score(y_test, y_pred)))
            - print("precision: {0:.5f}".format(precision_score(y_test, y_pred)))
            - print("   recall: {0:.5f}".format(recall_score(y_test, y_pred)))
            - print(" f1 score: {0:.5f}".format(f1_score(y_test, y_pred)))
    - Multiple Training and Test Sets:
        - Take the dataset as set of 5 chunks
        - 

        - For these 5 chunks: 4 will be training and 1 will be testing


1 Train Train Train Train Test
2 Train Train Train Test Train
3 Train Train Test Train Train
4 Train Test Train Train Train
5 Test Train Train Train Train

    - ■
- Building and Evaluating with Multiple Training and Test Sets:
    - Processing for creating multiple training and test set is called k-fold cross validation
    - K is the number of chunks we split a dataset into
    - Standard practice is using 5
    - We don’t actually need these models and want to build the best possible model. The best possible model is going to be a model that uses all of the data. So we keep track of our calculated values for our evaluation metrics and then build a model using all of the data.

- K Fold Cross Validation in Sklearn:
    - KFold Class:
        - Simple Example:
            - Start with 6 datapoints and 2 features:
                - X = df[['Age', 'Fare']].values[:6]
                - y = df['Survived'].values[:6]
            - Make KFold class (contains two parameters: n_split (number of chunks) and shuffle (randomize it or not))
                - kf = KFold(n_splits=3, shuffle=True)
            - KFold class has a split method that creates 3 splits for data:
                - list(kf.split(X))
            - Whole Simple Example:
                - from sklearn.model_selection import KFold
                - import pandas as pd
                -
                - df = pd.read_csv('https://sololearn.com/uploads/files/titanic.csv')
                - X = df[['Age', 'Fare']].values[:6]
                - y = df['Survived'].values[:6]
                -
                - kf = KFold(n_splits=3, shuffle=True)
                - for train, test in kf.split(X):

        - print(train, test)
- Creating Training and Test Sets with the Folds:
    - First let’s pull out the first split.
        - splits = list(kf.split(X))
        - first_split = splits[0]
        - print(first_split)
        - # (array([0, 2, 3, 5]), array([1, 4]))
    - The first array is the indices for the training set and the second is the indices for the test set. Let’s create these variables.
        - train_indices, test_indices = first_split
        - print("training set indices:", train_indices)
        - print("test set indices:", test_indices)
        - # training set indices: [0, 2, 3, 5]
        - # test set indices: [1, 4]
    - Now we can create an X_train, y_train, X_test, and y_test based on these indices.
        - X_train = X[train_indices]
        - X_test = X[test_indices]
        - y_train = y[train_indices]
        - y_test = y[test_indices]
    - If we print each of these out, we’ll see that we have four of the datapoints in X_train and their target values in y_train. The remaining 2 datapoints are in X_test and their target values in y_test.
        - print("X_train")
        - print(X_train)
        - print("y_train", y_train)
        - print("X_test")
        - print(X_test)
        - print("y_test", y_test)
        -
    - Run this code to see the results:
        - from sklearn.model_selection import KFold
        - import pandas as pd
        - df = pd.read_csv('https://sololearn.com/uploads/files/titanic.csv')
        - X = df[['Age', 'Fare']].values[:6]
        - y = df['Survived'].values[:6]
        -
        - kf = KFold(n_splits=3, shuffle=True)
        -
        - splits = list(kf.split(X))
        - first_split = splits[0]
        - train_indices, test_indices = first_split
        - print("training set indices:", train_indices)

        - print("test set indices:", test_indices)
        -
        - X_train = X[train_indices]
        - X_test = X[test_indices]
        - y_train = y[train_indices]
        - y_test = y[test_indices]
        - print("X_train")
        - print(X_train)
        - print("y_train", y_train)
        - print("X_test")
        - print(X_test)
        - print("y_test", y_test)
    - At this point, we have training and test sets in the same format as we did using the train_test_split function.
- Build a Model:
    - Entirety of the code to build and score the model on the first fold of a 5-fold cross validation:
        - from sklearn.model_selection import KFold
        - from sklearn.linear_model import LogisticRegression
        - import pandas as pd
        -
        - df = pd.read_csv('https://sololearn.com/uploads/files/titanic.csv')
        - df['male'] = df['Sex'] == 'male'
        - X = df[['Pclass', 'male', 'Age', 'Siblings/Spouses', 'Parents/Children', 'Fare']].values
        - y = df['Survived'].values
        -
        - kf = KFold(n_splits=5, shuffle=True)
        -
        - splits = list(kf.split(X))
        - train_indices, test_indices = splits[0]
        - X_train = X[train_indices]
        - X_test = X[test_indices]
        - y_train = y[train_indices]
        - y_test = y[test_indices]
        -
        - model = LogisticRegression()
        - model.fit(X_train, y_train)
        - print(model.score(X_test, y_test))
    - So far, we've essentially done a single train/test split. In order to do a k-fold cross validation, we need to use each of the other 4 splits to build a model and score the model.
- Loops Over All the Folds:

        - How to Loop over all the folds:
            - scores = []
            - kf = KFold(n_splits=5, shuffle=True)
            - for train_index, test_index in kf.split(X):
            -   X_train, X_test = X[train_index], X[test_index]
            -   y_train, y_test = y[train_index], y[test_index]
            -   model = LogisticRegression()
            -   model.fit(X_train, y_train)
            -   scores.append(model.score(X_test, y_test))
            - print(scores)
            - # [0.75847, 0.83146, 0.85876, 0.76271, 0.74011]
        - You can check the accuracy by doing this:
            - print(np.mean(scores))
            - # 0.79029
        - Now that we've calculated the accuracy, we no longer need the 5 different models that we've built. For future use, we just want a single model. To get the single best possible model, we build a model on the whole dataset. If we're asked the accuracy of this model, we use the accuracy calculated by cross validation (0.79029) even though we haven't actually tested this particular model with a test set:
            - final_model = LogisticRegression()
            - final_model.fit(X, y)
        - Expect to get slightly different values every time you run the code. The KFold class is randomly splitting up the data each time, so a different split will result in different scores, though you should expect the average of the 5 scores to generally be about the same.
- Model Comparison:
    - Let's use our techniques to compare three models:
        - A logistic regression model using all of the features in our dataset
        - A logistic regression model using just the Pclass, Age, and Sex columns
        - A logistic regression model using just the Fare and Age columns
        - Evaluation techniques are essential for deciding between multiple model options.
    - Building the Models with Scikit-learn
        - Let's write the code to build the two models in scikit-learn. Then we'll use k-fold cross validation to calculate the accuracy, precision, recall and F1 score for the two models so that we can compare them.
        - First, we import the necessary modules and prep the data as we've done before.
            - from sklearn.model_selection import KFold
            - from sklearn.linear_model import LogisticRegression
            - from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

            - import pandas as pd
            - import numpy as np
            - df = pd.read_csv('https://sololearn.com/uploads/files/titanic.csv')
            - df['male'] = df['Sex'] == 'male'
        - Now we can build the KFold object. We'll use 5 splits as that's standard. Note that we want to create a single KFold object that all of the models will use. It would be unfair if different models got a different split of the data.
            - kf = KFold(n_splits=5, shuffle=True)
            - Now we'll create three different feature matrices X1, X2 and X3. All will have the same target y.
            - X1 = df[['Pclass', 'male', 'Age', 'Siblings/Spouses', 'Parents/Children', 'Fare']].values
            - X2 = df[['Pclass', 'male', 'Age']].values
            - X3 = df[['Fare', 'Age']].values
            - y = df['Survived'].values
        - Since we'll be doing it several times, let's write a function to score the model. This function uses the KFold object to calculate the accuracy, precision, recall and F1 score for a Logistic Regression model with the given feature matrix X and target array y.
        - Then we call our function three times for each of our three feature matrices and see the results.
            - print("Logistic Regression with all features")
            - score_model(X1, y, kf)
            - print()
            - print("Logistic Regression with Pclass, Sex & Age features")
            - score_model(X2, y, kf)
            - print()
            - print("Logistic Regression with Fare & Age features")
            - score_model(X3, y, kf)
        - Expect to get slightly different results every time you run the code. The k-fold splits are chosen randomly, so there will be a little variation depending on what split each datapoint ends up in.
    - Choosing a Best Model
        - Let's look at the results from the previous part.
            - Logistic Regression with all features
            - accuracy: 0.7959055418015616
            - precision: 0.764272127669388
            - recall: 0.6783206767486641
            - f1 score: 0.7163036778464393
            -
            - Logistic Regression with Pclass, Sex & Age features
            - accuracy: 0.7981908207960389

            - precision: 0.7715749823848419
            - recall: 0.6830371999703425
            - f1 score: 0.7232930032930033
            -
            - Logistic Regression with Fare & Age features
            - accuracy: 0.6538944962864216
            - precision: 0.6519918328980114
            - recall: 0.23722965720416847
            - f1 score: 0.34438594236494796
        - If we compare the first two models, they have almost identical scores. The third model has lower scores for all four metrics. The first two are thus much better options than the third. This matches intuition since the third model doesn't have access to the sex of the passenger. Our expectation is that women are more likely to survive, so having the sex would be a very valuable predictor.
        - Since the first two models have equivalent results, it makes sense to choose the simpler model, the one that uses the Pclass, Sex & Age features.
        - Now that we've made a choice of a best model, we build a single final model using all of the data.
            - model = LogisticRegression()
            - model.fit(X1, y)
            -
            - Now we can make a prediction with our model.
            - model.predict([[3, False, 25, 0, 1, 2]])
            - # Output: [1]
        - We have only tried three different combinations of features. It's possible a different combination would also work

Decision Tree Model:

- What is a Decision Tree:
    - A Nonparametric Machine Learning Algorithm:
        - This is a type of nonparametric algorithm that is not defined by a list of parameters (coefficients used)
    - Tree Terminology:
        - Decision tree are very easy to interpret:
            - Like a flow chart of question that give you answers about a datapoint

            - Each rectangle is called a node and when nodes have a feature to split is called internal node
            - The very first internal node at the top is called the root node
            - The final nodes where the prediction occurs is called a leaf node
            - Internal nodes all have two nodes below the which are call children
        - Diagram of Decision Tree:
    - Running through a decision tree:
        - Depending on the initial condition and factor (10 year old, male, P Class: 2), you start at the first node which could be general category that works well
        - After the first condition is meet you either move left or right depending on what the condition is so if it's a male on titanic they move to the right
        - After that they reach internal node about Age and if certain confition are meet for age then the item move left or right since the target was 10 years old they move left
        - Finally, with P Class, since the target was a Class 2 P Class this allows for them to move right and this determines that this 10 year male survived the crash of the Titantic
- How to Build a Decision Tree:
    - What makes a Good Split:
        - To do this we need to score every possible split so we an choose the split with the highest score
        - Want to perfectly split the data and this is done by using the information gain
        - This is a value ranging from 0-1 which tells use how useless or useful the split will be
        - Example from Titanic Set:
            - You can split the set with age, but with threhold of >30 or <=30
            - That split would let the left hand side have:
                - Survived –> 197
                - Didn't survived -> 328

            - Right side:
                - Survived: 145
                - Didn’t survive: 217
        - This type of split does not work since 40% of the ppl survived and this split does not give us meaninful data
        - If we use sex now that will result in:
            - Left side:
                - Survived: 233
                - Didn’t sirvive: 81
            - Right side:
                - Survived: 109
                - Didn’t survive: 464
            - This is a good split since majority of female lived while the majority of the males did not survive
            - This is close to the even split this data set needs
    - Gini Impurity:
        - Measure of how pure a set is
    - Entropy:
        - Another measurement of purity of a set
    - Information Gain:
        - Use both Entropy and Gini Impurity to figure out how to get \
- Decision Trees in Scikit-learn:
    - How to use it:
        - Importing the Decision Tree:
            - rom sklearn.tree import DecisionTreeClassifier
        - Creating the object:
            - model = DecisionTreeClassifier()
        - Train/Test split using random state:

        - X_train, X_test, y_train, y_test = train_test_split(X,y,random_state 22)
    - Fit method to train the model:
        - model.fit(X_train, y_train)
    - Predict method to see if the model predicted correctly:
        - Print(model.print([[3,True,22,1,0,7.25]])
- Scoring a Decision Tree:
    - Print("accuracy", model.score(X_test,y_test))
    - Y_predict = model.predict(X_test)
    - Print("precision", precision_score(y_test,y_predict))
    - Print("recall",recall_score(y_test,y_predict))
- Gini vs Entropy:
    - Default impurity criterion is Gini Impurity
    - Decision Tree with Entropy:
        - Dt = DecisionTreeClassifer(criterion = 'entropy')
    - Comparison of the two types of impurities:
        - kf = KFold(n_splits=5, shuffle=True) for criterion in ['gini', 'entropy']:
            - print("Decision Tree - {}".format(criterion))
            - accuracy = []
            - precision = []
            - recall = []
            - for train_index, test_index in kf.split(X):
                - X_train, X_test = X[train_index], X[test_index] y_train, y_test = y[train_index],
                - y[test_index] dt =
                - DecisionTreeClassifier(criterion=criterion) dt.fit(X_train, y_train)
                - y_pred = dt.predict(X_test)

                - accuracy.append(accuracy_score(y_test, y_pred))
                - precision.append(precision_score(y_test, y_pred))
                - recall.append(recall_score(y_test, y_pred))
            - print("accuracy:", np.mean(accuracy)) print("precision:", np.mean(precision)) print("recall:", np.mean(recall))
    - Visualizing Decision Trees:
        - Import:
            - from sklearn.tree import export_graphviz
            - dot_file = export_graphviz(dt, feature_names=feature_names)
        - Import graphviz mod to covert it to png image:
            - Import graphviz
            - Graph = graphviz.Source(dot_file)
        - Render to create file and give it format:
            - graph.render(filenames=’tree’,format =’png’, cleanup =True)
        - Whole Code for render Decision Tree for any model:
            - From sklearn.tree import export_graphviz
            - Import graphviz
            - From Ipython.display import Image
            - Feature_names = [‘Pclass’, ‘male’]
            - x = df[feature_names].values
            - y = df[‘Survived’].values
            - dt = DecisionTreeClassifier()
            - dt.fit(X,y)
            - dot_file = export_graphviz(dt,feature_names=feature_names)
            - graph =graphviz.Source(dot_file)
            - graph.render(filename=’tree’, format=’png’, cleanup=True)
- Overfitting:

    - Overfitting is when you do a good job of building a model for the training set, but does not work well on the test set
    - Pruning:
        - To fix overfitting, pruning is needed
        - Pre-pruning is the process of having rules of when to stop builiding a tree
        - Post-pruning is the process of building the whole tree and then review the tree to decide which leaves to remove to make the tree smaller
    - Pre-Pruning:
        - Using this technique since it easier to do
        - Common techniques for pre-pruning:
            - Max Depth:
                - Only grow the tree up to certain depth or height of the tree. Max depth of 3 would create at most 3 split for each datapoint
            - Leaf Size:
                - Don't split a node if the number of samples at that node is under a threshold
            - Number of leaf nodes:
                - Limit the number of leaf nodes allowed in the tree
            - Too much pruning could result in underfitting
- Pruning your Decision Tree in Scikit-learn:
    - Same techniques used above in code format:
    - Example:
        - Create a Decision Tree with max depth 3, min sample per leaf 2, and max number of leaf node 10
            - dt = DecisionTreeClassifer(max_depth =3, min_samples_leaf = 2, max_leaf_nodes =10)
    - Grid Search:
        - Used to find the best values for pre-pruning

        - Class is called GridSearchCV:
            - from sklearn.model_selection import GridSearchCV
        - GridSearchCV parameters:
            - The model
            - Param grid -> dictionary of parameters names and possible values
            - What metric to use (accuracy, precision, recall, etc)
            - How many fold for k fold cross validation?
        - Example of this:
            - Param_grid = {
                - 'max_depth' : [5,15,25],
                - 'min_samples_leaf' : [1,3],
                - "max_leaf_nodes' : [10,20,35,50]}
        - Creating a gridsearch object:
            - dt = DecisionTreeClassifer()
            - gs = GridSearchCV(dt,param_grid,scoring = 'f1', cv=5)
        - Use fit method to try all possible combo that would give good depth, min sample leaf, and max leaf node
            - gs.fit(X,y)
        - To see the best parameters to use:
            - print("best params:" gs.best_params_)
        - Best score used to tell us score of winning model:
            - Print('best score', gs.best_score_)
- Pros and Cons of Decision Trees:
    - Computation:
        - DT are very computationally expensive to build
        - Every node we are trying every single feature and threshold as possible split

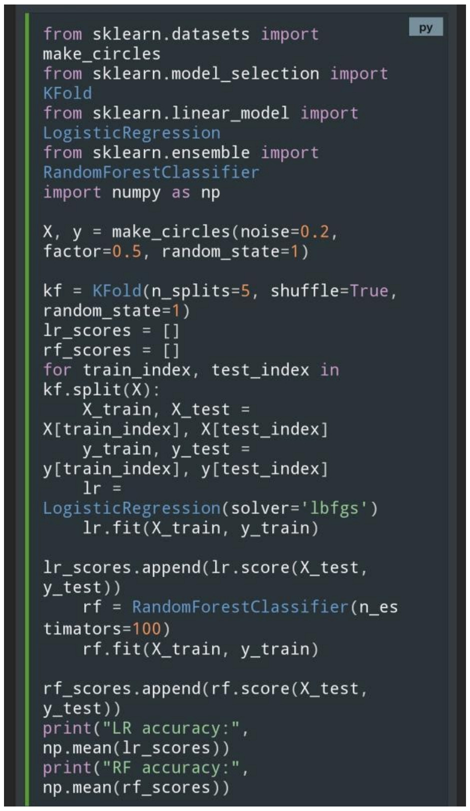
        - Calculate the information gain of each of these possible splits
        - Predicting decision tree is computation inexenpsive due to it being a series of yes/no questions
        - Care more about the prediction sincce those need to happen in real time while a user is waiting for a result
    - Performance:
        - Perform decent when depending on the dataset, but they tend to overfit
        - Pruning is required
        - Take time to work on par with other models that require no tuning
    - Interpretability:
        - Explain of the prediction are simple and follow a hierarchy structure

Random Forest Model:

- Definition of a Random Forest:
    - Improving on Decision Trees:
        - Random trees are model built with mutiple trees and the goal is to take advantage of decision trees while mitgating the variance issues
    - Bootstrapping:
        - Bootstrapping sample is a random sample of datapoints where we randomly select with replacement datapoints from our original datatset to create a dataset of the same size
        - Some points will appear mutiple times and done to mimic mutippe samples
    - Bagging Decision Trees:
        - Bootstrapuing Aggregation or Bagging is technique for reducing the variance in an individual model by creating an ensemble from mutiple modelbs built on bootstrapped samples
        - Create mutiple (usually 10) bootstrapped resamples of our training datasetso each resample will have # datapoints randomly cchosen from our traning set
        - Make prediction with each of the (usually 10) decision trees and then each decision tree gets a vote. The most votes is the final predction
    - Decorrelate the Trees:
        - Adding restriction to the model when building each decision tree allows for it to have more variation (decorrelate the trees)
        - Each decision tree within a random forrest is probably worse than standard decision tree, but when we average them we get a very strong model

        - Standard choice is the number of features squared
- Random Forests with Sklearn:
    - Code for Random Forest using Breast Cancer Data:
    - import pandas as pd from sklearn.datasets
    - import load_breast_cancer cancer_data = load_breast_cancer()
    - df = pd.DataFrame(cancer_data['data'], columns=cancer_data['feature_names']) df['target'] = cancer_data['target']
    - X = df[cancer_data.feature_names].values
    - y = df['target'].values print('data dimensions', X.shape)
    - from sklearn.ensemble import RandomForestClassifier
    - from sklearn.model_selection import train_test_split X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=101)
    - rf = RandomForestClassifier() rf.fit(X_train, y_train)
    - first_row = X_test[0] print("prediction:", rf.predict([first_row]))
    - print("true value:", y_test[0]) # prediction: [1] # true value: 1
    - print("random forest accuracy:", rf.score(X_test, y_test)) # random forest accuracy: 0.965034965034965
    - dt = DecisionTreeClassifier() dt.fit(X_train, y_train) print("decision tree accuracy:", dt.score(X_test, y_test)) # decision tree accuracy: 0.9020979020979021
    - dt = DecisionTreeClassifier() dt.fit(X_train, y_train) print("decision tree accuracy:", dt.score(X_test, y_test)) # decision tree accuracy: 0.9020979020979021
- Tuning a Random Forest:
    - Random Forest Parameters:
        - Since Random Forest is made up of Decision Trees it will have the same max_depth, min_samples_leaf, and max_leaf_nodes
        - Two new parameters: n_estimator (number of trees) and max_features (the number of feautes to consider at each split) are added
        - Default for random trees:
            - rf = RandomForestClassifier(max_features =5)
        - Default for number of estimators:
            - rf = RandomForestClassifier(n_estimators =10)
    - Grid Search:
        - Used for optimal choice of parameters
        - Param_grid = {
            - ‘n_estimators’ : [10,25,50,75,100], }
        - Creating a random forest and grid search objects:
            - rf= RandomForestClassifier()
            - gs=GridSearchCV(rf,param_grid,cv=5)
        - Fit method for run the grid search
            - gs.fit(X,y)
            - print(‘best param’, gs.best_params_)
        - using F1 test:
            - rf = RandomForestClassifier(random_state=123)
            - gs = GridSearchCV(rf, param_grid, scoring = ‘f1’, cv=5)
            - gs.fit(X,y)
            - print(“best param:”, gs.best_params_)
            - #best param -> {n estimator: 25}

        - You can add max_features and parameter values to the param_grid dictionary to compare more decision trees
    - Elbow Graph:
        - The number of trees in a random forest will not hurt the performance, but at some point the system will level out
        - Elbow Graphs are used to find the sweet spot in which number of trees in random forest is best for performance
        - To find the optimal value:
            - n_estimators = list(range(1,101))
            - param_grid = { 'n_estimators' : n_estimator, }
            - rf = RandomForestClassifier()
            - gs = GridSearchCV(rf, param_grid, cv=5_
            - gs.fit(X,y)
        - To find the value we are looking for we need to look at the cv_result section which contains a value called mean_test_score
            - scores = gs.cv_results_['mean_test_score']
            - #[0.91564,0.906,....]
        - Use matplotlib to graph the various results:
            - import matplotlib.pyplot as plt
            - scores = gs.cv_results_['mean_test_score']
            - plt.plot(n_estimators,score)
            - plt.xlabel("n_estimators")
            - plt.ylabel("accuracy")
            - plt.xlim(0,100)
            - plt.ylim(0.9,1)
            - plt.show()
        - Want to choose the minimum number that results in the highest performance
- Feature Importance:
    - Feature importance shows the relative importance of each feature
    - The numbers are scaled down and sum to 1
    - Example of random forest with n_estimator = 10:
        - rf=RandomForestClassifier(n_estimators=10,random_state=111)
        - rf.fit(X_train,y_train)
        - ft_imp = pd.Series(rf.features_importances_,index=cancer_data.feature_names).sort_ values(ascending=False
        - ft_imp.head(10)
    - New Model on Selected Features:
        - Why do we want to perform feature selection?:
            - Enables us to train a model faster
            - it reduces complexity of a model thus makes it easier to interpret
            - With right subset, it can improve the accuracy of the model
            - Relies on domain knowledge, some art, and luck
        - Building a new model with selected features:
            - rf=RandomForestClassifier(n_estimators=10,random_state=111)
            - rf.fit(X_train,y_train)
            - rf.score(X_test,y_test)

            - #0.965
        - Finding the worst feature whose name is worst:
            - worst_col = [col for col in df.columns if ‘worst; in col]
            - print(worst_col)
        - Creating another dataframe with the selected features:
            - X_worst = df[worst_cols]
            - X_train, X_test, y_train, y_test = train_test_split(X_worst,y,random_state=101)
        - Fit the model:
            - rf.fit(X_train, y_train)
            - rf.score(X_test,y_test)
            - #0.972
        - Improve the accuracy since we used a subset of features, a third of the total feature and removed sosme of the noise and highly correlated features which results in increased accuracy
- Random Forest Pros and Cons:
    - Performance:
        - Biggest advantage is that generally performa well without any tunning
        - Decently well on all dataset
        - Example code of this
        - 

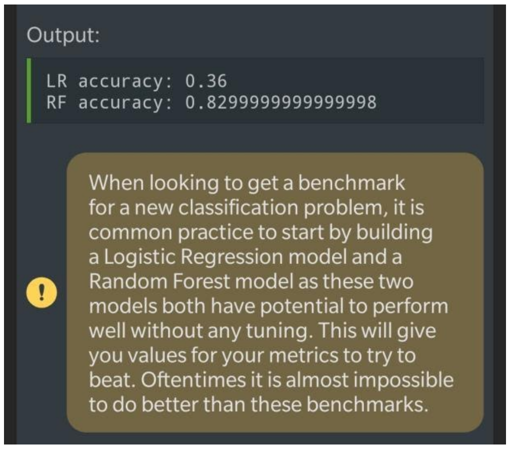
```py
from sklearn.datasets import
make_circles
from sklearn.model_selection import
KFold
from sklearn.linear_model import
LogisticRegression
from sklearn.ensemble import
RandomForestClassifier
import numpy as np

X, y = make_circles(noise=0.2,
factor=0.5, random_state=1)

kf = KFold(n_splits=5, shuffle=True,
random_state=1)
lr_scores = []
rf_scores = []
for train_index, test_index in
kf.split(X):
    X_train, X_test =
X[train_index], X[test_index]
    y_train, y_test =
y[train_index], y[test_index]
    lr =
LogisticRegression(solver='lbfgs')
    lr.fit(X_train, y_train)

lr_scores.append(lr.score(X_test,
y_test))
    rf = RandomForestClassifier(n_es
timators=100)
    rf.fit(X_train, y_train)

rf_scores.append(rf.score(X_test,
y_test))
print("LR accuracy:",
np.mean(lr_scores))
print("RF accuracy:",
np.mean(rf_scores))
```

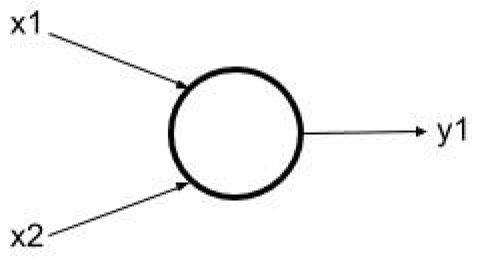

Output:

```
LR accuracy: 0.36
RF accuracy: 0.8299999999999998
```

When looking to get a benchmark for a new classification problem, it is common practice to start by building a Logistic Regression model and a Random Forest model as these two models both have potential to perform well without any tuning. This will give you values for your metrics to try to beat. Oftentimes it is almost impossible to do better than these benchmarks.

- Interpretability:
  - Not a good choice when looking at interpretability since several different decision trees are made and then averaged this will have misleading results that work, but are not good for interpretation
  - Most case this is not that important
- Computation:
  - Slow to build if there are lot of trees in a random forest
  - Each decision tree is faster to build than standard decision tree since of how we do not compare every feature at every split
  - Given the amount od DT within a RF it causes it to build slowly
  - This is generally not issue since computation power of computer is a lot and should make up for slow building time

- Neural Networks:
  - Neural Network Use Cases:
    - This is a popular and powerful machine learning model
    - It performs well in cases that have lots of features as they automatically do feature engineering without need domain knowledge to restructure the features
  - Biological Neural Network:
    - Neural Networks are Artifical Neural Networks (ANN) and they inspired by the biological neural networks of the brain
- A Neuron:
  - What's a Neuron?:
    - An artificial neuron or node is modeled after a biological neuron
    - Can take simple input and output something using some calculations
    - Neuron/Node looks something like this:



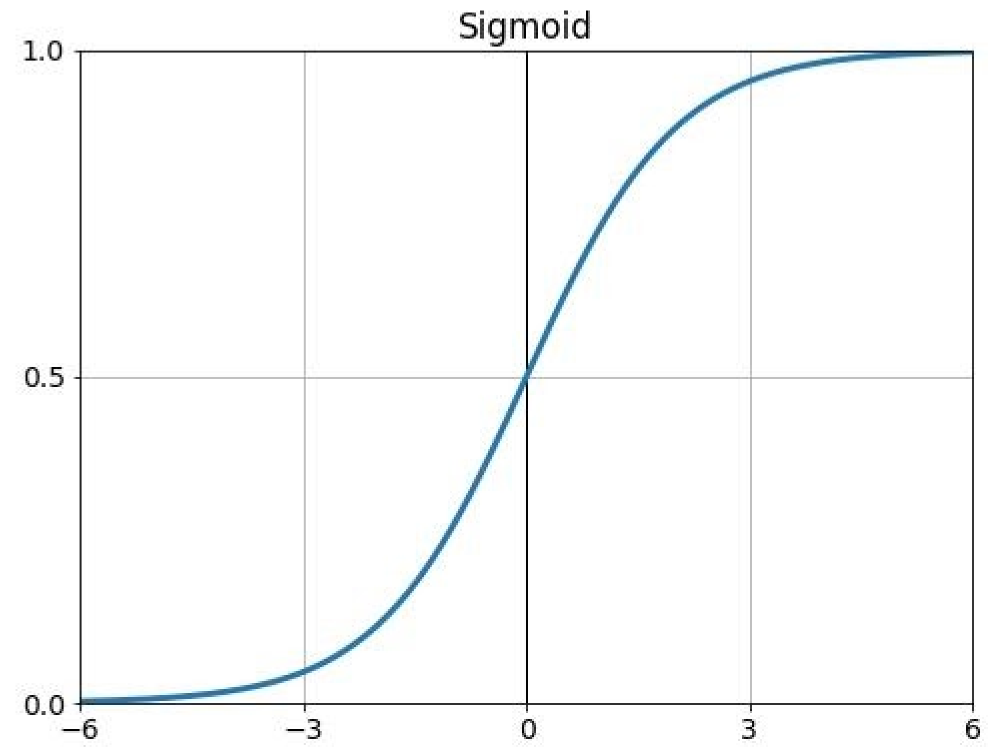
        - Each neuron can only do small amount of computation, but when they are working together they can do more work
- Neuron Computations:
    - The computation do inside the neuron looks something like this:

$$w_1x_1 + w_2x_2 + b$$

        -
    - X_1 and x_2 represent the inputs and the w_1 and w_2 represent the weights, and lastly the b represents the bias (X, w, and b are parameters of this formula)
    - After that the values are plugged into activation function which job is to condense the range to be within 0-1
    - The most common activation function is the sigmoid function, it's a similar to the logistics regression and its formula is:

$$\mathrm{sigmoid}(x) = \frac{1}{1+e}$$

        -
        - The product of this function value from 0-1
    - Shape of the Sigmoid graph:



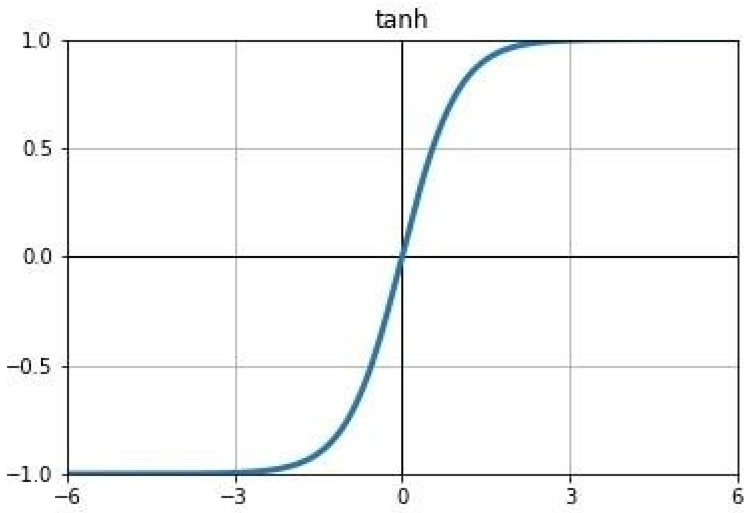
-
  - Putting both acitivation formula and regular formula together produces:

$$y = f(w_1 x_1 + w_2 x_2 + b) = \frac{1}{1 + e^{-(w_1 x_1 +}}$$

-

- Activation Functions:
  - Commonly used activation functions are sigmoid, tanh, and ReLu:
    - Tanh has similar form to sigmoid though the range is -1 to 1 and it's a hyperbolic tan function
    - Formula and graph:
      -

$$f(x) = \tanh(x) = \frac{\sinh(x)}{\cosh(x)} = \frac{e^{x} -}{e^{x} +}$$

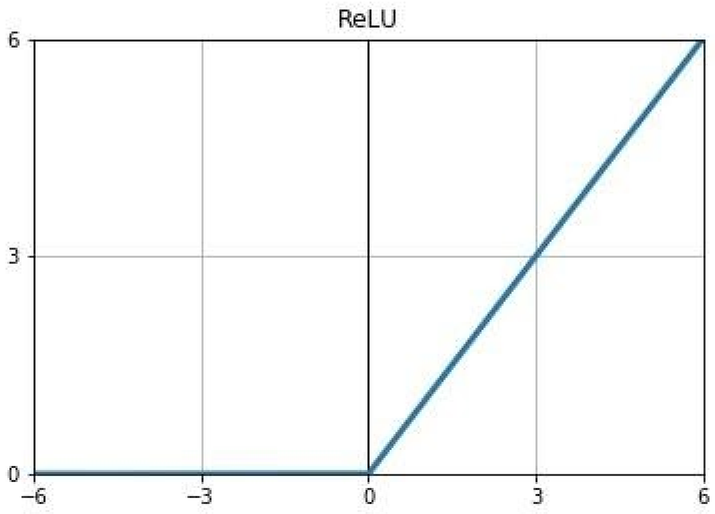


o

- ReLU stands for Rectified Linear Unit and its identidy function for positive numbers and sends negative numbers to 0
- Equation and Graph:

$$\mathrm{ReLU}(x) = \begin{cases} 0 & \text{if} \\ x & \text{if} \end{cases}$$

o



o

        - The activation will depend on the data we are given, but we can use metrics to figure out the best on to use
    - An Example:
        - Assume we have a neuron that takes 2 inputs and produces 1 output and whose activation function is the **sigmoid**. The parameters are: Weights (w1, w2) = [0, 1] Bias (b) = 2
        - If we give the neuron input (1, 2) we get the following calculation.

$$\begin{aligned} w_1x_1 + w_2x_2 + b &= 0 \times 1 + 1 \times 2 \\ &= 0 + 2 + 2 = 4 \end{aligned}$$

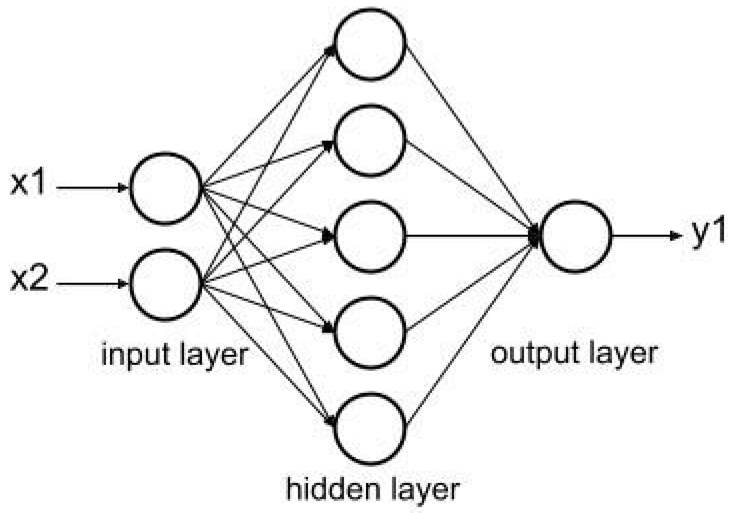
$$y = f\left(w_1x_1 + w_2x_2 + b\right) = \frac{1}{1 + e^{-4}} = 0.9$$

        -
        - The neuron yields an output of 0.9820.
        - Alternatively, if we give the neuron input (2, -2) we get the following calculation.

$$\begin{aligned} w_1x_1 + w_2x_2 + b &= 0 \times 2 + 1 \times -2 \\ &= 0 - 2 + 2 = 0 \end{aligned}$$

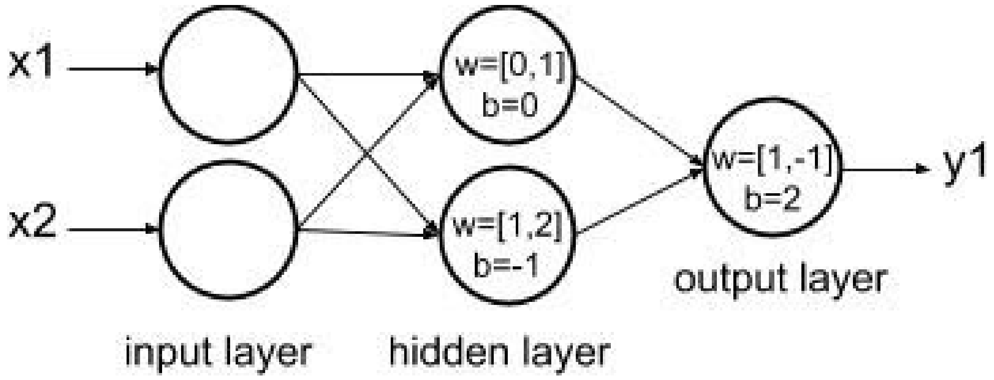
$$y = f\left(w_1x_1 + w_2x_2 + b\right) = \frac{1}{1 + e^{0}} = 0.5$$

        -
        - The neuron with this input yields an output of 0.5.
- Neural Network:
    - Multi-Layer Perceptron (MLP):
        - To create a neural network you combine neurons together so that the outputs of some neurons input into other neurons
        - Use feed forward neural networks which means that neuron only send singals in one direction
        - MLP has multiple layes which we see depicted below:



-
    - MLP has one input and output layer with a neuron or node for each input
    - The number of hidden layers and each hidden layer can have any number of nodes
    - Single Layer Perceptron is a neural network without any hidden layer and these are rarely used
    - Most neural networks are MLP with at least 1-2 hidden layers

o Example Neural Network:
    - Let's dive deeper into how this works with an example. A neural network that solves any real problem will be too large to interpret, so we will walk through a simple example.
    - We have a neural network with two inputs, a single hidden layer with two nodes and one output. The **weights** and **bias** are given in the nodes below. All the nodes use the sigmoid activation function.



    -
    - Let's see what output we get for the input (3,2).
    - Here is the output for the first node in the hidden layer.
    -
    - Here is the output from the node in the output layer. Note that this node takes the outputs from the hidden layer as input.

$$\begin{aligned} h_1 &= f(0 \times x_1 + 1 \times x_2 + \\ &= f(0 \times 3 + 1 \times 2 + 0) \\ &= f(2) \\ &= \frac{1}{1 + e^{-2}} \\ &= 0.8808 \end{aligned}$$

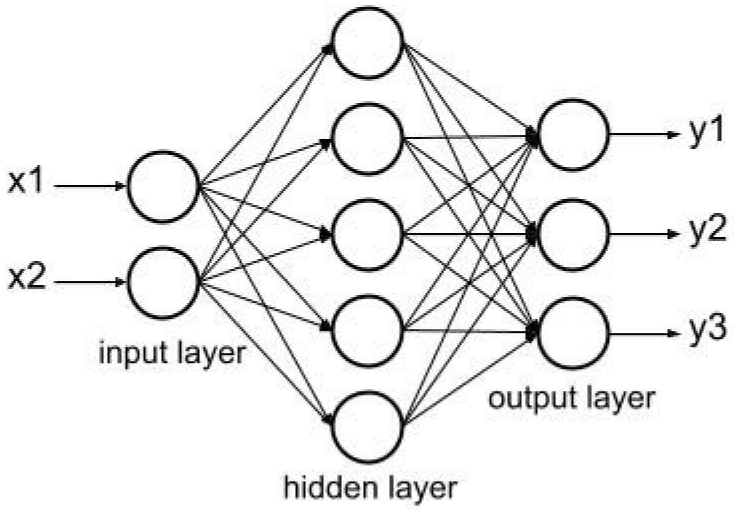
-
- Here is the output for the second node in the hidden layer.

$$\begin{aligned} h_2 &= f(1 \times x_1 + 2 \times x_2 - \\ &= f(1 \times 3 + 2 \times 2 - 1) \\ &= f(6) \\ &= \frac{1}{1 + e^{-6}} \\ &= 0.9975 \end{aligned}$$

-
- Here is the output from the node in the output layer. Note that this node takes the outputs from the hidden layer as input.

$$\begin{aligned} y_1 &= f(1 \times h_1 - 1 \times h_2 + 2) \\ &= f(1 \times 0.8808 - 1 \times 0.9975 + \\ &= f(1.8833) \\ &= \frac{1}{1 + e^{-1.8833}} \\ &= 0.8680 \end{aligned}$$

  - Thus for the input (3, 2), the output of the neural network is 0.8680.
  - To change how the neural network performs we can chane the weights and bias values
- More than 2 Target Values:
  - A nice benefit of an MLP classifier is that it easily extends to problems that have **more than 2 target values**. In the previous modules, we have dealt with predicting 0 or 1 (true or false, survived or not, cancerous or not,). In some cases, we will be choosing among 3 or more possible outputs. A neural network does this naturally. We just need to add more nodes to the output layer. For example, if we are trying to predict if an image is a bird, cat or dog, we will have three output nodes. The first (y1) measures if the image is a bird, the second (y2) measures if the image is a cat, and the third (y3) measures if the image is a dog. The model chooses the output with the highest value.

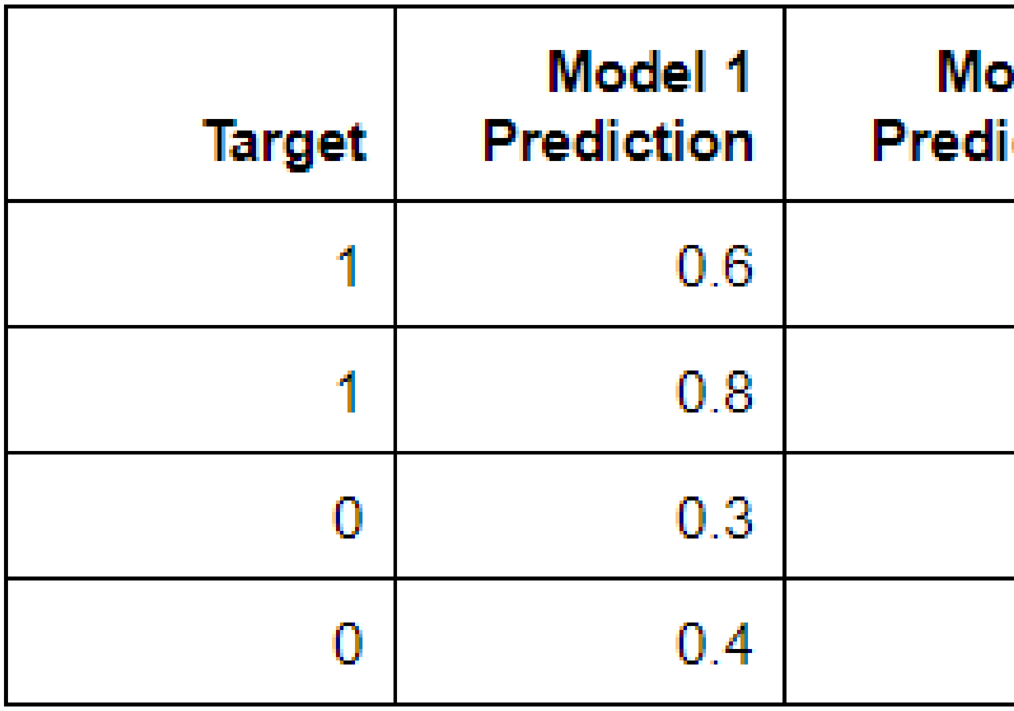


  -

        - For this example, the given image input, the neural net outputs y1= 0.3, y2 = 0.2, and y3 = 0.5
        - The model would then determine the image had a dog (y3) in it
        - We can use any classifier for a multi-class problem, but NN generalize naturally
- Training the Neural Network:
    - Loss:
        - To train a neural network we need to a loss function and this is a measure of how far off our neural network is from being perfect
        - Using cross entropy. Likelihood we used in logistic regression, but is called a different name in some context
        - Formula:

$$\text{cross entropy} = \begin{cases} p & \text{if} \\ 1-p & \text{if} \end{cases}$$

            -
        - Example of cross entropy:
            - We multiply together the cross entropy values for all the datapoints.
            - Let's say we have two models to compare on a tiny dataset with 4 datapoints. Here is a table of the true values, the predicted probabilities for model 1 and the predicted probabilities for model 2

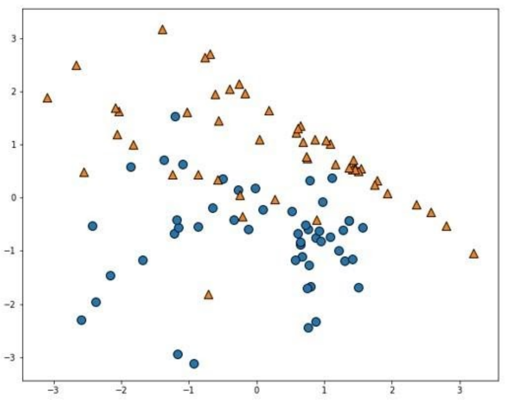


Target	Model 1 Prediction	Mo Predi
1	0.6	
1	0.8	
0	0.3	
0	0.4	

-
- The cross entropy for model 1 is as follows
    - $0.6 \times 0.8 \times (1 - 0.3) \times (1 - 0.4)$
- The cross entropy for model 2 is as follows
    - $0.5 \times 0.9 \times (1 - 0.1) \times (1 - 0.5)$
- Cross entropy will be higher the better the model is, thus since model 2 has higher cross entropy than model 1, it is the better model.
- All of this is done to find the best possible model for system at hand

o Backpropagation:
  - A neural network has a lot of parameters that we can control. There are several coefficients for each node and there can be a lot of nodes!
  - The process for updating these values to converge on the best possible model is quite complicated.
  - The neural network works backwards from the output node iteratively updating the coefficients of the nodes. This process of moving backwards through the neural network is called **backpropagation** or **backprop**.
  - We won't go through all the details here as it involves calculating partial derivatives, but the idea is that we initialize all the coefficient values and iteratively change the values so that at every iteration we see improvement in the loss function.
  - Eventually we cannot improve the loss function anymore and then we have found our **optimal model**.

        - Before we create a neural network we fix the number of nodes and number of layers
        - Then we use backprop to irealtive update all the coefficients values until we converge oin an optimal neural network
- Neural Networks in Scikit-learn:
    - Creating Artificial Dataset:
        - To test a model, we must create an artificial dataset and the size and complexity is up to you
        - Make a dataset easier to work with than a real dataset and this helps us understand how models work before applying them to real world data
        - We will use the **make_classification** function in scikit-learn. It generates a feature matrix X and target array y. We will give it these parameters:
            - **n_samples**: number of datapoints
            - **n_features**: number of features
            - **n_informative**: number of informative features
            - **n_redundant**: number of redundant features
            - **random_state**: random state to guarantee same result every time
        - You can look at the full documentation to see other parameters that you can tweak to change the result.
        - Here is the code to generate a dataset:
            - from sklearn.datasets import make_classification
              X, y = make_classification(n_features=2, n_redundant=0, n_informative=2, random_state=3)
        - Here's the code to plot the data so we can look at it visually:
            - from matplotlib import pyplot as plt
              plt.scatter(X[y==0][:, 0], X[y==0][:, 1], s=100, edgecolors='k')
              plt.scatter(X[y==1][:, 0], X[y==1][:, 1], s=100, edgecolors='k', marker='^')
              plt.show()
        - Graph:
            - Dsd



            -
            - Other functions such as make_circle and make_moons if you want to play around with more artificial datasets
    - MLPClassifier:

- Scikit-learn has an **MLPClassifier** class which is a multi-layer perceptron for classification. We can import the class from scikit-learn, create an MLPClassifier object and use the **fit method** to train.\
- EX:

  -
    ```
    from sklearn.model_selection import train_test_split
    from sklearn.neural_network import MLPClassifier
    from sklearn.datasets import make_classification

    X, y = make_classification(n_features=2, n_redundant=0,
    n_informative=2, random_state=3)
    X_train, X_test, y_train, y_test = train_test_split(X, y,
    random_state=3)
    mlp = MLPClassifier()
    mlp.fit(X_train, y_train)
    ```
  - output:

  -
    ```
    ConvergenceWarning: Stochastic Optimizer: Maximum iterations
    (200) reached and the optimization hasn't converged yet.
    % self.max_iter, ConvergenceWarning)
    ```
- You will notice that we get a **ConvergenceWarning**. This means that the neural network needs more iterations to converge on the optimal coefficients. The default number of iterations is 200. Let's up this value to 1000.
  - `mlp = MLPClassifier(max_iter=1000)`
- Now when we run this code, the neural network will converge. We can now use the score method to calculate the accuracy on the test set.
- Ex:

  -
    ```
    from sklearn.model_selection import train_test_split
    from sklearn.neural_network import MLPClassifier

    X_train, X_test, y_train, y_test = train_test_split(X, y,
    random_state=3)
    mlp = MLPClassifier(max_iter=1000)
    mlp.fit(X_train, y_train)
    print("accuracy:", mlp.score(X_test, y_test))
    ```
- NN are complicated, but scikit-learn make them approachable

o Parameters for MLPClassifier:
- There are a couple of parameters that you may find yourself needing to change in the MLPClassifier.
- You can configure the number of **hidden layers** and how many **nodes** in each layer. The default MLPClassifier will have a single hidden layer of 100 nodes. This often works really well, but we can experiment with different values. This will create an MLPCLassifier with two hidden layers, one of 100 nodes and one of 50 nodes.
  - `mlp = MLPClassifier(max_iter=1000, hidden_layer_sizes=(100, 50))`
- We saw **max_iter** in the previous part. This is the number of iterations. In general, the more data you have, the fewer iterations you need to converge. If the value is too large, it will take too long to run the code. If the value is too small, the neural network will not converge on the optimal solution.

- We also sometimes need to change **alpha**, which is the step size. This is how much the neural network changes the coefficients at each iteration. If the value is too small, you may never converge on the optimal solution. If the value is too large, you may miss the optimal solution. Initially you can leave this at the default. The default value of alpha is 0.0001. Note that decreasing **alpha** often requires an increase in **max_iter**.
- Sometimes you will want to change the **solver**. This is what algorithm is used to find the optimal solution. All the solvers will work, but you may find for your dataset that a different solver finds the optimal solution faster. The options for solver are 'lbfgs', 'sgd' and 'adam'.
- Run this code in the playground and try changing the parameters for the MLPClassifier. The code uses a **random_state** to ensure that every time you run the code with the same parameters you will get the same output.

  -
    ```
    from sklearn.model_selection import train_test_split
    from sklearn.neural_network import MLPClassifier

    X_train, X_test, y_train, y_test = train_test_split(X, y,
    random_state=3)
    mlp = MLPClassifier(max_iter=1000, hidden_layer_sizes=(100, 50),
    alpha=0.0001, solver='adam', random_state=3)
    mlp.fit(X_train, y_train)
    print("accuracy:", mlp.score(X_test, y_test))
    ```

o Predicating Handwritten Digits:

- The MNIST Dataset:
  - NIST is Nationa Institute of Standard and Technioology and M = modified
  - This is a database of image of handwritten digits and you can build a classfier based on them
- In scikit-learn we can load the dataset using the **load_digits function**. To simplify the problem, we will initially only be working with two digits (0 and 1), so we use the n_class parameter to limit the number of target values to 2.

  -
    ```
    from sklearn.datasets import load_digits
    X, y = load_digits(n_class=2, return_X_y=True)
    ```
- We can see the dimensions of X and y and what the values look like as follows:

  -
    ```
    print(X.shape, y.shape)
    print(X[0])
    print(y[0])
    ```
  - output:

  -
    ```
    (360, 64) (360,)
    [ 0. 0. 5. 13. 9. 1. 0. 0. 0. 0. 13. 15. 10. 15. 5. 0. 0. 3.
    15. 2. 0. 11. 8. 0. 0. 4. 12. 0. 0. 8. 8. 0. 0. 5. 8. 0.
    0. 9. 8. 0. 0. 4. 11. 0. 1. 12. 7. 0. 0. 2. 14. 5. 10. 12.
    0. 0. 0. 0. 6. 13. 10. 0. 0. 0.]
    0
    ```
- We see that we have 300 **datapoints** and each datapoint has 64 **features**. We have 64 features because the image is 8 x 8 pixels and we have 1

feature per pixel. The value is on a grayscale where 0 is black and 16 is white.

  - To get a more intuitive view of the datapoint, reshape the array to be 8x8:
    - print(X[0].reshape(8, 8))
    - output:

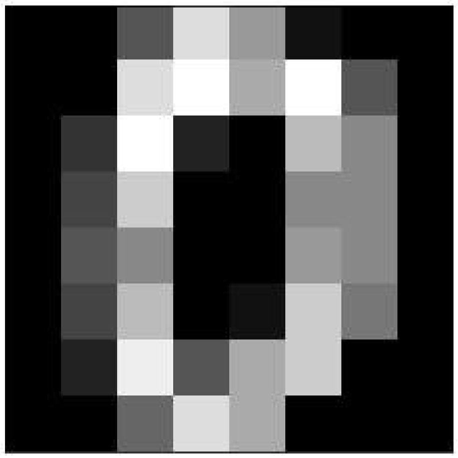
    -
      ```
      [[ 0. 0. 5. 13. 9. 1. 0. 0.]
      [ 0. 0. 13. 15. 10. 15. 5. 0.]
      [ 0. 3. 15. 2. 0. 11. 8. 0.]
      [ 0. 4. 12. 0. 0. 8. 8. 0.]
      [ 0. 5. 8. 0. 0. 9. 8. 0.]
      [ 0. 4. 11. 0. 1. 12. 7. 0.]
      [ 0. 2. 14. 5. 10. 12. 0. 0.]
      [ 0. 0. 6. 13. 10. 0. 0. 0.]]
      ```
  - We can see that this is a 0, though we will see in the next part that we can draw the image more clearly.

    -
      ```
      from sklearn.datasets import load_digits
      X, y = load_digits(n_class=2, return_X_y=True)
      print(X.shape, y.shape)
      print(X[0])
      print(y[0])
      print(X[0].reshape(8, 8))
      ```
  - There are different versions of this dataset with more pixels and wih colors

- Drawing the Digits:
  - You can build a model without ever looking at a visual representation of the images, but it can sometimes be helpful to draw the image.

    We use the matplotlib function **matshow** to draw the image.
    The **cmap** parameter is used to indicate that the image should be in a grayscale rather than colored.import matplotlib.pyplot as plt
    from sklearn.datasets import load_digits

    ```
    X, y = load_digits(n_class=2, return_X_y=True)
    plt.matshow(X[0].reshape(8, 8), cmap=plt.cm.gray)
    plt.xticks(()) # remove x tick marks
    plt.yticks(()) # remove y tick marks
    plt.show()
    ```
  - **Output:**



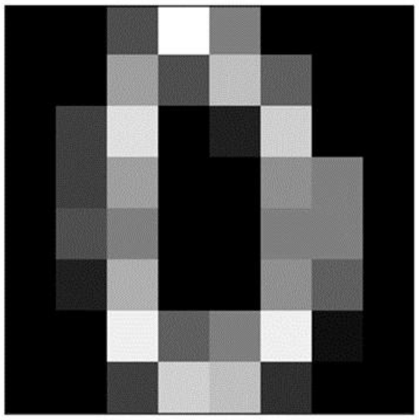
-
- You can see that with 64 pixels the image is very pixelated and these blurry images can build great models

o MLP for MNIST Dataset:

- Now let’s use the MLPClassifier to build a model for the MNIST dataset.
- We will do a **train/test split** and train an MLPClassifier on the training set.

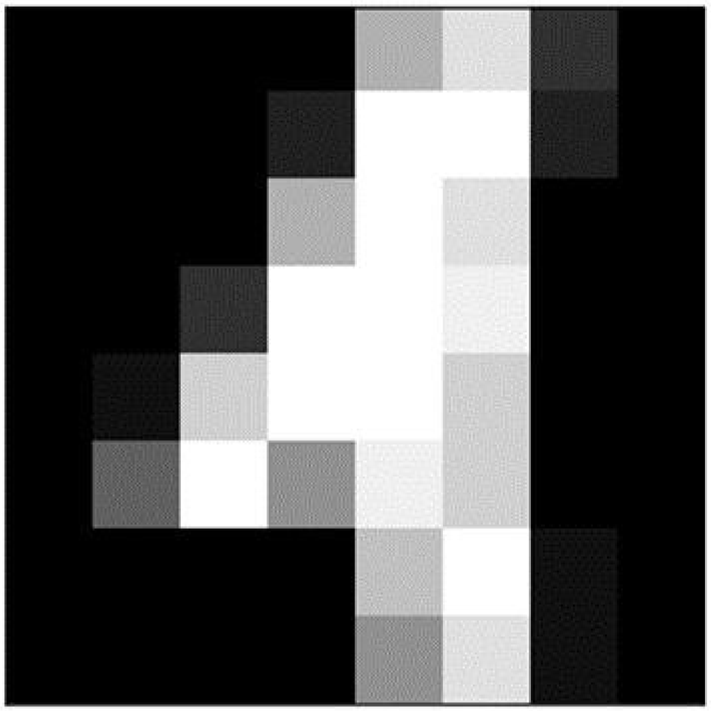
  -
    ```
    X_train, X_test, y_train, y_test = train_test_split(X, y,
    random_state=2)
    mlp = MLPClassifier()
    mlp.fit(X_train, y_train)
    ```
- We do not get a warning, so the default number of iterations is adequate in this case.
- Let’s look at how the model predicts the first datapoint in the test set. We use **matplotlib** to draw the images and then show the model’s prediction.

  -
    ```
    x = X_test[0]
    plt.matshow(x.reshape(8, 8), cmap=plt.cm.gray)
    plt.xticks(())
    plt.yticks(())
    plt.show()
    print(mlp.predict([x]))
    # 0
    ```
  - **Output:**



-
- We can see that this is a 0 and that our model correctly predicts 0.
- Similarly, let’s look at the second datapoint.

    -
      ```
      x = X_test[1]
      plt.matshow(x.reshape(8, 8), cmap=plt.cm.gray)
      plt.xticks(())
      plt.yticks(())
      plt.show()
      print(mlp.predict([x]))
      # 1
      ```
    - **Output:**



o

- This is clearly a 1 and our model again gets the correct prediction
  We can also see the model’s accuracy on the entire test set.

-
  ```
  print(mlp.score(X_test, y_test))
  # 1.0
  ```
- We can also see the model's accuracy on the entire test set. Thus our model gets 100% accuracy.
- 0 and 1 are two of the easier digits to distinguish, but we will see that the model can perform well with distinguishing harder examples

o Classifying all 10 Digits:

- Since **neural networks** easily generalize to handle multiple outputs, we can just use the same code to build a classifier to distinguish between all ten digits.
- This time when we load the digits, we do not limit the number of classes.

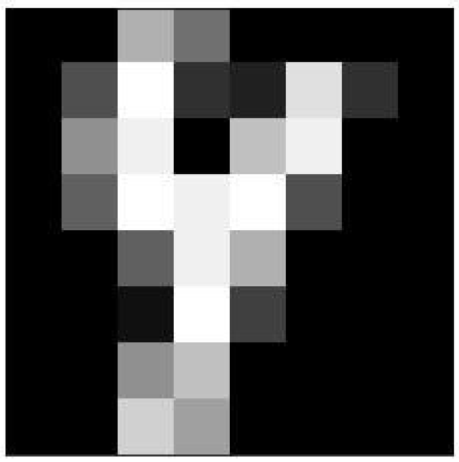
  -
    ```
    X, y = load_digits(return_X_y=True)
    X_train, X_test, y_train, y_test = train_test_split(X, y,
    random_state=2)
    mlp = MLPClassifier(random_state=2)
    mlp.fit(X_train, y_train)

    print(mlp.score(X_test, y_test))
    # 0.96
    ```

▪ So we got 96% of the datapoints in the test set correct. Let's look at the ones we got incorrect. We use a numpy mask to pull out just the datapoints we got incorrect. We pull the x values, the true y value as well as the predicted value.

    -
      ```
      y_pred = mlp.predict(X_test)
      incorrect = X_test[y_pred != y_test]
      incorrect_true = y_test[y_pred != y_test]
      incorrect_pred = y_pred[y_pred != y_test]
      ```
▪ Let's look at the first image that we got wrong and what our prediction was.

    -
      ```
      j = 0
      plt.matshow(incorrect[j].reshape(8, 8), cmap=plt.cm.gray)
      plt.xticks(())
      plt.yticks(())
      plt.show()
      print("true value:", incorrect_true[j])
      print("predicted value:", incorrect_pred[j])
      ```
    - output:



o

    - true value: 4
    - predicted value: 9
    - You can see from looking at the image that a human might also be confused. It is not obvious whether it is a 4 or a 9.

▪ Code for all 10 digits:

    -
      ```
      from sklearn.datasets import load_digits
      from sklearn.model_selection import train_test_split
      from sklearn.neural_network import MLPClassifier

      X, y = load_digits(return_X_y=True)
      X_train, X_test, y_train, y_test = train_test_split(X, y,
      random_state=2)
      mlp = MLPClassifier(random_state=2)
      mlp.fit(X_train, y_train)
      ```

```
print(mlp.score(X_test, y_test))

y_pred = mlp.predict(X_test)
incorrect = X_test[y_pred != y_test]
incorrect_true = y_test[y_pred != y_test]
incorrect_pred = y_pred[y_pred != y_test]

j = 0
print(incorrect[j].reshape(8, 8).astype(int))
print("true value:", incorrect_true[j])
print("predicted value:", incorrect_pred[j])
```

    - You can mod the code to see all of the datapoints the model predicted incorrectly

- Visualizing MLP Weights:
  - Open ML:
    - For this lesson, we will use a more granular version of the MNIST dataset. Instead of using the version in scikit-learn which has 64 pixel images, we will use a version from Open ML that has 784 pixels (28 x 28).
    - Open ML (www.openml.org) has a database of large datasets that can be used for a variety of machine learning problems. Scikit-learn has a function **fetch_openml** for directly downloading datasets from the Open ML database.
    - Use the following code to get our dataset.

      -
        ```
        from sklearn.datasets import fetch_openml
        X, y = fetch_openml('mnist_784', version=1, return_X_y=True)
        ```
    - We can briefly look at the shape of the arrays, the range of the features values, and the first few values of the target array to better understand the dataset.

      -
        ```
        print(X.shape, y.shape)
        print(np.min(X), np.max(X))
        print(y[0:5])
        Output:(70000, 784) (70000,)
        0.0 255.0
        ['5' '0' '4' '1' '9']
        ```
    - We can see that we have 70,000 datapoints with 784 features. The feature values range from 0 to 255 (which we interpret on a gray scale with 0 being white and 255 being black). The target values are the numbers 0-9. Note that the **target values** are stored as strings and not integers.
    - For our example, we will be using only the digits 0-3, so we can use the following code to segment out that portion of the dataset.

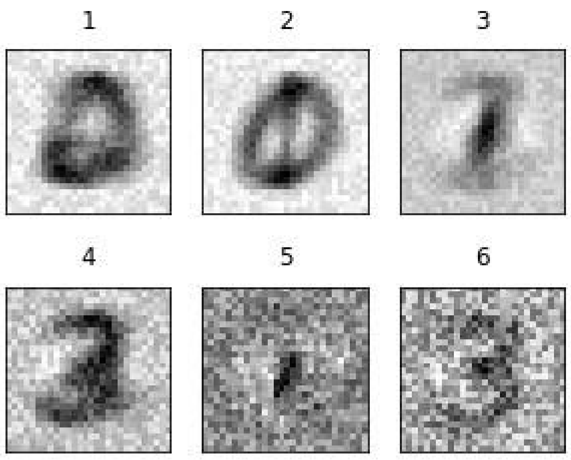
      -
        ```
        X5 = X[y <= '3']
        y5 = y[y <= '3']
        ```
    - We will be modifying some of the default parameters in the MLPClassifier to build the model. Since our goal will be to visualize the weights of the hidden layer, we will use only 6 nodes in the hidden layer so that we can look at all of them. We will use '**sgd**' (stochastic gradient descent) as our solver which requires us to decrease alpha (the learning rate).

            -
              ```
              mlp=MLPClassifier(
              hidden_layer_sizes=(6,),
              max_iter=200, alpha=1e-4,
              solver='sgd', random_state=2)

              mlp.fit(X5, y5)
              ```
        - If we run this code we will see that it converges.
    - MLPClassifier Coefficients:
        - The MLPClassifier stores the coefficients in the coefs_ attribute. Let's see what it looks like.
            - `print(mlp.coefs_)`
        - **Output:**

            -
              ```
              [array([[-0.01115571, -0.08262824, 0.00865588, -0.01127292,
              -0.01387942,
              -0.02957163],
              ...
              ```
        - First we see that it is a list with two elements.
            - `print(len(mlp.coefs_))`
            - **Output:**2
        - The two elements in the list correspond to the two layers: the **hidden layer** and the **output layer**. We have an array of coefficients for each of these layers. Let's look at the shape of the coefficients for the hidden layer.
            - `print(mlp.coefs_[0].shape)`
            - Output:
            - (784, 6)
        - We see that we have a 2-dimensional array of size 784 x 6. There are 6 nodes and 784 input values feeding into each node, and we have a weight for each of these connections.
        - Visualize the Hidden Layer:
            - To get a better understanding of what the neural network is doing, we can visualize the weights of the hidden layer to get some insight into what each node is doing.
            - We will use the matshow function from matplotlib again to draw the images. In matplotlib we can use the **subplots** function to create multiple plots within a single plot.

                -
                  ```
                  fig, axes = plt.subplots(2, 3, figsize=(5, 4))
                  for i, ax in enumerate(axes.ravel()):
                  coef = mlp.coefs_[0][:, i]
                  ax.matshow(coef.reshape(28, 28), cmap=plt.cm.gray)
                  ax.set_xticks(())
                  ax.set_yticks(())
                  ax.set_title(i + 1)
                  plt.show()
                  ```
                - **Output:**



- o
    - You can see that nodes 4 and 6 are determining if the digit is a 3. Node 1 is determining if the digit is a 0 or a 2 since you can see both of those values in the image. Not every hidden node will have an obvious use.
    - If you change the random state in the MLPClassifer, you will likely get different results and there are many equivalently optimal neural network that work differently

- Neural Networks Pros and Cons:
    - Interpretability:
        - While we can visualize the nodes in the hidden layer to understand on a high level what the neural network is doing, it is impossible to answer the question "Why did datapoint x get prediction y?" Since there are so many nodes, each with their own coefficients, it is not feasible to get a simple explanation of what the neural network is doing. This makes it a difficult model to **interpret** and use in certain business use cases.
        - Neural Networks are not a good option for interpretability.
    - Computation:
        - Neural networks can take a decent amount of **time to train**. Each node has its own coefficients and to train they are iteratively updated, so this can be time consuming. However, they are parallelizable, so it is possible to throw computer power at them to make them train faster.
        - Once they are built, neural networks are not slow to make predictions, however, they are not as fast as some of the other models.
    - Performance:
        - The main draw to neural networks is their **performance**. On many problems, their performance simply cannot be beat by other models. They can take some tuning of parameters to find the optimal performance, but they benefit from needing minimal feature engineering prior to building the model.

- A lot of simpler problems, you can achieve equivalent performance with a simpler model like logistic regression, but with large unstructured datasets, neural networks outperform other models.
- The key advantage of neural networks is their performance capabilities.