Final Portfolio

Regression Models (and Perceptron)

Perceptron:

This module is the building block for my machine learning education. It is a simple class that creates a binary model for classification between given points. For example, you can input an array of points with a training array and it will classify the points with a 1 or 0 based off the points relation to the training data. In this class, there are 4 functions:

```
"""
    In:
        rate - rate of training for the model
        niter - number of iterations of training for the model to go through.
    Returns: \

    Class object initialization
"""
def __init__(self, rate = 0.01, niter = 10):
```

```
"""
    In:
        X - data set
        y - training set
    Returns: \

    This function fits the data with a perceptron model.
    to be used with other various functions
"""
def fit(self, X, y):
```

```
"""
    In:
        X - data set
    Returns:
        int - dot product of the weight and bias
"""
def net_input(self, X):
```

```
"""
    In:
        X - data set
    Returns:
        array - Part of the input data where the y value at the same index
        equals a certain value.
"""
def predict(self, X):
```

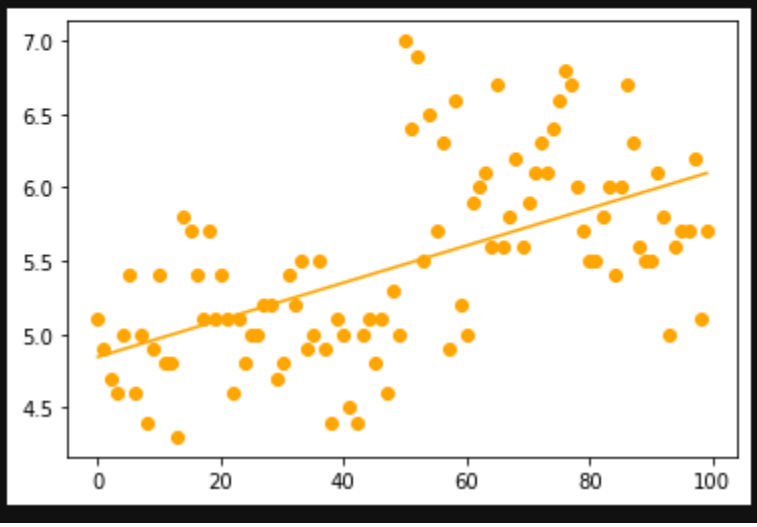
Linear Regression:

This module fits the given data with a linear equation. This equation follows the simple method of y = mx + b. However the main purpose of the class is to find the weight of the line from the points, which is the 'm' in the above equation. This process fits a linear regression line to the data. The purpose of this data is to take the relationship between 2 numbers and put it into a single number. There are 4 functions in this class:

```
    """
        In: \
        Returns: \

        Class object initialization
    """
    def __init__(self):
_____________________________________________________________________________
    """
        In:
            x - data set
            y - training set
        Returns: \

        This function fits a linear regression line to the given
        input data.
    """
    def fit(self, x, y):
_____________________________________________________________________________
    """
        In:
            inputData - data set (Commonly X)
        Returns:
            int - slope of linear prediction line
    """
    def predict(self, inputData):
_____________________________________________________________________________
    """
        In:
            x - data set
            y - training set
            predictions - predicted linear reg model
            xLoc - The location of the desired interval
        Returns: \

        This creates a scatter plot showing the linear regression line,
        as well as the interval the points could appear in at the given point.
    """
    def interval(self, x, y, predictions, xLoc):
```

To accompany the functions, there is also a scatterplot with the line to exemplify the usage of the linear regression function:

[7]:
```
#Linear reggression line
predictions = model.predict(x)
plt.scatter(x = x, y = y, color='orange')
plt.plot(predictions, color='orange')
plt.show()
```



Logistic Regression:

Corresponding to linear regression in a very similar fashion. Logistic regression does the exact same thing, but in a slightly different manner. Logistic regression is used to model a binary statistic (Something like pass or fail, win or loss, iris-setosa or iris-versicolor) but insteat of a linear delimiter, it is logarithmic in fashion, this can be seen in things like S-curves. This algorithm requires the use of 7 functions:

```
    """
        In:
            lr - learning rate
            niter - Number of iterations
            fitIntercept - Boolean for if the algorithm needs to add more
                                intercepts to the data
        Returns: \

        Class object initialization
    """
    def __init__(self, lr=0.01, niter=100000, fitIntercept=True):
```

```
    """
        In:
            X - data set
        Returns:
            array - Data set with an added X value
    """
    def addIntercept(self, X):
```

```
    """
        In:
            z - angle
        Returns:
            sigmoid - based off the input angle
    """
    def sigmoid(self, z):
```

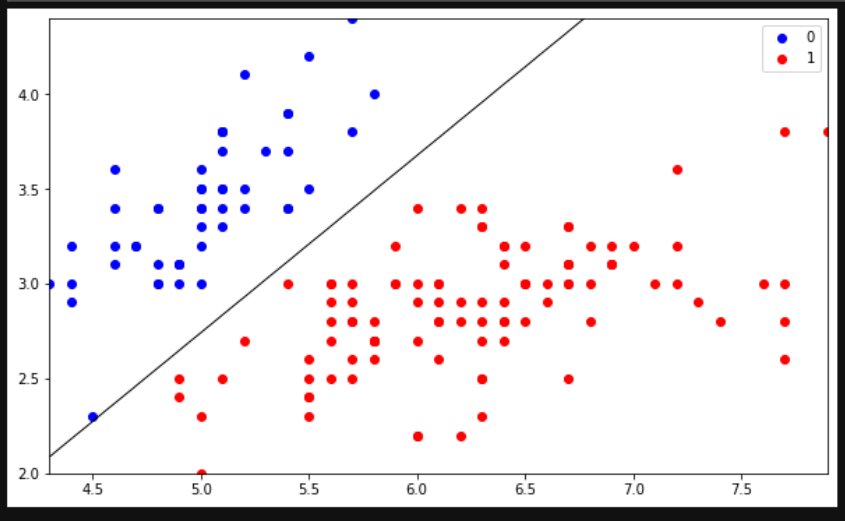
```
    """
        In:
            h - Sigmoid
            y - Training set
        Returns:
            int - loss value between the two inputs
    """
    def loss(self, h, y):
```

```
    """
        In:
            X - Data set
            y - Training set
        Returns: \
```

```
    This takes the data set, and applies a logistic regression line
    to the set based on the training set
    """
    def fit(self, X, y):
    ____________________________________________________________________
    """
        In:
            X - data set
        Returns:
            int - dot product between X and Theta
    """
    def predict_prob(self, X):
    ____________________________________________________________________
    """
        In:
            X - data set
        Returns:
            int - rounded prediction
    """
    def predict(self, X):
```

These functions can be used to create a logistic regression line as seen below:

```
[13]: #Logistic Model
      logModel = logReg(lr=0.1, niter=300000)
      logModel.fit(X, y)
      preds = logModel.predict(X)

[14]: plt.figure(figsize=(10, 6))
      plt.scatter(X[y == 0][:, 0], X[y == 0][:, 1], color='b', label='0')
      plt.scatter(X[y == 1][:, 0], X[y == 1][:, 1], color='r', label='1')
      plt.legend()
      x1_min, x1_max = X[:,0].min(), X[:,0].max(),
      x2_min, x2_max = X[:,1].min(), X[:,1].max(),
      xx1, xx2 = np.meshgrid(np.linspace(x1_min, x1_max), np.linspace(x2_min, x2_max))
      grid = np.c_[xx1.ravel(), xx2.ravel()]
      probs = logModel.predict_prob(grid).reshape(xx1.shape)
      plt.contour(xx1, xx2, probs, [0.5], linewidths=1, colors='black');
```



Hypothesis Classes for Low VC Dimensions

Threshold Function:

The threshold function takes in the data set, training set, as well as the linear regression prediction model. This allows the algorithm to create a binary threshold where the data is split into two classification categories. This is in a single function:

```
"""
   In:
      x - Data set
      y - training set
      pred - predictions from the linear regression model
   Returns:
      array - classified points from x

   Using the linear regression as a threshold, creates a binary classified array
"""
def thresholdFunc(x, y, pred):
```

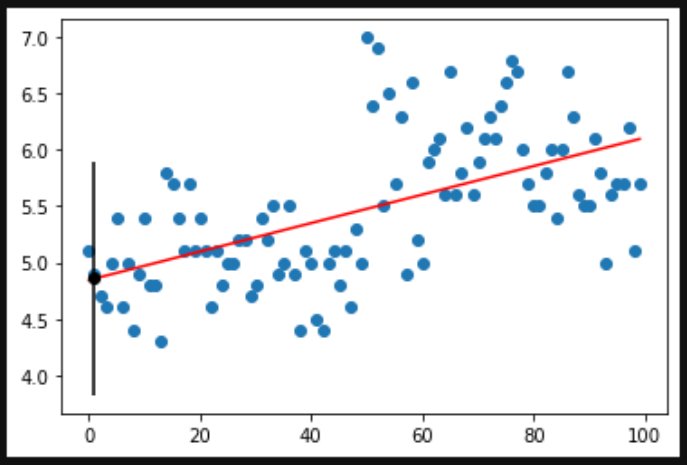
Intervals:

This is a single function hiding in the linear regression class. It uses the data from the linear regression to calculate the error it may have at any given point. By inputting the data, training set, linear regression model, and the x value you would like to see the interval at, it will output a scatterplot showing the data, linear regression line, and the interval of error at the given point:

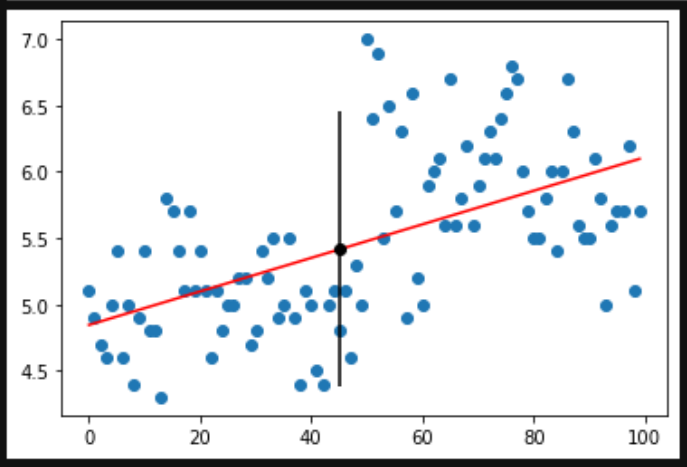
```
   """
      In:
         x - data set
         y - training set
         predictions - predicted linear reg model
         xLoc - The location of the desired interval
      Returns: \

      This creates a scatter plot showing the linear regression line, as well as
      the interval the points could appear in at the given point.
   """
   def interval(self, x, y, predictions, xLoc):
```

The example graphs are seen below, make note of the integer value on the right-hand side of each function call, and the position of the error bar on the graph.

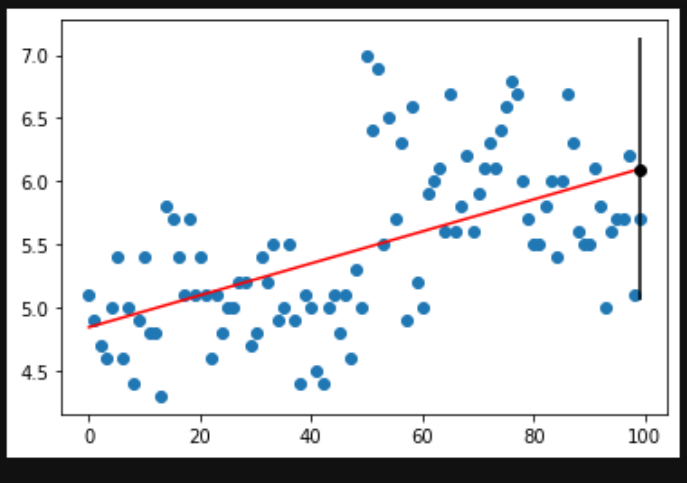
[9]:
```
#intervals
model.interval(x, y, predictions, 1)
```



[10]:
```
model.interval(x, y, predictions, 45)
```



[11]:
```
model.interval(x, y, predictions, 99)
```



Support Vector Machine

Hard SVM:

This algorithm is used to find the largest area between two parallel lines that exist between two classifications of points. This is done so that the visual difference between two categories is as large as possible. This also helps classify new points as they are added, since they can fall is one margin or the other helping predict those new points with little effort. This is done through 4 functions:

```
    """
        In:
            visualization - Boolean for plotting the graph or not
        Returns: \

        Class object initialization
    """
    def __init__(self,visualization=True):
```

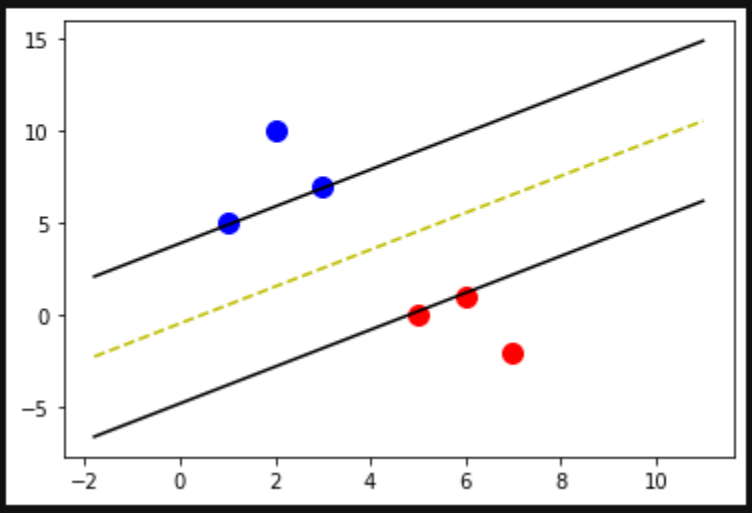
```
    """
        In:
            data - array of points
        Returns: \

        This function fits a linear vector to set of points
        and finds the margins between that vector and the points
        themselves
    """
    def fit(self, data):
```

```
    """
        In:
            features - array of points to be predicted
        Returns:
            int - Predicted classification of points
    """
    def predict(self, features):
```

```
    """
        In: \
        Returns: \

        This function visualizes the SVM on a graph in
        the lab book
    """
    def visualize(self):
```

Example of the hard-margin SVM is below:

```
[18]: #Soft SVM
      data = {-1:np.array([[1,5],[2,10],[3,7]]),1:np.array([[5,0],[6,1],[7,-2]])}
      svm = SVM() # Linear Kernel
      svm.fit(data=data)
      svm.visualize()
```



K-Nearest Neighbor

K-NN:

This algorithm is another classification algorithm. It takes in the data set, training data, the point in the data set you want to classify, and how many neighbors to look at. The algorithm will find the distance between the queried point and it's given number of neighbors, figure out which classification of neighbors it is closest too, and classify the queried point the same way. This is done through two functions:

```
"""
    In:
        A - a point
        B - a point
    Returns:
        int - The distance between the points
"""
def distance(A, B):
```

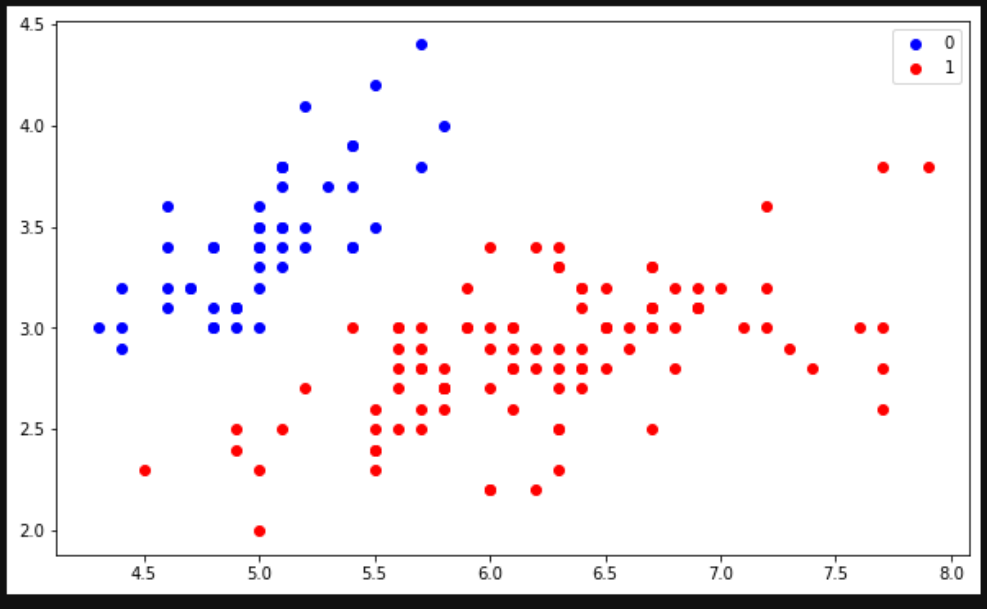
```
"""
    In:
        X - Data set
        y - Training set
        xQuery - Point in X that we want to classify
        k - The number of neighbors we want to look at

    Returns:
        int - The suggested classification based on the nearby neighbors
            (This will be a value that is present in y)
"""
def kNN (X, y, xQuery, k=3):
```

A graphed example is below. Note the bottom left most point (~4.5, ~2.35) is classified as blue in the actual data set, however it's nearest neighbors were red, so that's how it ended up being classified using K-NN.

[16]:
```
#Here we use kNN as a classifier.
#It finds the distance to the nearest points
#then which ever points are closest, it will classify them the same as that
#based on the training data
#As you can see in the scatter plot, the lower left point at (~4.5, ~2.35)
#was previously blue in the logReg example, but is now red based on
#kNN's classification
pred = []
for i in range(150):
    temp = ml.kNN(X, y, X[i], k=5)
    pred.append(temp)
pred = np.array(pred)
```

[17]:
```
plt.figure(figsize=(10, 6))
plt.scatter(X[pred == 0][:, 0], X[pred == 0][:, 1], color='b', label='0')
plt.scatter(X[pred == 1][:, 0], X[pred == 1][:, 1], color='r', label='1')
plt.legend()
```

[17]:
```
<matplotlib.legend.Legend at 0x185f89eaa30>
```

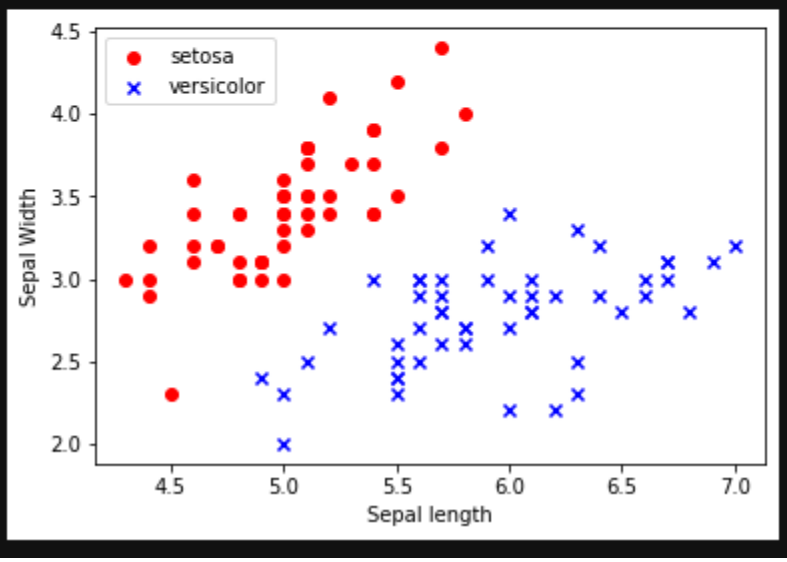


Visualization

Scatter plot:

This is a simple matplotlib set of commands in JupyterLab, so there is no external code needed:

```
[2]: #Scatter plot
plt.scatter(X[:50, 0], X[:50, 1], color='red', marker='o',
label='setosa')
plt.scatter(X[50:100, 0], X[50:100, 1], color='blue',
marker='x', label='versicolor')
plt.xlabel('Sepal length')
plt.ylabel('Sepal Width')
plt.legend(loc='upper left')
plt.show()
```



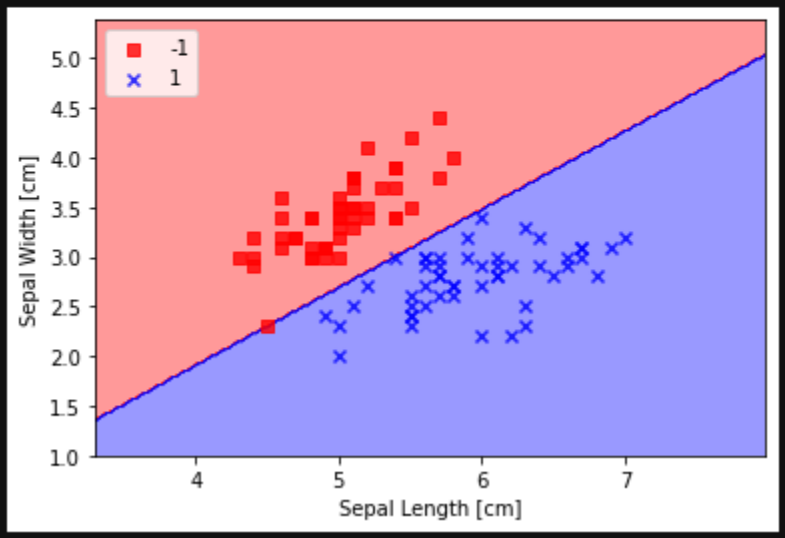
Decision Boundary:

This did require a bit of backend code. The single function takes in the data set, training data, perceptron model, and a variable called resolution. The resolution is how small the steps in the line are, so the smaller the resolution the more like a line it looks. Otherwise this function takes in the data and categorizes it based on how it is classified. This creates a clear boundary from one classification to the other:

```
"""
    In:
        X - Data set
        y - training set
        classifier - perceptron model
        resolution - How clear the line is (smaller number = more clear)
    Returns: \

    This plots the boundry and color codes the points given from the
    perceptron to create a clear visualization of the model
"""
def plot_decision_regions(X, y, classifier, resolution):
```

The example graph is seen below:

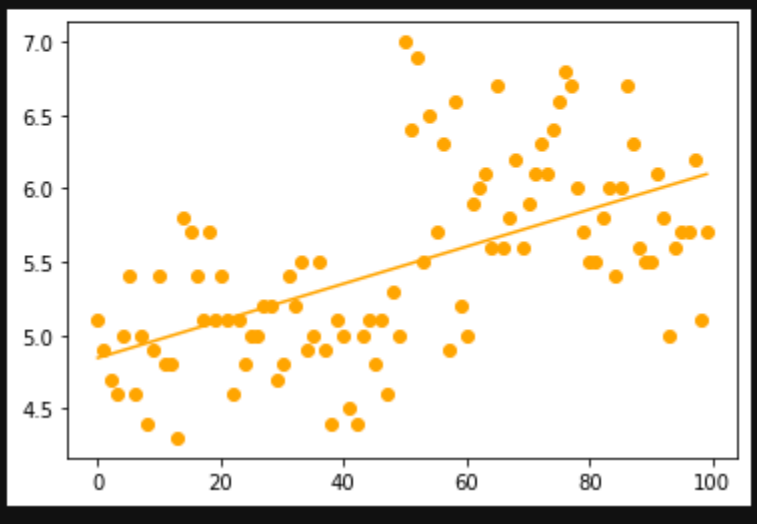
```
[4]: #Decision Boundry
     ml.plot_decision_regions(X, y, pn, 0.02)
     plt.xlabel('Sepal Length [cm]')
     plt.ylabel('Sepal Width [cm]')
     plt.legend(loc='upper left')
     plt.show()
```



Regression Line

This was seen earlier and exemplified in the Linear Regression class, but it will be here for redundancy's sake. Since the linear regression functions: predict() returns a single value. That value is used as the 'm' value in the line equation (y = mx + b). this allows us to easily put the regression line onto a graph using matplotlib's plot() function. This can be seen below:

```
[7]: #Linear reggression line
     predictions = model.predict(x)
     plt.scatter(x = x, y = y, color='orange')
     plt.plot(predictions, color='orange')
     plt.show()
```



ML.py and iris.ipynb are the two files everything is condensed into. They can be found on

https://github.com/WeissTMark/ML