# MACHINE LEARNING

Ethan Steidl

## Table of Contents

# Machine Learning Portfolio

## 1. About the Author

*Image of Ethan Steidl*

The creator of this machine learning portfolio is Ethan Steidl. Ethan is a senior undergraduate Computer Scientist at the South Dakota School of Mines and Technology in Rapid City, South Dakota. His areas of expertise include assembly language development and reverse engineering of binaries. He has one year of experience in both these fields and enjoys them dearly. In the field of machine learning, he is a novice and looking to expand his expertise. Upon graduation in spring of 2021, he will begin work full time working as a Systems Vulnerability Engineer for a technology company on the east coast. Further into his career, Ethan would like to work in instruction set design for processors in the field of performance computing.

## 2. Abstract

The project documented in this portfolio is a machine learning library for Python. Currently the library features seven hypotheses classes. There exists a Perceptron, Linear Regression, and Axis Aligned Rectangle, Decision Stump, Logistic Regression, Hard Support Vector Machine, and K Nearest Neighbor functional hypotheses classes in the library. The classes available in this library are functional for any dimension of data unless otherwise stated. When utilizing the graphing components of the library, only two-dimensional data may be used unless otherwise stated.

## 3. Document Conventions

This document will consist of three text types. The first type consists of headers and titles. These are larger than other text and are denoted in a blue color. The second type is standard text. Standard text makes up the majority of the document and is meant to give insight to the reader. Standard text is in Times New Roman font and displayed at 12pt. The third type of text is programming example text. These sections have their text colored to help the reader understand what is happening in a section of code. These sections are meant as an example of how to implement the library documented by this portfolio.

## 3. Learning Models

A machine learning model is an object that has been trained to recognize specific data patterns. Models are trained by providing them an algorithm, data, and expected results. This section will discuss the seven learning models implemented in the python library.

### 3.1 Perceptron

#### 3.1.1 Introduction

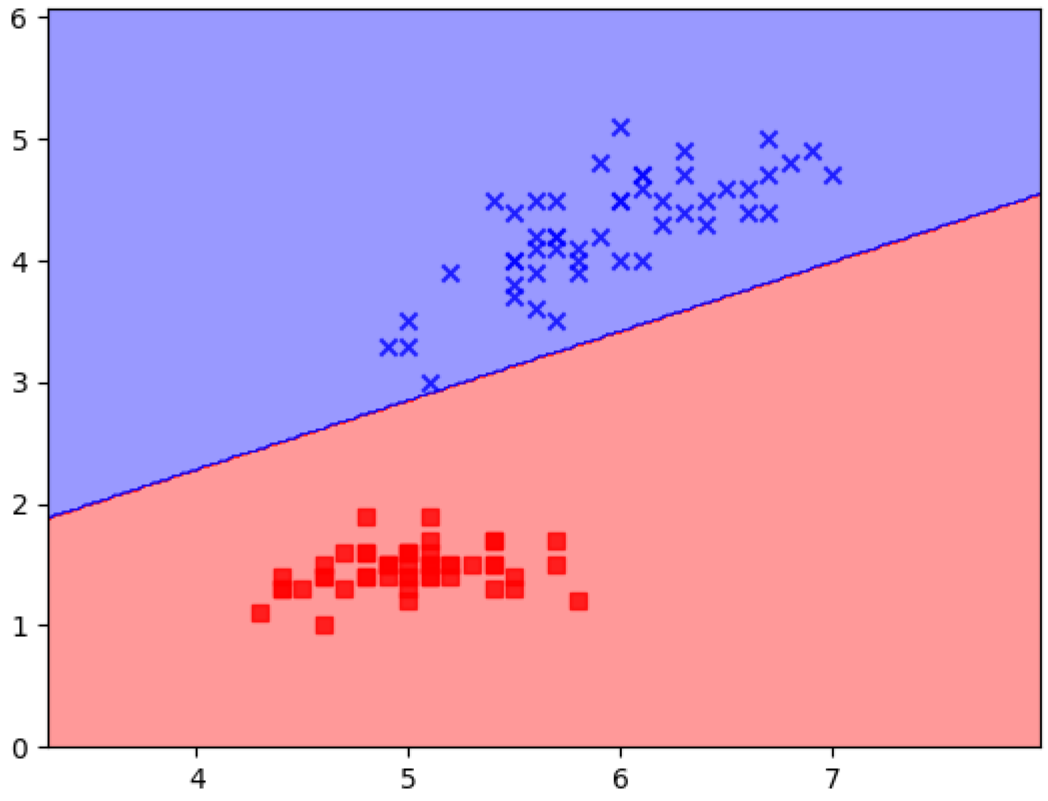
This model allows the user to create a linear binary classifier that can separate a data set into two groups based on a set of expected results.  The class that implements the model has five functions and a constructor.  When using the class, the user passes the model a learning rate and an iteration count.  When fitting the model to the data, the learning rate is used to increase how quickly the model adapts.  The iteration count is the maximum amount of times the model will try to correct itself to be more accurate while learning.  The model also stores internal weights and errors.  During each iteration of fitting, errors are stored in the errors array and weights are updated in the weights array.  If the model is believed to be learned, when the errors produced in an iteration are zero the learning is stopped and returns out back to the calling function.  The perceptron class can also plot data against what it has been trained on.  The plot will separate where the classifier has been trained to separate data.

#### 3.1.2 Usage

The file containing the machine learning library is “ML.py”.  Sample code is found in “main.py” in a function called “perceptron_example”. To use a perceptron, create a perceptron object with a given learning rate and iteration count. To fit the perceptron to data, call the member function “fit” and pass it input data and desired binary output. The input data must come in a vector (X) of vectors (feature lists). The desired output is a vector of -1 or 1 values where 1 represents the given features that should result in a pass classification and -1 represents, they should fail. Below is a code example from “main.py”.

```
import pandas as pd
import numpy as np
from ML import Perceptron
# captures data
df = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-
databases/iris/iris.data', header=None)

# divides up data, y is training set. X is input data
y = df.iloc[0:100, 4].values
y = np.where(y == 'Iris-setosa', -1, 1)

X = df.iloc[0:100, [0, 2]].values

# create a perceptron object
pn = Perceptron(0.1, 10)

# fit the model
pn.fit(X, y)

# check the errors array
print(pn.errors)

# plot the graph
pn.plot_decision_regions(X, y)
```

In this code example, data from a csv is read into a pandas DataFrame. The data comes from the Iris data set at “'https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data”. Then y is set to be the last column of the data. The last column of this data set is a plant species. The first and third column will be used as input data to train the model to. Next the perceptron is created with a learning rate of 0.1 and an iteration count of 10. The model is then fit to the data. Finally, the data is plotted with the model’s prediction on a chart.  An example of the plot output is shown below.



*Perceptron represented graphically.*

In this image, blue points are classified as passes or 1's and red points are failures or -1's. The perceptron learns a line that will break the data into the two groups, pass or fail. The line dividing the middle of the image is the line learned by the algorithm. Everything above the line highlighted in blue is deemed a pass and everything highlighted in red is deemed a fail.

### 3.1.3 Definitions and Algorithms

The perceptron is a function that maps an input **X** to an output value F(**X**). In this case X can be a vector of values and the output of F(**X**) is a single binary value. F(**X**) functions as follows:

$$f(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{w} \cdot \mathbf{x} + b > 0, \\ 0 & \text{otherwise} \end{cases}$$

**W** is a vector of real numbers representing weights. The bias is represented by b. This approach for a learning algorithm has a minor flaw. The perceptron learner will not converge if the learning set is not linearly separable. This means the learner will never reach a point where the data is classified correctly.

Listed below are steps to complete the Fitting algorithm for the perceptron according to Wikipedia.

1. Initialize the weights to 0
2. For each element j in our training set, perform the following steps over the input $\mathbf{X}_j$ and desired output $d_j$
    a. Calculate the actual output:

$$\begin{aligned} y_j(t) &= f[\mathbf{w}(t) \cdot \mathbf{x}_j] \\ &= f[w_0(t)x_{j,0} + w_1(t)x_{j,1} + w_2(t)x_{j,2} + \cdots + w_n(t)x_{j,n}] \end{aligned}$$

    b. Update the weights:

$$w_i(t+1) = w_i(t) + r \cdot (d_j - y_j(t))x_{j,i}, \text{ for all features } 0 \leq i \leq n$$

The steps above are repeated an amount of times specified by the user, known as iterations. Once the learner converges, the training has successfully completed. The algorithm used will allow for early breakout if convergence occurs.

### 3.1.4 Functions

Listed below are the functions used to create the learning model.

fit( X, y):

The fit function takes two arguments X and y. X is a matrix of vectors where each vector contains an element of each feature of the input data. The fit function attempts to use the algorithm listed above to calculate what the output is of the classifier and update the weights, as necessary.

net_input( X ):

Calculates the dot product of the matrix X and the weight vector. Then adds the bias to each element of the result. This new value is then returned

predict( X ):

Given matrix **X** of vectors, creates a temp vector the size of the number of vectors. For each vector in Matrix **X**, if the vector's values represent a pass according to the learned model, the corresponding index in the temp vector is set to 1. If not the value in the temp vector is set to -1. This temp vector is then returned and represents what rows of the matrix **X** pass or fail.

plot_decision_regions( X, y, resolution=0.02 ):

Plots the input data on a graph with a line dividing the two separable sections of data points. The line is derived through the perceptron. This graph will highlight the passes in red and fails in blue. The graphing function only will work on data sets with two features.

## 3.2 Axis Aligned Rectangles

### 3.2.1 Introduction

This model allows the user to create a linear binary classifier that can separate a data set into two groups based on a set of expected results. The class implementing this model has four functions and a constructor. This binary classifier learns a region bounded by two points is considered a pass while everything outside the bounded region is considered a fail. The image below is a visual example of the binary classifier.

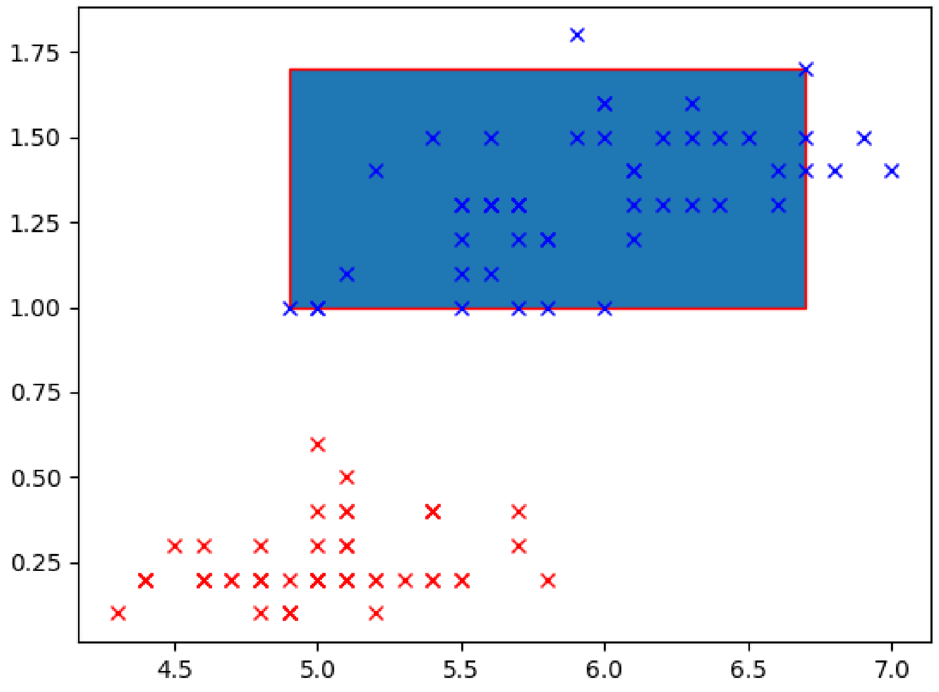


*Image of Axis Aligned Rectangle classifier*

In this image, blue points are expected to pass while red points are expected to fail. Everything inside the blue rectangle is expected to pass. This rectangle is learned by the hypotheses class.

### 3.2.2 Usage

Sample code is found in “main.py” in a function called “axis_aligned_example”. To use an Axis Aligned Rectangle model create an AxisAlignedRectangle object. To fit the model to the data, call the member function “fit” and pass it input data and desired binary output. The input data must come in a vector (**X**) of vectors (feature lists). The desired output is a vector of -1 or 1 values where 1 represent the given features should result in a pass classification and -1 represents, they should fail. Below is a code example from “main.py”.

```
import pandas as pd
import numpy as np
from ML import AxisAlignedRectangle

# captures data
df = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-
databases/iris/iris.data', header=None)

# divides up data, y is training set. X is input data
y = df.iloc[0:100, 4].values
y = np.where(y == 'Iris-setosa', -1 , 1)

X = df.iloc[0:100, [0, 2]].values

#create an axis aligned rectangle
aa = AxisAlignedRectangle()

#fit the model
aa.fit(X,y)

#check dimensions, corners, and accuracy of the model
print(aa.dimensions, aa.corners, aa.accuracy)

#example of predicting values
print(aa.predict(X))

#plotting the graph
aa.plot(X,y)
```

The setup is extremely similar to the perceptron example. The only difference is instead of calling the Perceptron constructor, the AxisAlignedRectangle constructor is called with no parameters. Once created, call “fit” on the object passing in training data as a pandas DataFrame and the desired output vector. Once taken, the model will fit a rectangle to the data with the best chance of classifying correctly. Next the model can be plotted on a chart that looks like the following image.

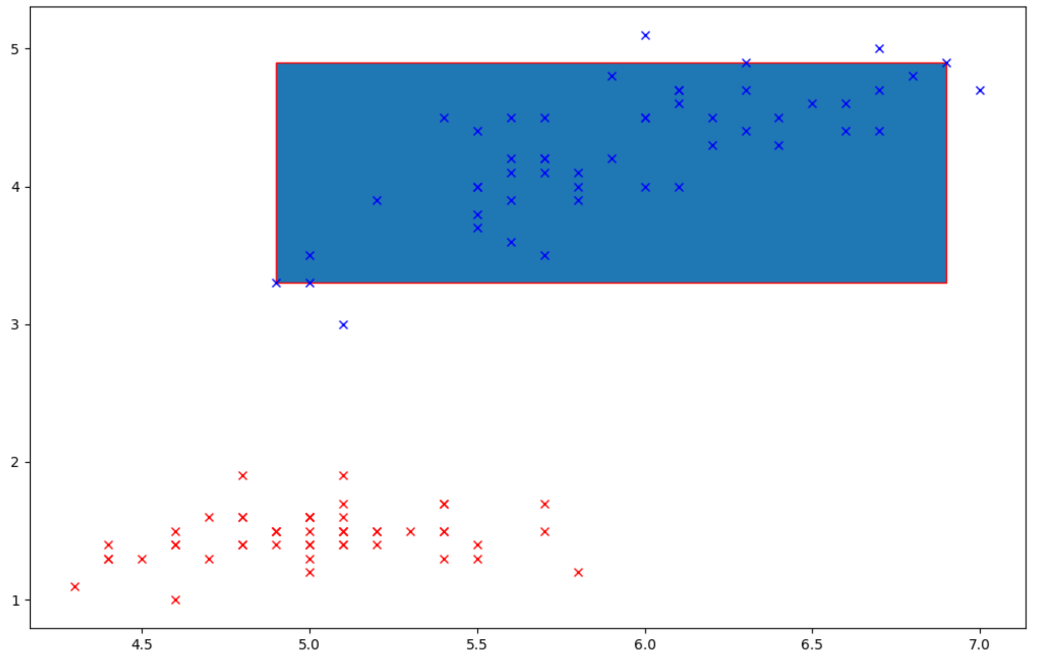


*Image of Axis Aligned Rectangle classifying new data*

In this example image, two-dimensional data is plotted from the Iris data set. The blue x's correspond to instances that should be classified as a pass or 1. The red x values correspond to instances that should be classified as a fail or -1. The fitting algorithm generates a rectangle of the same dimension as the training data to best represent the region of pass values.

### 3.2.3 Definition and Algorithm

The axis aligned rectangle model is a function that maps an input **X** to a binary output value F(**X**). In this case, **X** is a vector of values and the output vector is of the value 1 or -1. The algorithm is performed by creating a D dimensional rectangle with two points as corners where D is the number of features in **X.** The points chosen are a subset of the instances of **X**. Once created, every data instance is checked to see if it resides in the D dimensional rectangle. If the expected value was 1 as in it is expected this point be classified as a true and the instance was inside the rectangle, the value one is added to a score counter. If the expected value was a -1 as in it is expected this instance be classified as a fail and the instance was outside the rectangle, the value one is added to a score counter. If points are on the edge they are considered to be outside the rectangle.  Once every instance has been tested, the final score is divided by the number of instances used.  The score is now a value between 0.0 and 1.0.  This is the accuracy of this specific rectangle on the training data.  The above steps are repeated for every possible rectangle for all points.  The rectangle with the highest accuracy has its accuracy and two boundary points saved for predicting future data.  The training step has a runtime of ( D / 2 ) * ( N^3 – N^2 ) where D is the dimension of the rectangle also known as the feature count and N is the instance count of **X**.  This results in a runtime of O( D * N^3 ) but as D is likely orders of magnitude lower than N, the runtime can simplify to O( N^3 ).

There was a tradeoff made between speed and accuracy for this algorithm.  Speed was chosen over accuracy.  The rectangle used for fitting during the algorithm is always defined by two of the points in the training set. This can lead to a rectangle that will guess wrong when the data is shaped as an L as it will look for the highest scoring region made.  To fix this flaw, the rectangle would have its two defining points calculated from every combination of all values for the corresponding dimension.  For example, if the training points were (1,1,1), (3,6,7), and (2,3,5) the model would fit its two points with all permutations of the three points for each dimension.  In words of runtime, it would take on the order of O( N! ).  Although this answer is more accurate than the one calculated, it will prove impossible to calculate for most data sets.  Therefore, the lower runtime solution was chosen.

### 3.2.4 Functions

fit( X, y):

The fit function takes two arguments X and y. X is a matrix of vectors where each vector contains an element of each feature of the input data. The fit function attempts to use the algorithm listed above to calculate a proper rectangle boundary for the classifier with the highest accuracy calculated by solve_accuracy().

solve_accuracy( p1, p2, X, y ):

This function takes two bounding points of a rectangle, a matrix of vectors where each vector contains an element of each feature of the input data, and the expected result of the classifier as a value of -1 or 1 where 1 is a pass.  The dimensions of X and the two bounding points must be equivalent.  The function will return a decimal number between 0.0 and 1.0

representing the accuracy of the rectangle given.  A point inside the rectangle with an expected result of 1 is a pass.  A point outside the rectangle with an expected result of -1 is a pass.  Everything else is deemed a fail.  The accuracy is calculated by dividing the number of passes by the total instances tested.  This value is then returned.

predict( X ):

This function takes a matrix of vectors where each vector is an instance containing elements of each feature of the input data.  Each instance is checked to see if it is inside or outside of the rectangle created by the classifier.  If the instance is inside it is deemed a pass.  If the instance is outside it is deemed a failure.  A numpy array will be returned of the pass/failure status of each instance where pass is represented by a 1 and failure is represented by a -1.

plot( X, y ):

Plots the input data on a graph with a rectangle displayed in blue representing the bounds of the rectangular classifier.  Instances that are expected to be a 1 or pass are in blue font.  Everything else is in red font.  Below is an example of the output

## 3.3 Linear Regression

### 3.3.1 Introduction

A linear regression is used to approximate a dependent variable given a set of independent variables.  The approach taken in this library was to create a multiple linear regression capable of estimating based on any number of independent variables. A linear regression is a line of best fit that accurately as possible predicts dependent variables from a number of independent variables. Below is an example of a linear regression.

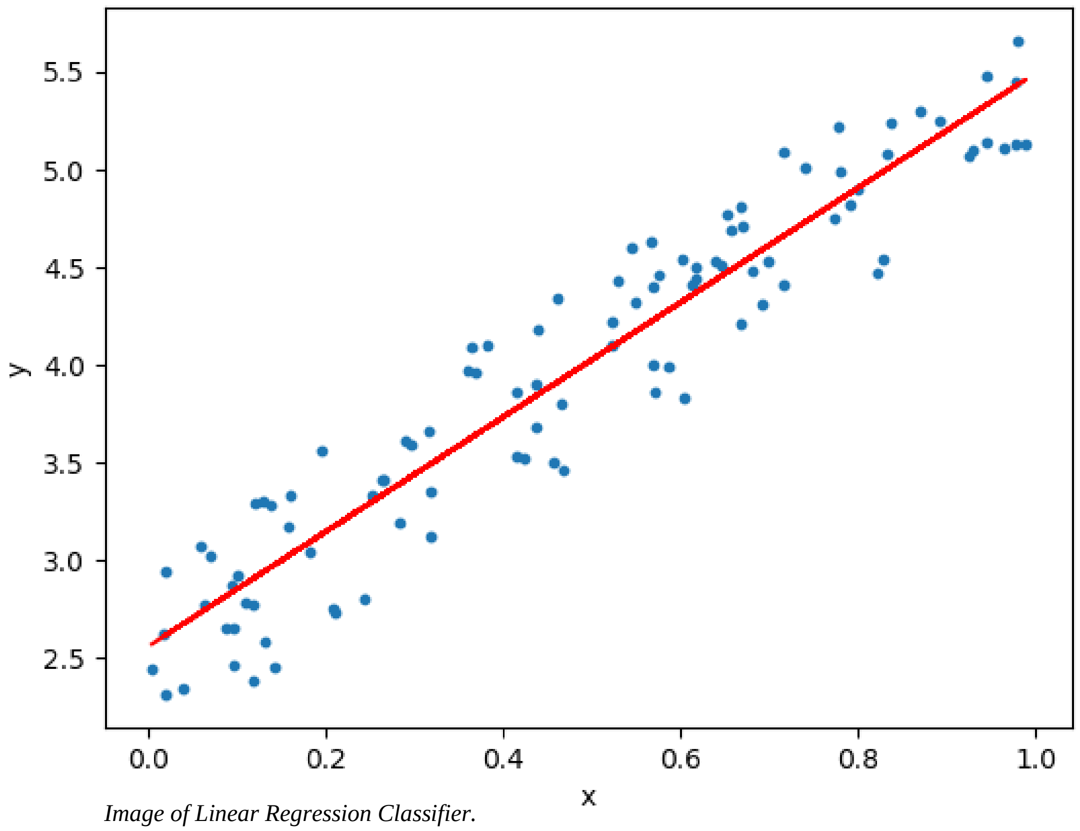


*Image of Linear Regression Classifier.*

The line of best fit for this data set was found and drawn on the data's scatter plot. This line will predict the trend of data and will predict any new data points value. For this model to function accurately the data must be linearly increasing or decreasing.

## 3.3.2 Usage

The file containing the machine learning library is "ML.py". The location of sample code can be found in "main.py" in a function called "linear_regression_example". To use the linear regression, create a LinearRegression object with no parameters. Next call "fit" on the model passing it training data and expected results. This process is similar to that of the perceptron. The training data is a vector of vectors. The outer vector is an instance or row and the inner vector is the list of features. The expected results are a one-dimensional vector consisting of 1 or -1 values for what the expected result of the model to be. Now the model is fit and ready to go.  Call weights to retrieve the weights of the model. The first value in the weights will be the base weight, each additional value will correspond to the features learned in order. The predict function will take input data and return a one-dimensional vector consisting of 1 or -1 values. Below is a code example on using the linear regression model.

```
# captures data
df = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-
databases/iris/iris.data', header=None)

# divides up data, y is training set. X is input data
y = df.iloc[0:100, 4].values
y = np.where(y == 'Iris-setosa', -1, 1)

X = df.iloc[0:100, [0, 2]].values

# create a linearRegression object with threshold
lr = LinearRegression(0.05)

#fit the model
lr.fit(X, y)

#look at weights
print(lr.weights)

#predict some data set
print(lr.predict(X))
```

In this example data was extracted into a DataFrame. This DataFrame was then converted into a numpy array where it was passed into the fitting function of the linear regression. Next the weights are printed out. The user can also use the model to predict future data sets.

### 3.3.3 Definition and Algorithm

When fitting, the weights vector is calculated according to Bremer's linear algebra equation displayed below (Bremer).

$$\hat{\beta} = (\mathbf{X}'\mathbf{X})^{-1}\mathbf{X}'\mathbf{y}$$

Calculating a regression traditionally would take factorial time with respect to the dimension D of the data set. Using the linear algebra calculation above, the order of magnitude drops from O( N! ) to O ( N^3 ) where N is the instance entries in the matrix X. To satisfy the equation above X and y must be in a specific format as defined below.

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}, \qquad \mathbf{X} = \begin{bmatrix} 1 & x_{11} & x_{12} & \cdots & x_{1k} \\ 1 & x_{21} & x_{22} & \cdots & x_{2k} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_{n1} & x_{n2} & \cdots & x_{nk} \end{bmatrix}$$

The linear algebra approach works by finding the best vector **B** such that the sum of squared residuals is minimized. Vector **B** is then stored as the weights vector. When predicting data, the weights are treated as coefficients for a linear equation where the variables are the features of the data the prediction is occurring on.

### 3.3.4 Functions

The linear regression class has two functions and a constructor. The class holds a single variable representing the vector of weights.

fit(X, y):

The fit function takes two arguments **X** and **y**. **X** is a matrix of vectors where each vector contains an element of each feature of the input data. The fit function attempts to use the algorithm listed above to calculate linear equation that solves the expected value. The equation made looks similar to the following. B + w1 * feature1 + w2 * feature2 + … wD * featureD = y. The base is an intercept value. The values stored are the base (B) and weights (w).

predict(X):

This function takes a matrix of vectors where each vector is an instance containing elements of each feature of the input data. Each instance has its features placed into the linear regression equation to calculate a value near the range of -1 to 1. The values are then rounded to -1 or 1. A numpy array will be returned of the pass(1) / failure(-1) status of each instance where pass is represented by a 1 and failure is represented by a -1.

## 3.4 Decision Stump

### 3.4.1 Introduction

The Decision Stump learning model is a single level decision tree. The Decision Stump makes a prediction based off a single input feature. Depicted below is an example decision stump. The model learns an internal value from a single feature of a single data instance. Then when it is asked to predict future data, it looks at a single feature corresponding to its Internal Value and determines whether new the feature is greater or less than itself. If it is true, the right path is taken. If it is false, the left path is taken. In this example, model choses whether it will classify the instance the feature belongs to as class A or class B.

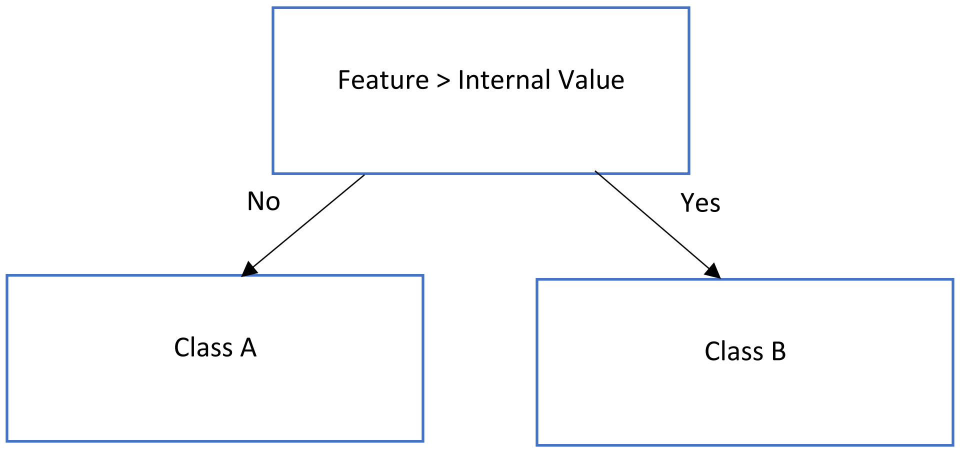


*Image of Decision Stump discriminating between two classes.*

### 3.4.2 Usage

Sample code can be located in the file named "main.py" inside a function called "decision_stump_example". The decision stump must be made by calling the constructor DecisionStump with no parameters. To train the model, call "fit" passing in the training data and the desired binary output of the model as a -1 or 1. This process is similar to that of the perceptron. To create a prediction, call "predict" passing in the input data. The accuracy of the training can be found by calling "accuracy". Below is a piece of example code.

```
from ML import DecisionStump
import pandas as pd
import numpy as np
# captures data
df = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-
databases/iris/iris.data', header=None)

# divides up data, y is training set. X is input data
y = df.iloc[0:100, 4].values
y = np.where(y == 'Iris-setosa', -1, 1)
X = df.iloc[0:100, [0, 2]].values

# create a perceptron object
ds = DecisionStump()

# fit the model
ds.fit(X, y)

#make prediction
print(ds.predict(X))

#check model accuracy
print(ds.accuracy)

# plot the graph
ds.plot(X, y)
```

In this code example, data is read into a DataFrame along with a binary classification vector. Next the decision stump is created. It is then fit to the data and makes new predictions. The accuracy is printed out and the Decision Stump is plotted. Here is an example of the plot that outputs.

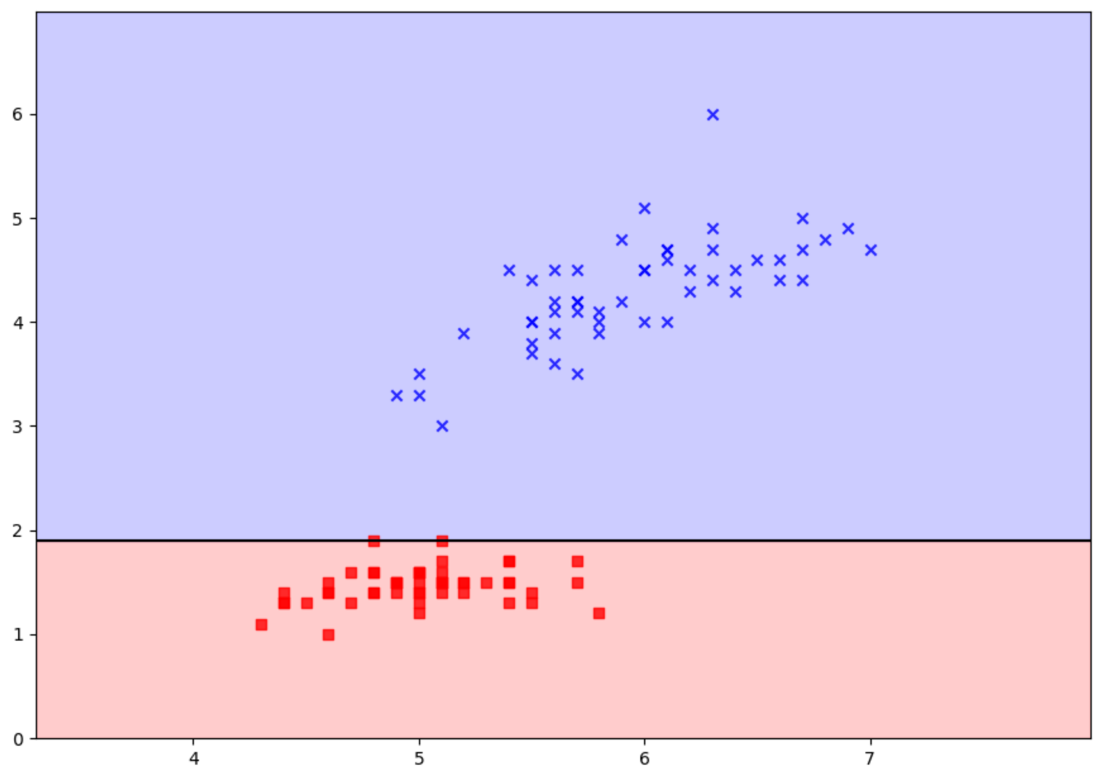


*Image of Decision Stump plot.*

This stump has learned that if it evaluates the feature that is graphed vertically on this plot, it will have the highest odds of deciding the result correctly. The stump value is around 1.95 in this example. Everything above 1.95 is classified as a success or 1. Everything below or equal to 1.95 is classified as a failure. All blue marks are intended to be classified as passes or 1's and all red marks are intended to be classified as failures or -1's.

### 3.4.2 Description and Algorithms

The decision obtains a single value from a single feature column and uses that for predicting future results. This value is found through the following process. For each feature column find the best value. The best value is that which has the highest number of other feature values that are larger than itself that classify as a 1 and has the highest number of other feature values that are smaller than itself that classify as a -1. The accuracy of each values classification is then recorded. Once this process is completed for each column, the value which has the highest accuracy is kept along with the feature column index it came from. When predicting future data, that column of the data is used to compare against.

An optimization was made when accuracy falls below 50%. When this occurs, a variable holding a Boolean representing a flipped state becomes true and the accuracy becomes 1 – accuracy. This minor change adds no time complexity and increases the learning accuracy in some data sets. The time complexity of the learning is O( N^2 * F ) where N is the amount of data instances and F is the feature count. The time complexity of predicting is O( N ) where N is the amount of data instances to predict.

### 3.4.3 Functions

The Decision Stump has three functions. They include predict, fit, and plot.

predict(X):

This function takes a matrix of vectors where each vector is an instance containing elements of each feature of the input data.  The model then compares a specific feature of each instance against the models trained value. If the value is greater than the model's it will classify that instance as a 1 or success. If anything, else, the model will classify that instance as a -1 or failure. The model optimizes its equation and will sometimes compare to values less than its stored value. This has no effect on the outcome result. This function will return a vector of 1 and -1 values.

fit( X, y ):

The fit function takes two arguments **X** and **y**. **X** is a matrix of vectors where each vector contains an element of each feature of the input data. The y parameter is a vector containing the binary classification of each instance as a -1 or 1. The fit function attempts to use the algorithm listed above to calculate a feature column and comparison value to compare prediction data against.

plot( X, y, resolution=0.02 ):

Plots the input data on a graph with a linear line bisecting the data. The line will be of a single dimension that is determined during training.   Instances that are expected to be a 1 or pass are in blue font.  Everything else is in red font.  Areas highlighted in blue will be classified as 1 according to the model. Everything else will be classified as -1.

## 3.5 Logistic Regression

### 3.5.1 Introduction

The Logistic Regression learning model utilizes the sigmoid function for its predicitons. This learning model is a binary classifier that splits data into two different groups. The learning model in its standard form uses the sigmoid function to calculate a probability of success. The sigmoid, given a group of features will calculate a probability between 0.0 and 1.0. If the probability is above 0.5 it can be deemed a pass or 1. Anything else is deemed a failure. Below is an example with two features being classified by the model. The model is similar to that of a linear regression except for its shape.

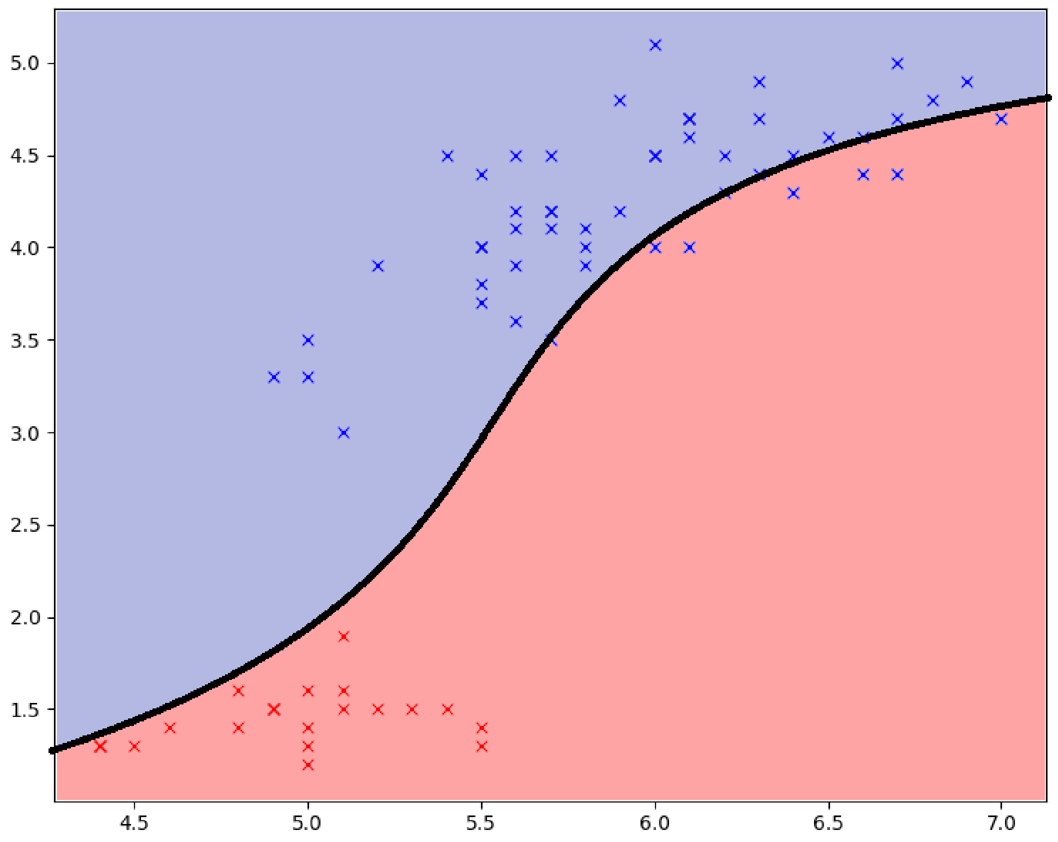


*Image of Logistic Regression sigmoid classification*

### 3.5.2 Usage

The file containing the machine learning library is “ML.py”. Sample code is found in “main.py” in a function called “logistic_regression_example”. To use a logistic regression, create a LogisticRegression object with an learning rate and iteration count. The learning rate is a floating point value between 0.0 and 1.1. The iteration count is an natural number. These paramteters are used in the algorithm described in the next section. To fit the model to data, call the member function “fit” and pass it input data and desired binary output. The input data must come in a vector (X) of vectors (feature lists). The desired output is a vector of -1 or 1 values where 1 represents the given features that should result in a pass classification and -1 represents, they should fail. Below is a code example from “main.py”.

```
import pandas as pd
import numpy as np
from ML import LogisticRegression
# captures data
df = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-
databases/iris/iris.data', header=None)

# divides up data, y is training set. X is input data
y = df.iloc[30:100, 4].values
y = np.where(y == 'Iris-setosa', -1, 1)
X = df.iloc[30:100, [0, 2]].values

# create a perceptron object
lr = LogisticRegression()

# fit the model
lr.fit(X, y)

# print the weights used
print(lr.weights)

#make prediction
print(lr.predict(X))

#check model accuracy
print(lr.accuracy)
```

In this example, data is read into a pandas DataFrame. Next a LogisticRegression object is created. The model is then fit to data and is asked to output the calculated weights. The model then performs a prediction on data and prints its accuracy out to the screen.

### 3.5.3 Description and Algorithms

The method used for this library's logistic regression is based off the work of Abhinav Sagar, a deep learning researcher at VIT Vellore. A logistic regression is modeled after the following equation.

$$\ell = \log_b \frac{p}{1-p} = \beta_0 + \beta_1 x_1 + \beta_2 x_2$$

Where l is the log-odds, p is the probability of success, each B is a learning parameter or weight, and x is the feature input. The equation can be manipulated by exponentiating the log-odds into the following form.

$$p = \frac{1}{1 + b^{-(\beta_0 + \beta_1 x_1 + \beta_2 x_2)}}$$

This equation has remarkable similarity to the sigmoid function.

$$S(x) = \frac{1}{1 + e^{-x}}$$

Using algebraic substitution of the sigmoid function into the previous equation, the following probability function can be constructed.

$$p = S_b(\beta_0 + \beta_1 x_1 + \beta_2 x_2)$$

With the probability function, learning with the model can be done. The steps to having the model learn are as follows

1. Assign all weights equal to zero where the amount of weights the count of features to be learned
2. Calculate the dot product of the feature matrix and weights vector. This will result in a temporary score vector.
3. Calculate the probability vector by applying the above probability function to the score vector

4. Find the output error by finding the difference between the target classifications and the probability vector
5. Calculate the change vector needed for the model to adapt by taking the dot product of the feature's matrix and the output error.
6. Add to the wights the change vector multiplied by the learning rate
7. Repeat steps 1-6 for however many iterations is desired.

With the weights the model has been trained and is ready for prediction.

### 3.5.4 Functions

sigmoid( scores ):

Takes in a one-dimensional vector of scores or weights. The logistic regression sigmoid equation is applied on the weights and the resulting probability equation is returned as a one dimensional vector

logistic_regression( X, y, iterations, learning_rate ):

Performs the logistic regression learning algorithm. Weights are calculated though an iterative process where the features are doted against weights and their success probability is assed by the sigmoid function. Error is calculated and the weights are adjusted based on the deviation between the predicted weights and the features. When adjusting the weights, the learning rate is multiplied against the deviation to increase or decrease learning speed.

fit( X, y):

The fit function takes two arguments X and y.  X is a matrix of vectors where each vector contains an element of each feature of the input data.  The fit function attempts to use the algorithm listed above to calculate what the output is of the classifier and update the weights, as necessary. This function also sets the internal accuracy of the model.

calculate_accuracy( X, y ):

The function takes two arguments X and y.  X is a matrix of vectors where each vector contains an element of each feature of the input data. After a model has been trained, the result of the input data X will be compared against y. The accuracy is calculated by dividing successful classification counts from X by the size of the data set. The accuracy is then recorded.

predict( X ):

Given matrix **X** of vectors, the sigmoid function is applied to the input data **X**. Though testing, when probabilities were greater than 10^-8, they were found to be classified as pass 1. Everything else is classified as a failure or -1. A vector of resulting classifications is returned to the caller.

## 3.6 Hard Support Vector Machine

### 3.6.1 Introduction

This model allows the user to create a linear binary classifier that can separate a data set into two groups based on a set of expected results.  When using the model, the user passes the model a learning rate and an iteration count.  When fitting the model to the data, the learning rate is used to increase how quickly the model adapts.  The iteration count is the maximum amount of times the model will try to correct itself to be more accurate while learning.  The model also stores internal weights and errors.  During each iteration of fitting, errors are stored in the errors array and weights are updated in the weights array.  If the model is believed to be learned, when the errors produced in an iteration are zero the learning is stopped and returns out back to the calling function.  The support vector machine class can also plot data against what it has been trained on.  The plot will separate where the classifier has been trained to separate data. The main difference between the hard support vector machine and a perceptron is the margin space added between the line separating the two data sets. The hard support vector machine will guarantee the same amount of marginal space between the closest data point from each classification set.

### 3.6.2 Usage

The file containing the machine learning library is “ML.py”.  Sample code is found in “main.py” in a function called “Hard_SVM_example”. Using this model is very similar to the perceptron. Data is read into a pandas DataFrame along with its expected classifications. Next a

HardSVM object is created with a learning rate and iteration count. The model is fit and outputs it's errors and a plot of the classifier. The data given to this model must be linearly separable.

```
# captures data
df = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-
databases/iris/iris.data', header=None)

# divides up data, y is training set. X is input data
y = df.iloc[0:100, 4].values
y = np.where(y == 'Iris-setosa', -1, 1)

X = df.iloc[0:100, [0, 2]].values

# create a perceptron object
svm = HardSVM(0.1, 10)

# fit the model
svm.fit(X, y)

# check the errors array
print(svm.errors)

# plot the graph
svm.plot_decision_regions(X, y)
```

### 3.6.3 Description and Algorithm

The support vector machine used follows a similar algorithm to that of the perceptron is a function that maps an input **X** to an output value F(**X**). In this case X can be a vector of values and the output of F(**X**) is a single binary value. F(**X**) functions as follows:

$$f(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{w} \cdot \mathbf{x} + b > 0, \\ 0 & \text{otherwise} \end{cases}$$

**W** is a vector of real numbers representing weights. The bias is represented by b. This approach for a learning algorithm has a minor flaw. The perceptron learner will not converge if the learning set is not linearly separable. This means the learner will never reach a point where the data is classified correctly.

Listed below are steps to complete the Fitting algorithm for the hard support vector machine

1. Initialize the weights to 0
2. For each element j in our training set, perform the following steps over the input $\mathbf{X}_j$ and desired output $d_j$
    a. Calculate the actual output:

$$\begin{aligned} y_j(t) &= f[\mathbf{w}(t) \cdot \mathbf{x}_j] \\ &= f[w_0(t)x_{j,0} + w_1(t)x_{j,1} + w_2(t)x_{j,2} + \cdots + w_n(t)x_{j,n}] \end{aligned}$$

b. Update the weights:

$w_i(t+1) = w_i(t) + r \cdot (d_j - y_j(t))x_{j,i}$, for all features $0 \leq i \leq n$

The steps above are repeated an amount of times specified by the user, known as iterations. Once the learner converges, the training has successfully completed. The algorithm used will allow for early breakout if convergence occurs.

One final step must be completed and that is the hard-margin calculation. A maximum-margin hyperplane must be found for the equation of the line calculated by the weights. This is done by finding the Euclidian distance to the closest point from each data classification set. Once found, the line is adjusted to have the smallest sum of least squares of the distance to each point. The line is adjusted by altering the weights.

### 3.6.4 Functions

Listed below are the functions used to create the learning model.

fit( X, y):

The fit function takes two arguments X and y. X is a matrix of vectors where each vector contains an element of each feature of the input data. The fit function attempts to use the algorithm listed above to calculate what the output is of the classifier and update the weights, as necessary.

net_input( X ):

Calculates the dot product of the matrix X and the weight vector. Then adds the bias to each element of the result. This new value is then returned

predict( X ):

Given matrix **X** of vectors, creates a temp vector the size of the number of vectors. For each vector in Matrix **X**, if the vector's values represent a pass according to the learned model, the corresponding index in the temp vector is set to 1. If not the value in the temp vector is set to -1. This temp vector is then returned and represents what rows of the matrix **X** pass or fail.

plot_decision_regions( X, y, resolution=0.02 ):

Plots the input data on a graph with a line dividing the two separable sections of data points. The line is derived through the perceptron. This graph will highlight the passes in red and fails in blue. The graphing function only will work on data sets with two features.

## 3.7 K Nearest Neighbor

### 3.7.1 Introduction

The K-NN or K Nearest Neighbor classifier categorizes input data by what the classification is of nearby data points. The classifier will look for K instances near a given point. It will then find the most common classification between those points and classify the new point as that most common classification. Below is an example of what the classifier does.

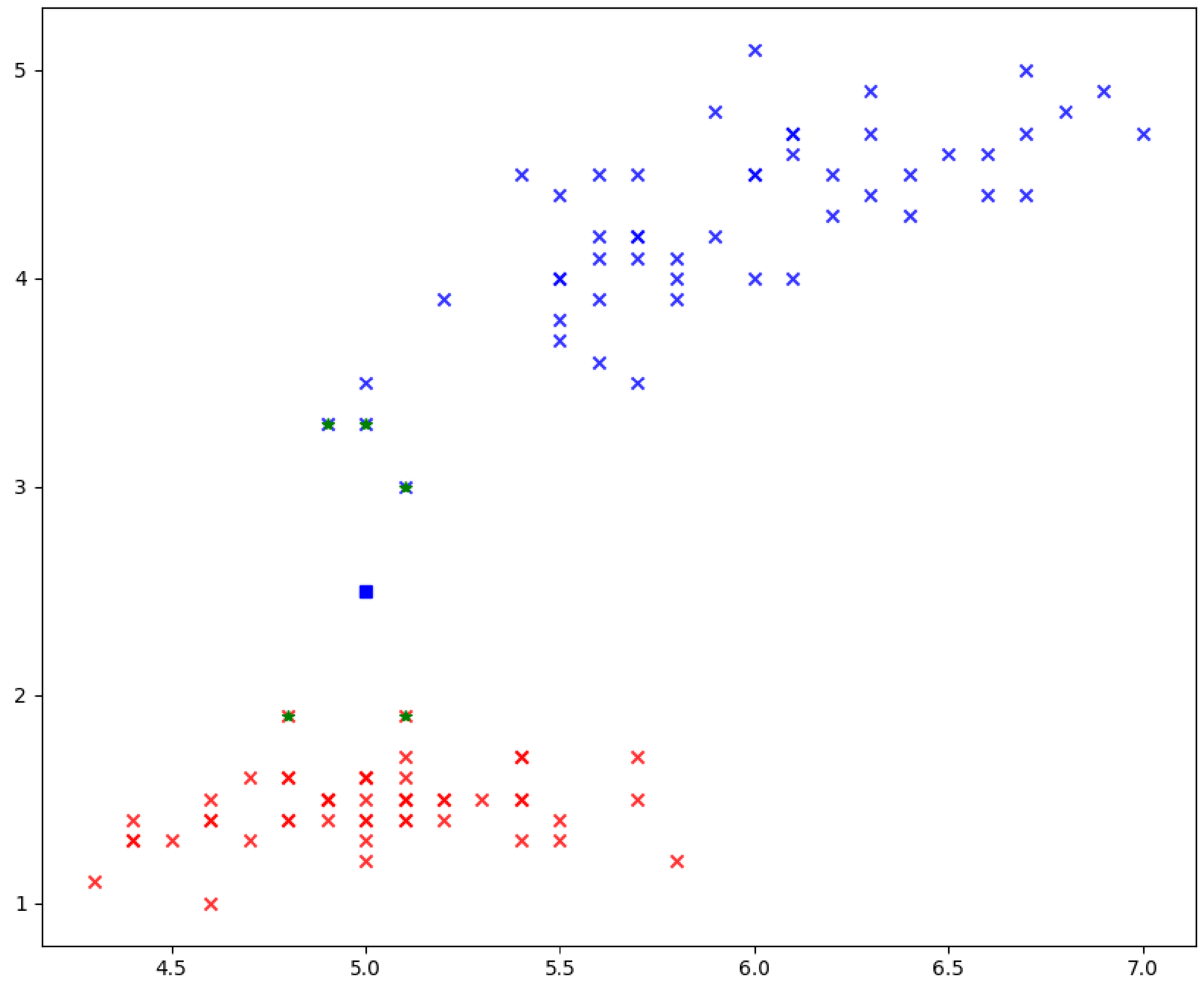


*Image of K Nearest Neighbors classifying K=5 points for a new point, the square*

The model in this figure has been trained on data that consisted of two different classifications. The different classifications can be seen as red or blue x's on the image. The square is a new piece of data that we would like to classify with the K-NN classifier. Using a K value of five, the five nearest points were found. The five points used are denoted with a green asterisk in them. The total of each type of classification was found. That classification was given to the new point, the square. This new points classification is denoted by its color.

### 3.7.2 Usage

The file containing sample code is "main.py" in a function called "KNN_example". To use the K Nearest Neighbor hypotheses class, create a KNN object with no parameters. Next call "fit" on the model passing it training data and expected results. The training data is a vector of vectors. The outer vector is a vector of instances or rows and the inner vectors are the list of features. Now the model is fit and ready to be used. The user can call predict on a set of data and the model will return a vector of classifications from the data. Call plot with a single instance of data to show a plot similar to that shown above. The plot will display the data the model was trained on, the points selected for the KNN algorithm and what the data instance was classified as. Both the predict and plot take an optional K parameter. This value is by default 1 and must be between 1 and the count of instances the model was trained on inclusively. Below is a code example on using the K Nearest Neighbor model.

```
import pandas as pd
import numpy as np
from ML import KNN
# captures data
df = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-
databases/iris/iris.data', header=None)

# divides up data, y is training set. X is input data
y = df.iloc[0:100, 4].values
y = np.where(y == 'Iris-setosa', -1, 1)
X = df.iloc[0:100, [0, 2]].values

#the prediction set is what the model will predict the classifier of
predictionSet = df.iloc[45:59, [0, 2]].values

#the plot value is what the model will try to find the nearest neighbors of
#This is an arbitrary new point to run against the model
plotValue = np.asarray([5.0, 2.5])

# create a perceptron object
knn = KNN()

# fit the model
knn.fit(X, y)

#shows the predictions the model gave
print(knn.predict(predictionSet, k=4))

# plot the graph
knn.plot(X, y, plotValue, k=5 )
```

In this example, data is read in to a pandas DataFrame. The training data is divided up from the input data. Next a subset of data is selected from the training data to test the model's predictions. A new value is then created to be tested against the model's predictions and to be used in the plot. A KNN model is then created and fit to the data. Next the model runs predictions on the input data with a K=4 and prints out its predictions. Following this, a plot is show where the model is classifying the new value created with a K=5.

### 3.7.3 Definition and Algorithm

KNN or K nearest neighbors is a machine learning classification algorithm. The model is trained purely by giving it data and expected classifications. It takes these values and stores them internally. When asked to predict a new data point, the model will look for the k nearest neighbors and assign classification based on the most popular classification of the k nearest neighbors.

The time complexity of training is O( N * D ) where N is the instances of data and D is the dimension of the data or feature count. The time complexity of predicting is O( N * D * K ) where K is the amount of closest neighbors desired. The method used in this library checks for the closest neighbor and removes that from the set. Then searches again for the nearest neighbor. This can be optimally calculated in O ( D * N log2 N + K ) time if sorting points before searching is done. The larger asymptotic runtime method was chosen for this problem because K is likely less than log2 of the sample size. The algorithm could be optimized further by choosing one or the other method based on the K value.

### 3.7.4 Functions

The KNN class utilizes five functions. The class stores two pieces of data. It stores the training data and the expected result of the training data. Private functions of this class are prefixed with a single underscore.

_predict_single( X ):

The purpose of this function is to find the nerest neighbor to a single instance of data. X is a vector instance of features. This function calculates the Euclidian distance between the X vector and every instance of data the model was trained on. When the shortest distance is found, a tuple containing the closest instance's classification, index in the data set, and feature list is returned.

predict( X, k=1 ):

Given a list of instances, a vector of -1 and 1 classifier values will be created and returned to the user. The index of the classifier value in the vector returned corresponds to the index of data given to the model. Values of 1 indicate the nearest neighbors were likely classified as a 1. While values of -1 indicate the neighbors were more likely to be classified as a -1.

fit( X, y ):

Trains the model on the dataset X and the expected classifier y. X is a vector of vectors where the inner vector is the set of features of the data. The vector y is the expected classification for each instance in vector X.

knn_group( X, k ):

X is a single instance of features. K is the number of desired neighbors to find. This function will calculate the K closest neighbors and return them as a list.

plot( X, y, v, k=1):

Given the set of input data X and the expected classifications y, a plot will be made to denote which k neighbors are selected for classification and what the classification of v was. The v parameter is a single instance of features.

## 4. Testing:

Testing was done on varying amounts of feature lengths. Every algorithm was tested using the Iris data set found at https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data. While the perceptron fitting function does not always converge, the algorithm will work on any feature length greater than one. Tested on Sepal Length against petal length Identifying Iris-senotas, this set works. The algorithms were tested on Sepal Width against petal length Identifying Iris-senotas, this also works. When testing on Sepal Length against Sepal width Identifying Iris-senotas, it does not work. I am not sure why the line does not draw correctly but it is likely caused by a bug in the perceptron. Other tests were conducted on different flower types as well and found the perceptron was working in most cases. The cases where the perceptron did not work were those whose data did not look linearly separable. The linear regression and axis aligned rectangles were also tested using the same data. The axis aligned rectangles has problems when the data deemed a pass is in the shape of an L. This is because a lower left and upper right point are chosen but since there are no points in the upper left to choose from, the algorithm does the best it can. The k nearest neighbors were tested five points that exist on the inside of the data set and five points that did not exist in the dataset. The k value was only tested for natural numbers larger than zero and smaller than the training data set. The logistic regression, support vector machine, and decision stump were also tested with the Iris data set. The decision stump and the support vector machine were found to always function properly. The logistic regression has issues with overfitting even when the iteration count is small. This issue becomes noticeable when the proportion of the two classifications is not around 1 : 3. This value was found though testing ranges of data.

## 5. Submission

The code submitted is in ML.py and main.py. The library, ML.py was broken into subfiles for each classifier. They are axisAlignedRectangle.py, linearRegression.py, perceptron.py, decisionStump.py, HardSVM.py, KNN.py, and LogisticRegression.py. ML.py contains imports to act as the main library file. Main.py contains sample code on how to use each of the learning models. Additionally, the Iris data set as a csv file is added along with this package in case the user does not have access to the online data set. The program was tested using python 3.6 in PyCharm. To run his program from windows command line, type "python main.py" from the directory the code is placed in. The writeup is in "Portfolio.pdf".

## 6. Works Used

Bremer, M. 2012. *Multiple Linear Regression*. PDF, https://drive.google.com/file/d/1FcpgVugKkqdB2RpXFyDPo9YE3jIWDH0/view?usp=sharing

Sagar, Abhinav. "How to Build Your Own Logistic Regression Model in Python." *KDnuggets*, 2020, www.kdnuggets.com/2019/10/build-logistic-regression-model-python.html.

Shalev-Shwartz, Shai, and Shai Ben-David. 2019. *Understanding Machine Learning: from Theory to Algorithms*. Cambridge University Press.

Wikipedia. "Decision Stump." *Wikipedia*, Wikimedia Foundation, 23 Dec. 2019, en.wikipedia.org/wiki/Decision_stump.

Wikipedia. "Logistic Regression." *Wikipedia*, Wikimedia Foundation, 30 Nov. 2020, en.wikipedia.org/wiki/Logistic_regression.

Wikipedia. “Perceptron.” *Wikipedia*, Wikimedia Foundation, 21 Sept. 2020, en.wikipedia.org/wiki/Perceptron.

Wikipedia. “Support Vector Machine.” *Wikipedia*, Wikimedia Foundation, 28 Nov. 2020, en.wikipedia.org/wiki/Support_vector_machine.