Supervised Machine Learning Library

Author: Vedang D Gaonkar

Date Updated: Dec 07, 2020

1.0 Overview

1.1 Purpose

This library exclusively implements and tests different data classifier algorithms under the umbrella of supervised machine learning. Supervised machine learning involves observing a dataset, usually stored in a vector to predict results for a given section of the same data. Most models are first trained with the given data to predict expected output. Supervised learning can be classified into two types: Classification and Regression. In classification, you observe the data and classify the unobserved data into different categories, often binary. Whereas in regression, the data is regressively run through to give the closest numerical metrics possible. We have implemented seven classification and regression algorithms in this library. You can find detailed descriptions of these in their respective sections (4.0 to 10.0).

1.2 Project Location

The necessary code and examples should be enclosed in the zip file where this document was found. If this document is being viewed independently, find the code here: https://github.com/vedangGaonkar/Machine_Learning_Library

1.3 Datasets Used

For the examples and scope of this document, we have specifically observed and tested on the iris flower dataset. This dataset is easy to understand and test, especially for beginners. It also has a lot of resources available to cross-check your results, since this is probably the most tested dataset in similar machine learning projects. The library was also tested on the below mentioned datasets.

1. Iris Flower Dataset - https://en.wikipedia.org/wiki/Iris_flower_data_set
2. NY Subway Passengers and Weather Dataset
3. Perfume Dataset - https://archive.ics.uci.edu/ml/datasets/Perfume+Data
4. Titanic Passengers Dataset - https://www.kaggle.com/c/titanic

For the next version release of this library, we will exclusively observe the titanic dataset to see who's fit to survive and who isn't! This is going to be lit.

2.0 Project Structure

This library currently contains the following files:

- ML.py

  This is our library. Right now, there are working implementations of seven different classification and regression models.

  1. Perceptron (section 4.0)
  2. Linear Regression (section 5.0)
  3. Logistic Regression (section 6.0)
  4. Decision Stumps (section 7.0)
  5. Prediction Intervals (section 8.0)
  6. Support Vector Machine (section 9.0)
  7. K-Nearest Neighbors (section 10.0)

- main.py

  As the name suggests, this contains the main function. It contains the code to let the user choose which model they want to run and makes the necessary calls accordingly.

- perceptron_examples.py (section 4.2)
- linear_regression_examples.py (section 5.2)
- logistic_regression_examples.py (section 6.2)
- decision_stumps_examples.py (section 7.2)
- prediction_intervals_examples.py (section 8.2)
- svm_examples.py (section 9.2)
- knn_examples.py (section 10.2)

3.0 Usage

3.1 Requirements

- Interpreter: Python 3.7 and above
- If you're using PyCharm - PyCharm 2019.2
- All the python files listed in section 2.0 should be in the root directory
- pip to install the necessary packages

3.2 Running

- Running in PyCharm or other IDE: Right click in main.py and hit Run 'main'. You might need to download and configure the interpreter.
- Running in a linux terminal: Run the command *Python3 main.py*

3.3 Input

After running main, you will see a menu to run the model of your choice.

```
Choose an option from the following:
1 - Run Perceptron
2 - Run Linear Regression
3 - Run Logistic Regression
4 - Run Decision Stumps
5 - Run Prediction Intervals
6 - Run Support Vector Machine
7 - Run K-Nearest Neighbors
0 - Quit
```

Enter the number corresponding to the model you want to run. Currently, the program runs the examples automatically and the user does not have to worry about any more inputs.

The expected results and examples for each model are specified in their respective sections.

4.0 Perceptron

4.1 Algorithm

Perceptron is a classifier algorithm used to divide the datasets into a binary classification for the given attributes. The objective of this algorithm is to draw a straight line that most satisfactorily divides the data into two groups. For this, we need training data with dependent and independent vectors. The independent variable will contain two of the attributes of the data to be compared against each other on the cartesian axes. Using our training data, we determine the coefficients (w) of the data. These coefficients basically dictate how the line is going to be for the perfect division. We also look at the errors generated in this training data and loop through our independent as well as dependent variables to adjust the coefficients accordingly to give as good a classification as possible. This algorithm has four functions: init(), fit(), net_input(), predict(). In this algorithm, the predictions for each datapoint are basically called through the fit function that first trains the data and then adjusts the coefficients. This is also called the step function. If you are familiar with linear algebra, the list of coefficientare used in their original form for one class while the other uses the transpose of each ‘w’. This results in the current coefficient of the x value to have a ‘1’ or ‘-1’ value which determines if that particular point is going to lie above or below the line. This way, we have our classification.

In our examples, we used attributes of two kinds of iris flowers each time against each other to classify them into separate categories.

4.2 Examples

I tested my perceptron on 3 different combinations of flowers and features from the Iris flower dataset. The tests and the expected outcomes are as follows. The first two examples also show miscalculations graphs.

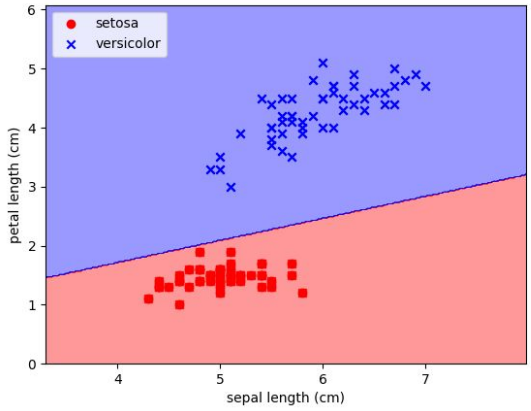
Note: Close each graph so that the next one pops up.

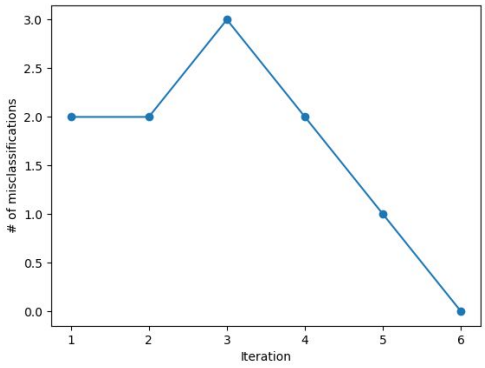
Test 1

Flowers Used: Setosa, Versicolor

Features Used: Sepal length, Petal Length

Plots:





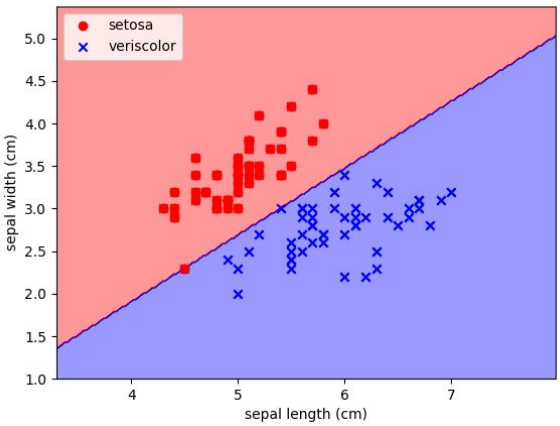
:

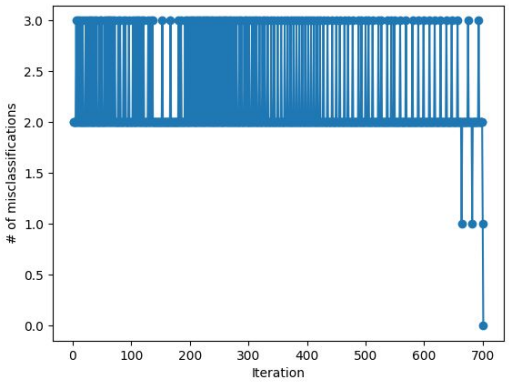
Test 2

Flowers Used: Setosa, Versicolor

Features Used: Sepal length, Sepal width

Plots:



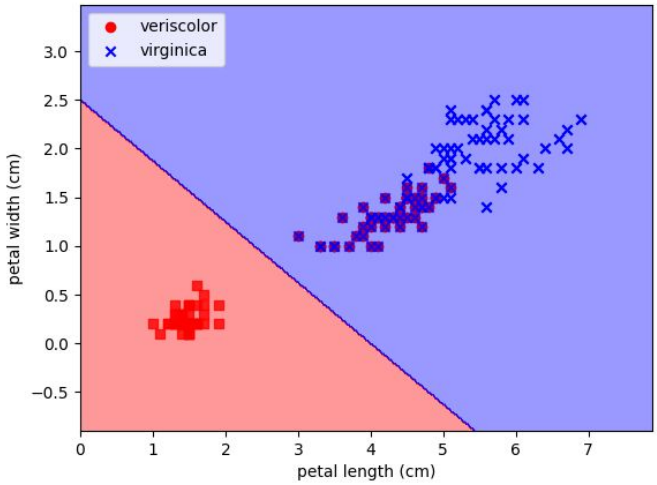


Test 3

Flowers Used: Versicolor, Virginica

Features Used: Petal length, Petal width

Plot:



5.0 Linear Regression

5.1 Algorithm

Linear regression is called the Hello world of the Machine Learning industry. Unlike the perceptron, linear regression regresses as the name suggests (actually don't be deceived by the names, you'll see why in section 6.0). As usual, we have training data that we use to generate a line of the form 'y = mx+c',. A major key element is that regression is best done on continuous data, whereas classification can work amazingly well on fragmented or discrete data. As you can see, this is a linear approach to define a relationship between independent or explanatory data and dependent variables or scalar responses.

Our linear regression algorithm is defined by two methods - fit and predict, used for training and predicting respectively. In fit, we first check the shape or the dimension of the explanatory data. For the scope of this algorithm, we are only training over d=1 data. The coefficient or the weight is stored in a matrix which currently uses only 1 spot. In short, fit trains the independent variable to be predicted according to the behavior of the explanatory variable. We use the partial differential equation of the summation of data points with respect to the weight or 'm' and also with respect to the intercept to predict. You can read this for more explanation on the partial differential equations.

$$\frac{\partial cost}{\partial m} = 2/n \sum_{i}^{n} (mX + C - ytrue) * X \qquad \frac{\partial cost}{\partial C} = 2/n \sum_{i}^{n} (mX + C - ytrue) * 1$$

If the fit results are provisioned to store somewhere, the fit function keeps getting better in determining the regression line.

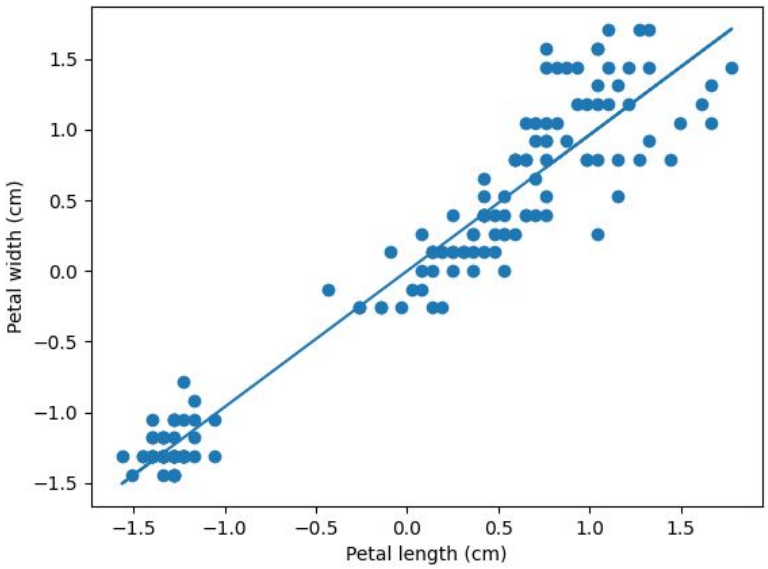
5.2 Examples

In our example, we are looking at similar representations as to section 4.2, only difference is that these are not classified into two classes but will only be used to determine the splitting line which is the main purpose of our algorithm.

Note: Close each graph for the next one to pop up.

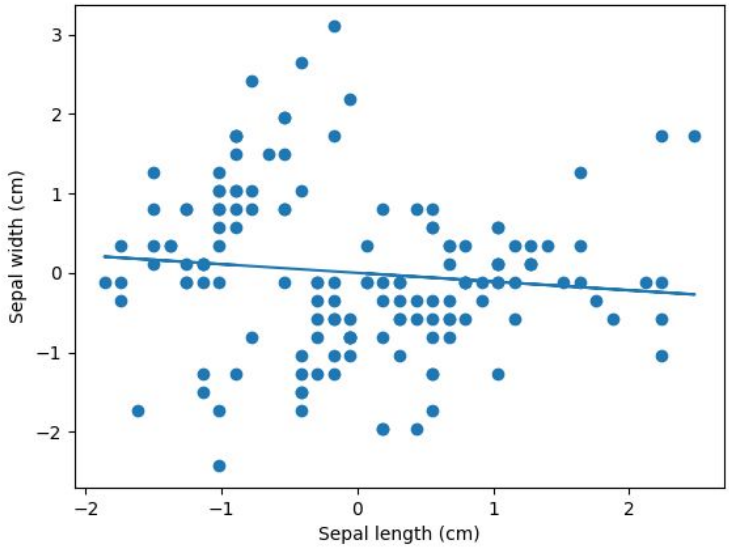
Test 1:

Features Compared: Petal Length vs Petal Width



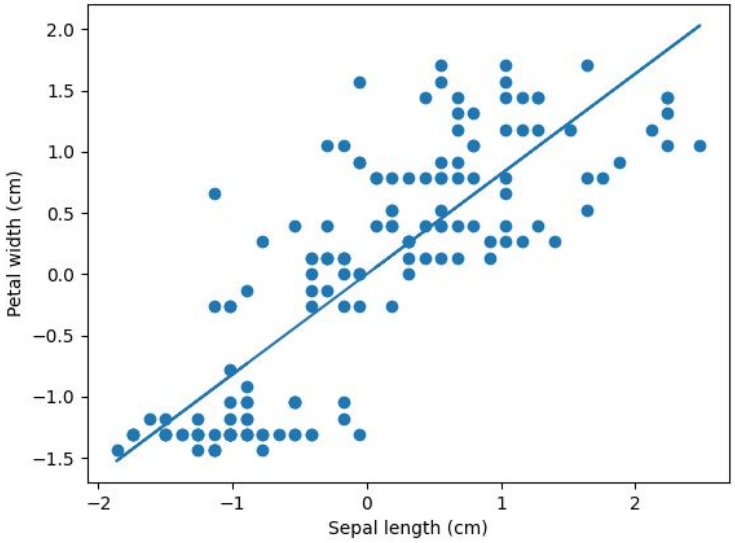
Test 2:

Features Compared: Sepal Length vs Sepal Width



Test 3:

Features Compared: Sepal Length vs Petal Width



6.0 Logistic Regression

6.1 Algorithm

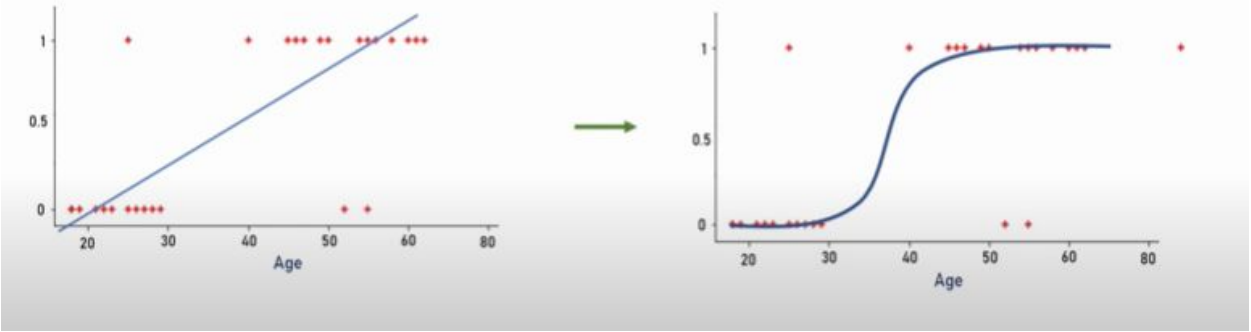
Logistic regression is a classification algorithm because as Shakespeare said, “What’s in a name?” We saw how linear regression works in section 5.0. A lot of times, it doesn’t make sense to draw a straight line as for a lot of datasets, there are outliers. The coefficient, or the ‘m’ in ‘y=mx+c’ gets manipulated to fit the outliers, giving inaccurate results for a lot of other data points that would have otherwise been on the correct side of the linear regression line. So, to fix this anomaly, we come up with a bent line, or a logarithmic curve. This curve is defined by the sigmoid function.

$$sigmoid(z) = \frac{1}{1 + e^{-z}}$$

‘e’ is Euler’s constant (around 2.7), so the denominator is always greater than or equal to 1. The concept behind using this function is to produce a value between zero and one, and classify it with the closest. You can see the comparison between linear regression and logistic regression in this figure.

$$y = m * x + b$$

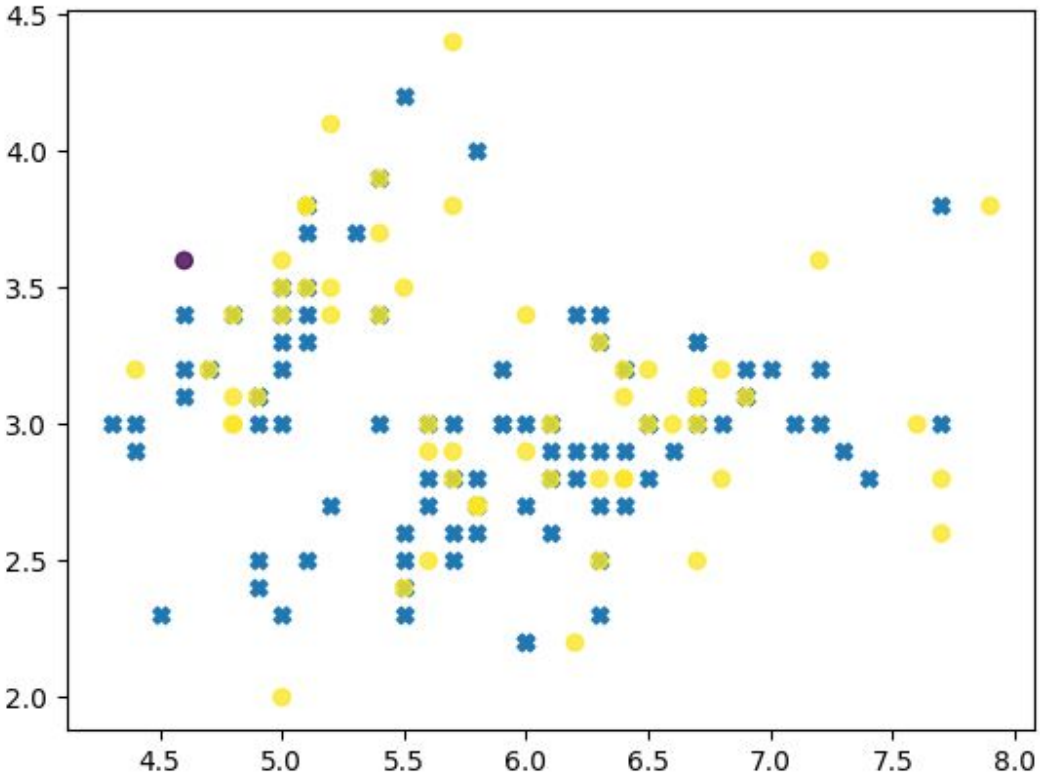
$$y = \frac{1}{1 + e^{-(m*x+b)}}$$



We obviously have a fit(function to train the data with this function and to continuously update the weights in it. We also have a find_sigmoid() function to compute the value of the function.

6.2 Examples

For our example, we have made the training data 2-Dimensional. We train on the sepal length and sepal width of the data keeping the petal length as the independent variable. 40% of the data is used for testing, with a learning rate of 0.3. The predictions are color coded to show the accuracy. Yellow dots show if it matched the expectations. Crosses are just the training data points.



7.0 Decision Stumps

7.1 Algorithm

To understand decision stumps, we need to know what decision trees are. In a nutshell, they are classification trees (can also be regression trees for other implementations) which branch out into different levels to predict the classification based on a binary distinction. These trees grow out logarithmically as you increase the features and introduce new variables.

Decision stumps are a simple implementation of the decision trees. They are basically one-level decision trees which literally compute a ‘stump’ in between a distribution of data to classify it.

According to the input, one can get different data from the decision stump according to the implementation. In this implementation, we have two functions. stumpClassify() and buildStump(). The purpose of the former is to just classify training data and do threshold analysis to label them +1 or -1. The comparisons are open for modification according to the needs of the user. It takes care of multiple dimensions of features too. In buildStump(), we call the stumpClassify() using different inputs to find the best decision stump for our data. This function also takes into account the weighted error, threshold, etc.

7.2 Examples

The decision stumps have been tested for the following examples. The best stump calculated is shown in the terminal. The remaining results are produced in the auto-generated text files in the same directory as the python files due to their size and to reduce clutter in the terminal. Both the tests will run together.

Test 1: Petal Length and Petal Width
Test 2: Sepal Length and Sepal Width

Terminal Output:

```
Test 1: Petal Length and Petal Width
The best stump is:
{'dim': 0, 'thresh': 1.59, 'ineq': 'gt'}
Test 2: Sepal Length and Sepal Width
The best stump is:
{'dim': 0, 'thresh': 3.94, 'ineq': 'lt'}
```

Files Generated:

Test1: decisionStumpsResults1.txt
Test2: decisionStumpsResults2.txt

The auto-generated report files will look somewhat like this:

```
Split:
Dimension: 0, Threshold: 0.41, Threshold Inequality: lt, The weighted error is 0.520

Split:
Dimension: 0, Threshold: 0.41, Threshold Inequality: gt, The weighted error is 0.480

Split:
Dimension: 0, Threshold: 1.00, Threshold Inequality: lt, The weighted error is 0.527

Split:
Dimension: 0, Threshold: 1.00, Threshold Inequality: gt, The weighted error is 0.473

Split:
Dimension: 0, Threshold: 1.59, Threshold Inequality: lt, The weighted error is 0.433

Split:
Dimension: 0, Threshold: 1.59, Threshold Inequality: gt, The weighted error is 0.567

Split:
Dimension: 0, Threshold: 2.18, Threshold Inequality: lt, The weighted error is 0.480
```

8.0 Prediction Intervals

8.1 Algorithm

For datasets of smaller size or for datasets with major outliers, linear regression cannot produce accurate results as we have discussed. This is where prediction intervals come into picture. The main purpose of implementing this algorithm is to predict an interval with a very high probability of being right, instead of predicting a specific value that has a high chance of being incorrect for a single datapoint. This overcomes the shortcomings of the predict function in linear regression but is also not as accurate so it has its own pros and cons. The objective of the algorithm is to not produce the optimum results but to reduce uncertainties or noise in linear regression. That is why I have made it a wrapper around linear regression, so that we can run them hand in hand which provides us with advantages of both.

First, we train our data using the fit() function in linear regression on the explanatory variable and the independent vector. Then we make our prediction line based on the coefficients and intercepts provided by the predict() function in linear regression. We then select a random datapoint from the data to be our sample for prediction. We then define the expected value, and the value to be predicted. The standard deviation for the predictable data is then calculated. For this algorithm, we're trying to hit a 95% accuracy for our interval, so we're using a factor of 1.96 for standard deviation calculations. Once we have our intervals set, we can just observe if the expected value lies within the bounds.

8.2 Examples

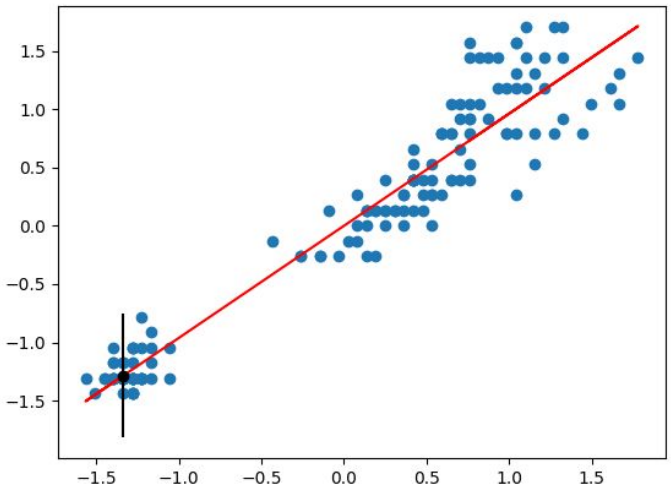
For the ease of understanding, the prediction intervals were run on the same three pairs of features as linear regression (section 5.2).

The first image of the terminal describes the prediction intervals and the expected value for the three test cases. The plots for the three tests follow. The black line represents the interval and the black dot represents the actual value. You might get different results for your run, since the datapoint to be tested at an instance is randomly chosen.

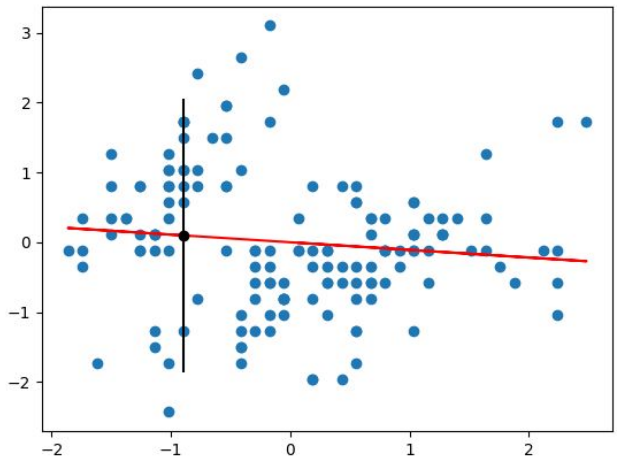
Note: Close each graph for the next one to pop up.

```
Predicting 95% chance of the value being between -1.819 and -0.755
Expected value: -1.309
The prediction interval is: 1.955
Predicting 95% chance of the value being between -1.857 and 2.053
Expected value: 1.029
The prediction interval is: 1.131
Predicting 95% chance of the value being between -2.360 and -0.097
Expected value: -1.309
```

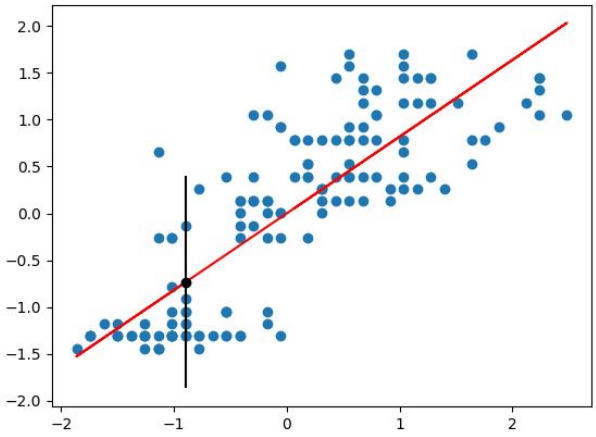
Test 1: Petal Length vs Petal Width



Test 2: Sepal Length vs Sepal Width



Test 3: Sepal Length vs Petal Width



9.0 Support Vector Machine

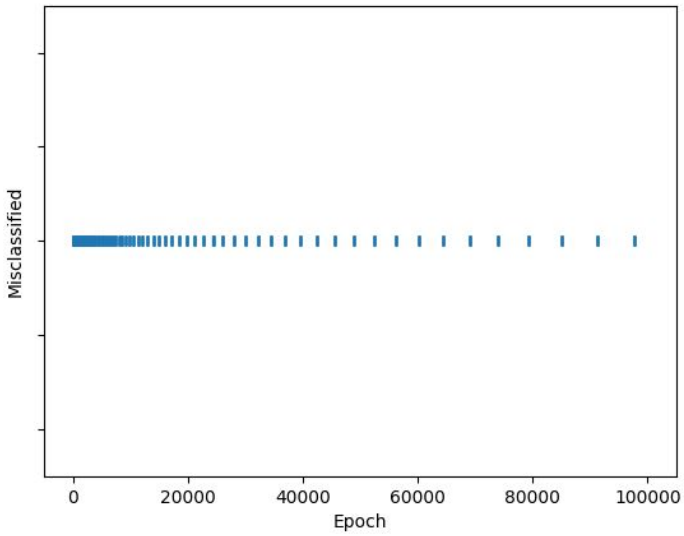
9.1 Algorithm

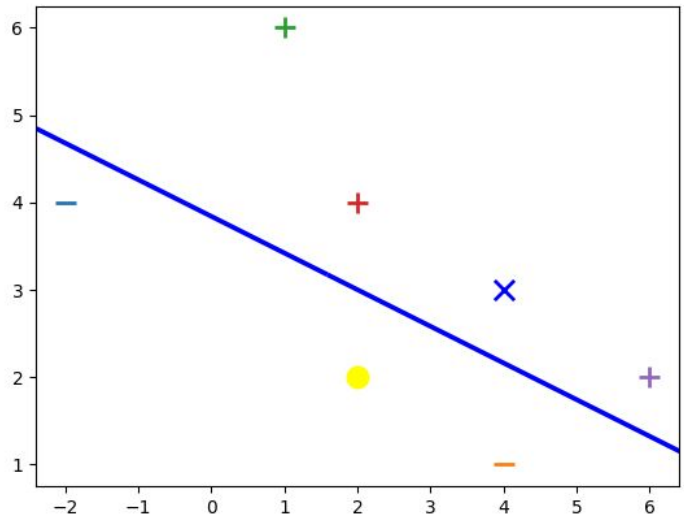
Support vector machines can act as classifiers as well as regression models, but are mostly used for classification purposes. The objective of a SVM is to draw a hyperplane that can divide the data into two parts. For classifications between two classes, you can say that the hyperplane is equivalent to a regression line in linear regression or the class division in a perceptron. SVMs come into picture when the data is not very big, but there are multiple classes involved. For example, when you have a basket of different fruits. The basic idea is to have all the classes as far away from the hyperplane as possible, or in other words, to find the largest area in the middle of all the data that puts each class on the correct side. Hence, SVMs work best on small datasets but it is difficult to generate a hyperplane for big datasets and 3D geometry comes into picture. Support vectors are just points from each class, closest to the hyperplane area. Classification is done by drawing lines to the center of the hyperplane from each class.

For this implementation, we have used the hinge loss algorithm. In this algorithm, we use a regularizer to balance between margin maximization and loss. We need regularization to find the decision surface that is maximally far away from any data points.This can be optimized by changing regularization factor and loss function via trial and error. We then update our weights using a gradient decent dictated by regularizer and loss.

9.2 Examples

For the purpose of running, we have used a very small manually created example. It shows two graphs: The miscalculations as the number of iterations or epochs increases. The second demonstrates the SVM with six different classes. One can add more data to the sample vector to classify further.


Misclassified
0
20000
40000
60000
80000
100000
Epoch


6
5
4
3
2
1
−2
−1
0
1
2
3
4
5
6

10.0 K-Nearest Neighbors

10.1 Algorithm

K-Nearest Neighbors can be used for both regression and classification according to one's needs. For the purpose of this project, we are using it as a typical classification model. While implementing this classifier, we are assuming a simple thing on which the whole algorithm is based - similar things fall in the same category or class, so data points that are close to each other on the cartesian plane fall in the same category. This algorithm very specifically is currently a 1-NN algorithm, but it can be modified into a K-NN with minor modifications. The tools and concepts used for this algorithm can be called a poor man's priority queue, since this is a similar implementation. Basically, we find the k points closest to the current point that we are predicting based on each feature of the data, since this takes care of a multidimensional feature input. These distances are just the euclidean distances between two points. A disadvantage of having a 1-NN algorithm is that our predictions have either a 0 or 100 percent chance of being correct for the current datapoint. Usually, at the edge of each class, there is a huge possibility of the point being closer to a point from another class. Having a higher value of k will increase the probability of predictions, however, having too big a k will result in unnecessary skewing of data.

10.2 Examples

We are using the iris dataset to predict the type of flower using the other four features. In our example, we have split 90% data for training and 10% for testing. When you run the KNN part, the predictions of the type of flower for each of the testing data points will show up in the terminal, along with the desired output. You can compare this just by looking at the differences. Note that the train/test split is not the same always, so your results may differ compared to the following figure. The key is to check if your KNN algorithm successfully predicted the type of flower or not.

```
Prediction Number: 1
Expected: Iris-versicolor
Our KNN Classification: Iris-versicolor
Prediction Number: 2
Expected: Iris-versicolor
Our KNN Classification: Iris-versicolor
Prediction Number: 3
Expected: Iris-setosa
Our KNN Classification: Iris-setosa
Prediction Number: 4
Expected: Iris-setosa
Our KNN Classification: Iris-setosa
Prediction Number: 5
Expected: Iris-virginica
Our KNN Classification: Iris-virginica
Prediction Number: 6
Expected: Iris-virginica
Our KNN Classification: Iris-virginica
Prediction Number: 7
Expected: Iris-versicolor
Our KNN Classification: Iris-versicolor
```

11.1 Future Expansions

1. As I mentioned above, this library can be leveraged to make a fully-fledged predictor for several datasets. The next one we'll be looking at is the Titanic passengers data.

2. Currently, the perceptron only works on cartesian coordinates. This can be changed into polar coordinates too. In some cases, drawing an elliptical division around the data when one class is too close to the center compared to the other one, makes much more sense and provides better results.

3. In addition to this, the linear regression model can be modified to run a variable number of dimensions instead of just one.

4. Cricket (sport) analysts use decision stumps a lot in their visuals to predict how a bowler is going to bowl. It would be cool to implement that.

5. There are a few other low VC-dimensional hypotheses classes that can be implemented. The most probable one as a next goal would be Axis Aligned Rectangles.