Machine Learning Library

Woodlin Smith

Introduction

Throughout CSC 448, I was exposed to a variety of machine learning algorithms, both for classification and regression. However, most of the exposure came in the form of the theory behind the algorithms, which while important, does not give me the hands on exposure to the algorithms that I would need were I actually wanting to use them. This portfolio hopes to remedy that, by demonstrating implementations of some of the most important algorithms that we covered throughout the class.

It also hopes to show that these algorithms are useful. By showing first hand that these relatively simple algorithms can glean useful things from data, it will hopefully demonstrate the importance of this field and why people study it.

Each section of the portfolio documentation contains

- A basic description of the algorithm, explaining the underlying logic and why it works.
- Usage information for the code.
- Descriptions of other functions in the code, if they exist.
- Testing information.
- Plots if applicable.

Every algorithm in the portfolio is implemented in ML.py. They are all implemented without using the built in versions included in Python libraries like scikit-learn.

All of the algorithms were developed and tested with the Iris dataset. The Iris dataset is a dataset with 150 elements, 4 features, and 3 classes. This dataset was chosen for its relative simplicity, high separability, and ease of use. Features of the dataset, in order, are

1. Sepal length
2. Sepal width
3. Petal length
4. Petal width

The three classes are:

- Iris-setosa
- Iris-versicolor
- Iris-virginica

Hopefully, this portfolio will show the use of these algorithms and why machine learning is a rapidly growing and utilized field in the world of computer science.

Machine Learning Library

Perceptron

Woodlin Smith

I. Introduction

The first part of this library is an implementation of a Perceptron. The Perceptron tries to classify 2 groups of separable data by finding a hyperplane between them. It can use any and all features of the groups it tries to classify, so long as there are only 2 groups and they are separable. It accomplishes this by slowly building a vector of "weights", which represents the hyperplane separating the data points. These weights are calculated through a series of iterations predicting the class of each point of data in the training set. If the prediction is incorrect, the weights are updated. This process eventually converges such that no elements in the training data are misclassified. Once this occurs, the data is considered fit to the model and the Perceptron stops.

II. Usage

In order to use the Perceptron, it needs to be imported with `from ML import Perceptron`. Then, the Perceptron can be instantiated 2 ways:

1. `Perc=Perceptron(rate, niter)`, where "rate" is the desired learning rate and "niter" is the number of desired iterations.
2. `Perc=Perceptron()`, in which case rate and niter are set to their default values of 0.1 and 10.

After it is instantiated, it can be used within Python's REPL. a. Fit

After the Perceptron is created, it needs to build a model from some set X of training data and a set y of target labels. This is done with by calling `perc.fit(X,y)`. Some things to note:

1. X is expected to be an array of arrays, with each individual array representing a data point.
2. y is expected to be an array of corresponding labels for these data points, with values of either -1 or 1.

b. Error

After the data is fit, you can view some statistics about the Perceptron. In order to view how many misclassifications occurred at each iteration, call `perc.errors`

c. Weights

In order to view the final weight array, call `perc.weights`

d. Net_input

Nominally a helper function that is used behind the scenes in the predict function. It calculates the dot product between a data point 'x' and the weight vector. Can also be called on its own by calling `perc.net_input(x).` It can also be called on a set of data points, in which case it will return an array of the corresponding dot products.

e. Predict

Predicts the label of a given data point "x" or array of data points. Can be called on its own by calling `perc.predict(x)` where "x" is a data point or array of data points.

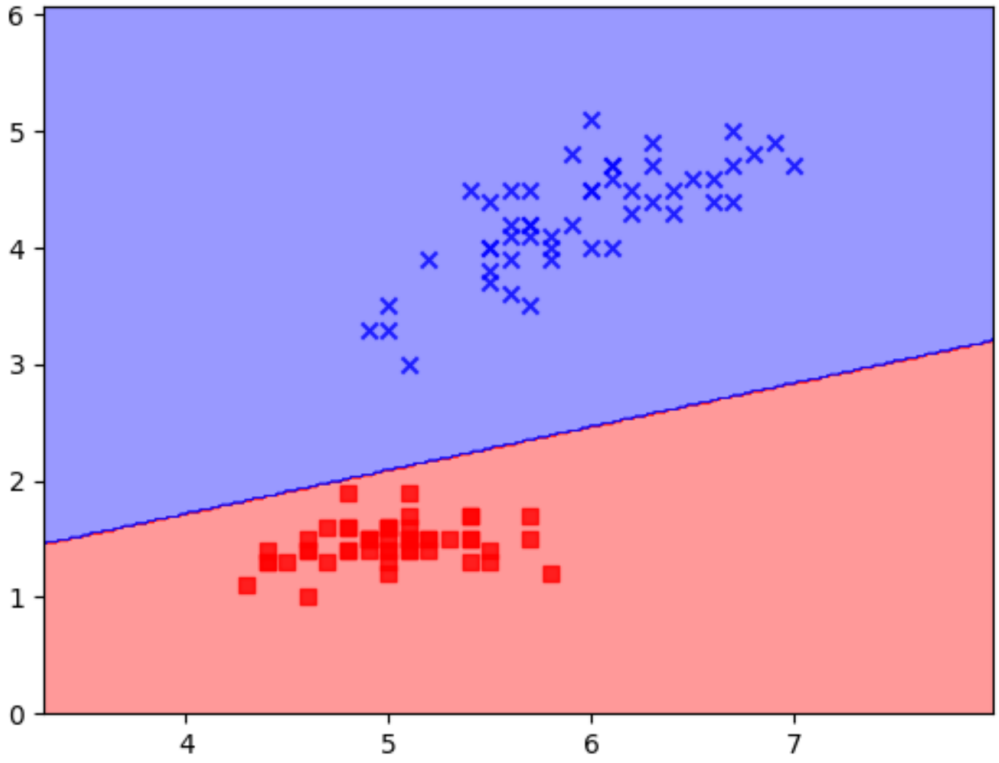
III. Dataset

In order to test this Perceptron, I used the Iris dataset[1]. This dataset contains 3 distinct classes, with 4 features per data point. This allowed for ease of separability and testing on feature sizes larger than 2.

IV. Driver Program

A driver program illustrating basic use is included. It can be run with `python driver_perc.py`

V. Plot



Machine Learning Library

Simple Linear Regression

I. Introduction

The library includes an implementation of a simple linear regression model. Simple linear regression differs from regular linear regression in that the dimensionality of the input features is 1, whereas the full version accepts input data of any dimensionality. Regression tries to fit data to a function, while linear regression will specifically fit the data to a line of the form y=mx+b. The model tries to learn the values of m and b. It accomplishes this by creating a "weight" vector of size 2. Then, using the ERM rule for linear regression, it fills the weight vector with b and m.

II. Algorithm Description

The linear regression algorithm takes in a vector of inputs (**x**) and a vector of their corresponding outputs (**y**). It first adds a column of 1's to **x** in order to generate a bias term, which will be the y-intercept in the resulting function. Then, it uses the least squares algorithm to find the values of the weight vector **w**. Least squares can be simplified into solving the following system of equations.

$$\boldsymbol{w}=A^{-1}\boldsymbol{b}$$

- A is a matrix generated by $\mathbf{x}^{\mathbf{T}}\mathbf{x}$**.**
- **b** is a vector generated by $\mathbf{x}^{\mathbf{T}}\mathbf{y}$

After this multiplication is complete, b is in **w**[0], and m is in **w**[1].

III. Usage

In order to use the linear regression model, it must first be imported using `from ML import LinearRegression`. Then, it can be instantiated with `lr=LinearRegression()`. After that it can be used within Python's REPL.

a. Fit

In order to fit the model, call `lr.fit(X,Y)`, where X is the input data vector and Y is the output data vector. These are both expected to be 1 dimensional vectors.

b. Weights

After fitting the model, call `lr.weights` to view the weight vector.

c. Prediction

After fitting the model, call `lr.predict(x_val)` to calculate what the corresponding output would be for the input. x_val is assumed to be a single value. If no model is fit, it will throw a ValueError and return -1. Otherwise it will return the predicted value

d. R^2

After fitting the model, call `lr.calc_rsquared(x_vec)`. This gives some insight into how the model is performing on the data. x_vec is assumed to be the original input vector that you used to fit the model. If no model is fit, it will raise a ValueError and return -1. Otherwise it will return the R^2 value.

IV. Dataset

All development and testing was done using the Iris dataset.

V. Testing

Model was tested against Microsoft Excel's linear regression. I ran my model, recorded the R^2 value, and then checked to make sure it matched in Excel.

VI. Driver Program

The submission includes a small script to run the model and demonstrate the usage and its accuracy. To run this program, just call `python driver_lg.py` from the command line in the project directory.

VII. Plots

The following are plots from a run on Iris Setosa. X was sepal width, Y was sepal Length

Linear Regression Model:

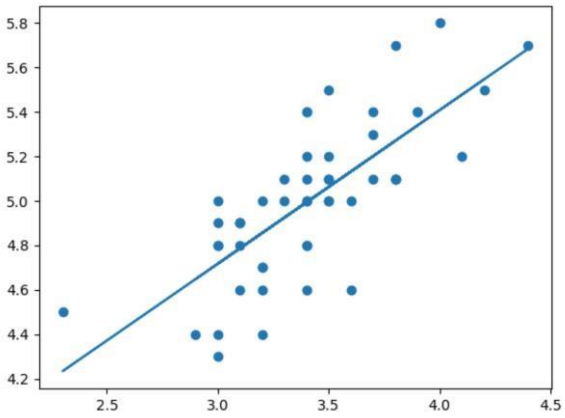


Figure 1- Plot of linear regression model. R^2 approx 0.55769

Excel's Linear Regression:

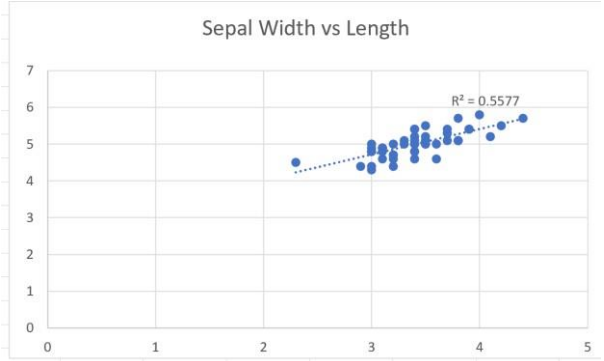


Figure 2- Plot of Excel's linear regression model. R^2 approx 0.5577

Machine Learning Library

Intervals Hypothesis Class

I. Introduction

The library contains an implementation of the hypothesis class of intervals. This class is any function h(x) such that

$$h(x) = \begin{cases} 1, a \leq x \leq \\ -1 \quad b \end{cases}$$

In order to provide a "usable" instance of this, the Interval class is set up to be similar to a binary classifier. The user provides a set **x** of input values and a set **y** of labels (1 and -1). The class then finds each instance of class 1, finds the minimum and maximum values of those instances. Those become the a and b values defined above. In practice, it is a fairly weak classifier, only truly useful if the input data is highly separated.

II. Usage

The interval hypothesis class must first be imported by calling `from ML import Interval`. After that, it can be instantiated with `inter=Interval()`. Then it can be used in the REPL.

a. Cutoffs

To find the values of a and b, call `inter.find_cutoffs(X, Y)`. X is assumed to be a set of values from a feature, and Y is assumed to be a set of class labels{1,-1}. It will find the minimum and maximum X values where the class label is 1, and set a and b equal to them.

b. Min threshold and Max threshold

To view the cutoff values, call `inter.min_thresh` and `inter.max_thresh`.

c. Predict class label

To predict the class label given a value or set of values, call

`inter.predict_label(x)`. If x is a single value, it will either return 1 or -1. If it is a set of values, it will return an array of labels.

III. Dataset

It was developed and tested using the Iris dataset, and the sepal length attribute. The sepal length attribute was chosen since Iris Setosa's sepal length is largely separated from the other 2, leading to an ideal binary classification scenario.

IV. Driver Program

Included is a small driver program demonstrating the capabilities of the class. In order to run it, call `python driver_int.py` in the project directory.

Machine Learning Library

Threshold Hypothesis Class

I. Introduction

The library contains an implementation of the threshold hypothesis class. A function h(x) exists in this class if it matches the form

$$h(x) = \begin{cases} 1, & x \leq a \\ -1 & \end{cases}$$

Similarly to the intervals class, this is implemented in the library as a binary classifier. The user will provide a set **x** of data points and a set **y** of labels (1 and -1). The class will find every instance of class 1, and then find the max value out of those instances. That becomes the value of *a* as shown in the equation. Like the intervals class, this is a weak classifier that only becomes useful if the data is separable. It is included as an example of a weak hypothesis class, which could be used in boosting algorithms like ADABoost.

II. Usage

The class must first be imported by calling `from ML import Threshold.` Then, it can be instantiated with `t=Threshold()`. After that, it can be used within Python's REPL.

a. Cutoff

To find the cutoff value, call `t.find_cutoff(X,Y)`. X is expected to be a set of values from a single feature, and Y is a set of labels from the set {1,-1}. It will find the max value within X, and set the cutoff value equal to that.

b. Predict

To try and predict a class label, call `t.find_cutoff(X)`. X is expected to be either a single value from a single feature, or a set of values from a single feature. For every value in X, it will check to see if it is less than or equal to the threshold value, returning 1 if it is and -1 if it is not.

c. Threshold

To view the threshold value, call `t.max_thresh.`

III. Testing

The class was tested using the Iris dataset, and the sepal length feature. Like the intervals class, this was picked because members of the "Iris-setosa" class are very separated from the other 2.

IV. Driver Program

To run the driver program, which demonstrates the usage of the class, simply call `python driver_thresh.py`

Machine Learning Library

K Nearest Neighbors

I. Introduction

The library contains an implementation of the K Nearest Neighbors algorithm, which can be used for both classification and regression. Unlike some other learning algorithms, the K Nearest Neighbors algorithm is actually intuitive. Its premise is simple: given an unlabeled data element, a set of data **x**, and a set of labels **y**, find the *k* closest labeled data elements. From there, one can do 2 things: classify the unknown element by finding the mode of the neighbors' labels, or find the average value of the neighbors' labels, which can be used for regression. This algorithm is useful because it's relatively simple to implement, and can be used for multi class classification without any adjustments.

It is implemented in the library as a class, and can be used for both classification and regression of unlabeled data points.

II. Usage

In order to access the KNN class, it must be imported by calling `from ML import KNN`. Then it can be instantiated with `knn=KNN(X,Y)`. X is a vector of data elements that can have any amount of features, though it is assumed the features are numeric. Y is a vector of the corresponding labels in the set {1, -1}.

a. Make Prediction

To make a prediction for the class label or average label value, call `t.predict(k, queries, distance_method, choice)`. "k" is an integer value for the amount of neighbors to use. "queries" is either a single unlabeled data element, or a list of unlabeled elements. "distance_method" is the desired distance equation to use, which must be taken from the set {"e", "m", "s"}. These correspond to the Euclidean, Manhattan, and Supremum methods for calculating distance respectively. "choice" is which mode to run the prediction in, which will change what is returned. This must be taken from the set {"c", "r"}, which represent classification and regression respectively. The distance_method and choice default to "e" and "c" respectively.

III. Other Functions

This class contains various other utility functions that are used throughout the prediction function.

a. knn: float or int, depending on mode
    i. Description: performs the KNN algorithm on a query vs a training dataset. For each data element, it will calculate the distance between it and the query element. Then, it will add the distance and index to a list. After the

distances are calculated, the list is sorted, and the labels for the first k elements of the list are retrieved. Then, based on the user's desired results, it will either return the mean or mode of the k nearest labels

ii. Parameters
    1. k:int – the amount of neighbors to pull
    2. query:list[float] – the query example to test against
    3. distance_method="e": str – the distance method to use
    4. choice="c" – the return method to use

iii. Returns either the mean of the labels or the mode of the labels depending on what the user specifies.

b. euclidean_dist: float
    i. Description: Calculates the Euclidean distance between a data element and a query example.
    ii. Parameters:
        1. Element: list[float] – a data element
        2. Query: list[float] – the query example
    iii. Returns the Euclidean distance between these elements

c. manhattan_dist: float
    i. Description: Calculates the Manhattan distance between a data element and a query example.
    ii. Parameters:
        1. Element: list[float] – a data element
        2. Query: list[float] – the query example
    iii. Returns the Manhattan distance between these elements

d. supremum_dist: float
    i. Description: Calculates the Supremum distance between a data element and a query example.
    ii. Parameters:
        1. Element: list[float] – a data element
        2. Query: list[float] – the query example
    iii. Returns the Supremum distance between these elements

e. knn_mode: int
    i. Description: An implementation of a mode function to use on the k nearest labels.
    ii. Parameters:
        1. Labels: list[int] – a list of labels
    iii. Returns the mode of the list

IV. Testing

This class was tested using the iris dataset by taking the odd elements as a training set and the even elements as a test set. I checked to see that it was classifying the unseen elements correctly.

V. Driver Program

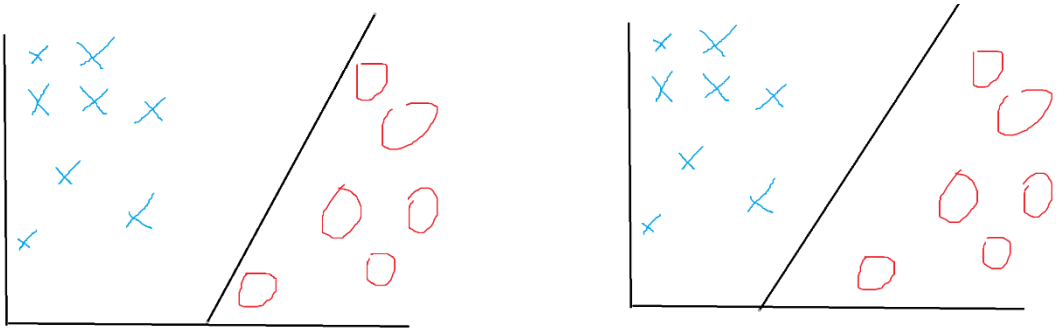
To run the driver program, simply call `python driver_knn.py`.

Machine Learning Library

Soft SVM

I. Introduction

The library contains an implementation of a “soft” Support Vector Machine. A Support Vector Machine is a binary classification algorithm that tries to find the “best” hyperplane separating two classes of data. While it is hard to quantify what “best” means, it is easier to visualize. Consider the 2 plots below.



While both hyperplanes separate the data, it is easier to see that the plot on the right is more representative of how the classes are structured. A Support Vector Machine tries to generate hyperplanes more in line with the plot on the right. It does this by trying to maximize the margins between the two classes. It tries to find two vectors that touch the points in each class that are closest to the other. These become the “support vectors”.

A “hard” SVM is good when the data is guaranteed to be linearly separable. It will try to find the hyperplane with the maximum distance from each class, with the support vectors lying on the points that are closest to the other class. However, as the name suggests, it is not very flexible. If the data is not linearly separable, it will be almost unusable. That is where a “soft” SVM comes into play.

A “Soft” SVM is an extension of the “hard” SVM. It accounts for non-linearly separable data with an addition of the hinge loss function when an example is on the “wrong” side of the margin. The hinge loss function allows for examples to be misclassified, but it makes the SVM usable in more cases.

The soft SVM's loss function can be minimized by using a technique called Stochastic Gradient Descent. In general, Gradient Descent is a minimization technique that finds the smallest value in a function by calculating the gradient, and "walking away" from it. Stochastic Gradient Descent is a modification where the gradient is calculated at randomized individual points in the training set. This process is repeated many times to truly minimize the loss function.

The SVM is implemented as a class in the library.

II. Usage

The SVM must first be imported with `from ML import SupportVectorMachine`. After that, it can be instantiated with `svm=SupportVectorMachine(num_iters, learning_rate)`. "num_iters" is the number of iterations to run the stochastic gradient descent, and "learning_rate" is the learning rate for the descent. They have default values of 1000 and 0.001 respectively. After it is instantiated, it can be used in the REPL.

a. Fit
   To fit a model, call `svm.fit(X,Y)`. X is assumed to be a list of data objects or a list of selected features from data objects. Y is the corresponding label vector. The function first adds a bias term to the X vector, and then it performs the stochastic gradient descent to get the specific weights needed.
b. Predict
   To predict a value, call `svm.fit(val)`. Val is either an individual data object, or a list of data objects. The prediction is done by returning the sign of the dot product of the weights with the val matrix.
c. Weights
   To view the weight vector, call `svm.weights`.

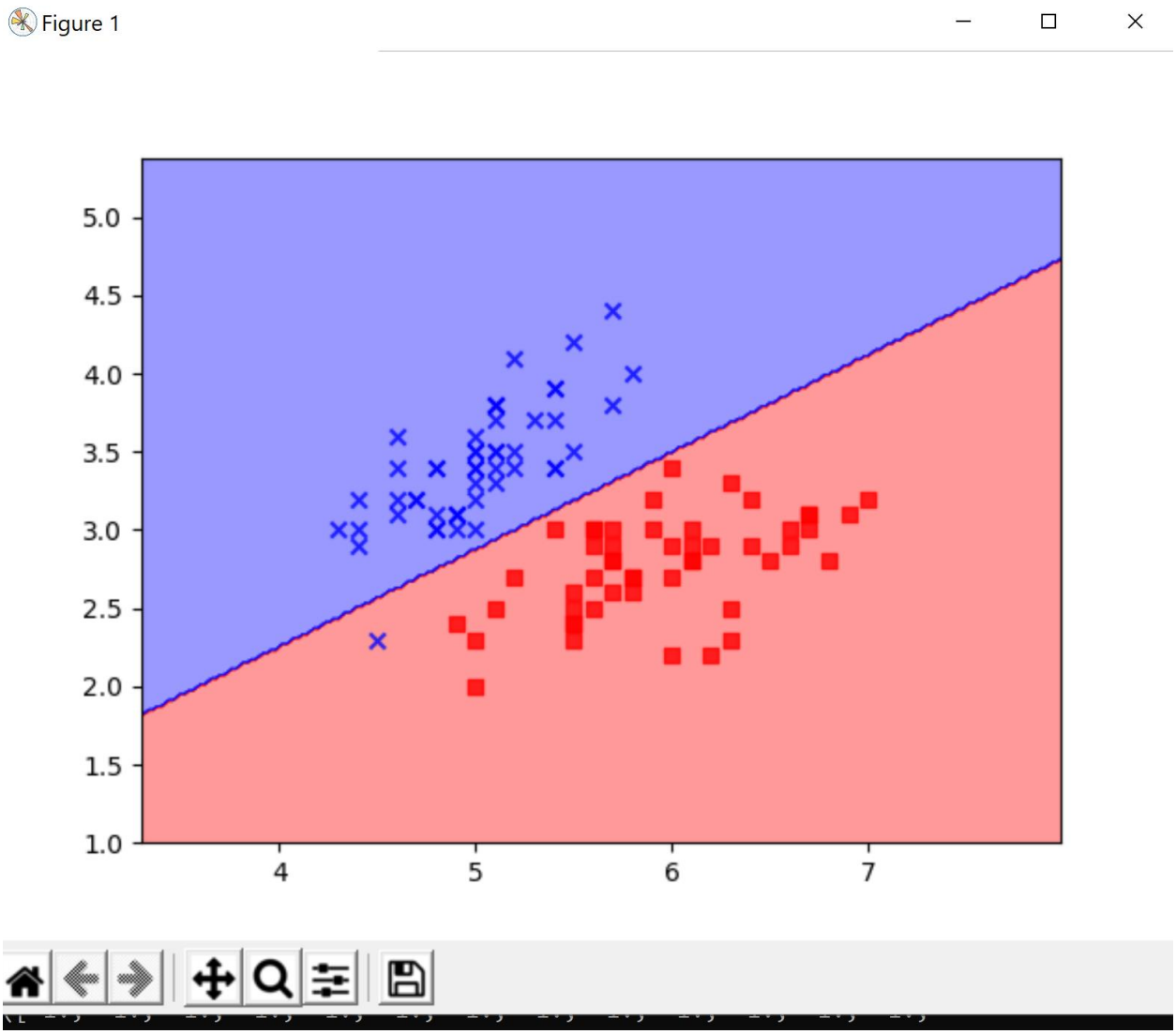
III. Testing

The classification accuracy of this algorithm was tested with the iris dataset, on features of sepal length and width. I ensured that it had no runtime errors and had a reasonable classification accuracy.

IV. Driver Program

The driver program can be run by simply calling `python driver_svm.py` from the REPL

V. Plots



As can be seen, the SVM performs fairly well, although there was a misclassification. This comes from the error possible in the “soft” SVM, which will misclassify outliers.

Machine Learning Library

Logistic Regression

I. Introduction

The library contains an implementation of Logistic Regression. Logistic Regression is, despite the name, a classifier. The reason is that the regression is onto a special function that predicts the probability that a condition is true. This function is called the sigmoid function, which is shown below.

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

This generates an "S" shaped curve between 0 and 1. The values represent the probability of an outcome. This is useful, since it can be used to classify data. For instance, it can say that a certain data point **x**i has probability *p* of being in class Y. With the simple addition of a threshold, this can be used to classify data.

The logistic regression is implemented as a class in the library. Gradient descent is used to minimize the loss function..

II. Usage

The logistic regression must first be imported with `from ML import LogisticRegression.` After that, it can be instantiated with `lg=LogisticRegression(learning_rate, num_iters).` "num_iters" is the number of iterations to run the stochastic gradient descent, and "learning_rate" is the learning rate for the descent. They have default values of 1000 and 0.01 respectively. After it is instantiated, it can be used in the REPL.

a. Fit

   To fit the model, call `lg.fit(X,Y).` X is assumed to be list of data objects, and Y is a list of labels from the set {1,0}. The logistic regression uses {1,0} for labels the sigmoid function runs from [0,1]. The function uses gradient descent to find the correct value of **w**.

b. Predict

   To predict the value of data, call `lg.fit(X).` X is a list of data objects. It predicts the value by calling the sigmoid function on the dot product of X and the weight matrix.

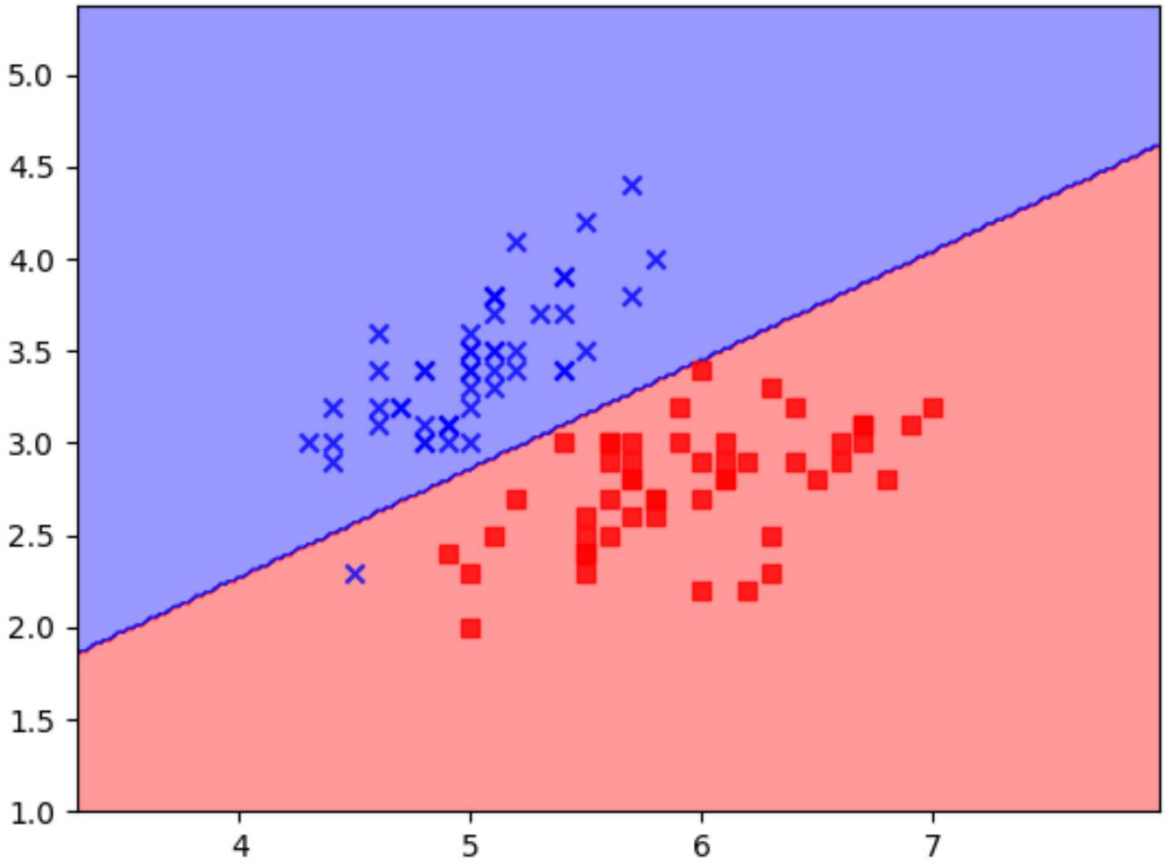
III. Testing

This model was tested on the iris dataset for correctness and classification accuracy. I tested it with the sepal length and width features.

IV. Driver Program

The Driver program can be run by calling `python driver_log.py`

V. Plots



As can be seen, it misclassified one point, although it is understandable as to why, as that point is an outlier. The Support Vector Machine had the same problem on this exact data set.

Machine Learning Library

Other Functions

I. Introduction

The library also contains 3 simple visualization functions. These functions will plot the decision boundary of a binary classifier, show a scatter plot of input data, and show a regression line plotted over input data.

II. Usage

All three functions can be imported by calling `from ML import <Function_Name>`. Then they can be used within the REPL.

III. Functions

a. `plot_decision_regions`
    a. Parameters:
        i. X – a set of data objects
        ii. y – a set of labels
        iii. classifier – the classifier to use
        iv. resolution=0.02 – scaling factor or plot
    b. Description:
        i. Plots the classifiers predicted boundary between 2 classes

b. `plot_scatter`
    a. Parameters:
        i. X – a set of features from a data object
        ii. Y – a different set of features from a data object to plot X against
    b. Description:
        i. Makes a scatter plot of the data

c. `regression_plot`
    a. Parameters:
        i. X – a set of features from a data object
        ii. Y – a different set of features from a data object to plot X against
        iii. reg_model – the regression model to use
    b. Description:
        i. Plots the regression line on top of a scatter plot of the data.