Programming Assignment 4

Perceptron Algorithm, Pocket Algorithm, Logistic Regression, and Linear Regression

Patrick Humphries (7097-1087-72, pvhumphr@usc.edu)
INF552 Machine Learning for Data Science (32458)
University of Southern California
Viterbi School of Engineering
Spring 2020

Contents

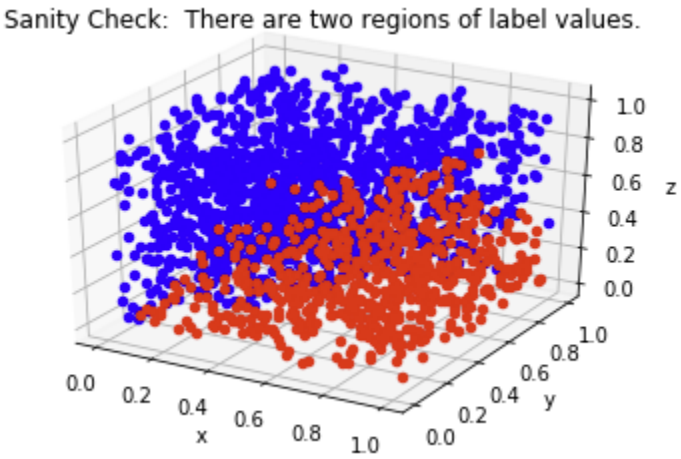
Part 1: Implementation

Requirement 1: Perceptron Learning Algorithm

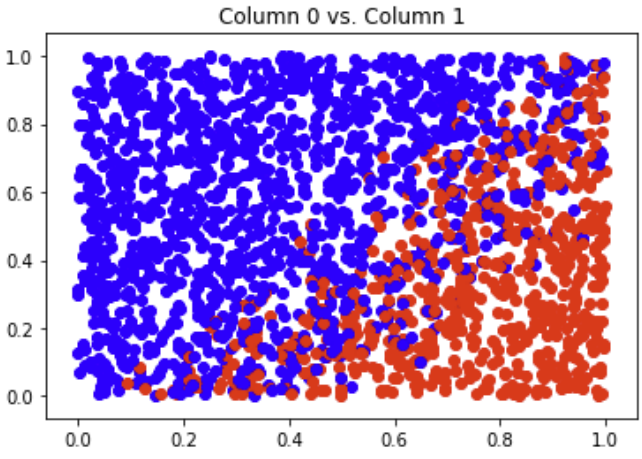
Abstract The Perceptron Learning Algorithm is a supervised learning algorithm. Base on input, a binary decision is made. The teaching dataset consisted of three attributes. The label dataset consisted of a Boolean type values. The process is iterated, refining the weights for the three attributes. The process ends when there are no further changes in the weights.

Step 1: Sanity Check

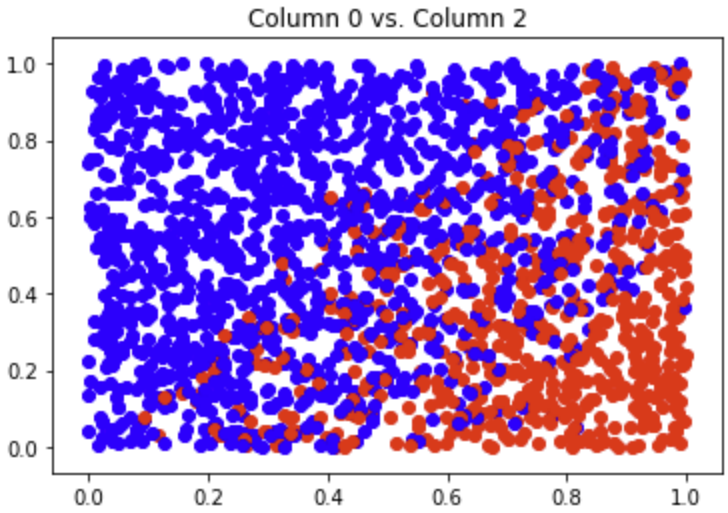
In order for this algorithm to work, there must be some division between samples. In two dimensions, it would be a line. In three dimensions it would be a plane. The first three-dimensional plot indicates that there are two groups. The blues indicate the negative values and positive values by the red values.



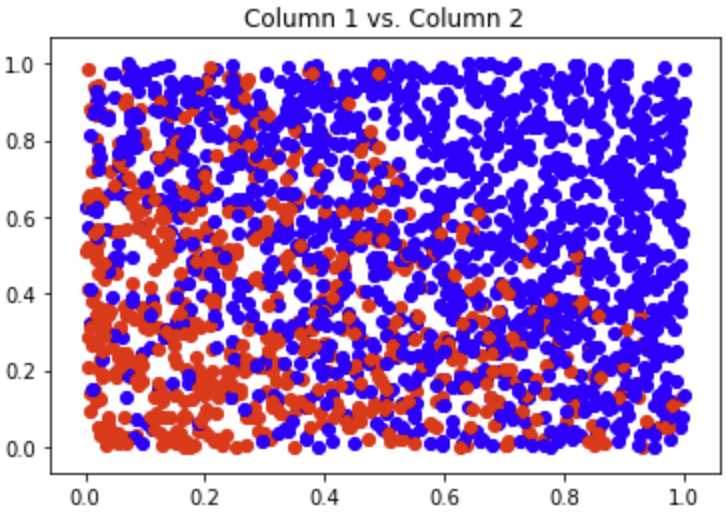
To be certain of a potential plane, the data was plotted using data from the first two columns of the input file. This looks relatively good. There is some commingling of data points. This will result in overlapping predictions. A Gaussian probabilities could be used for the step function which determines the Boolean type output.



Things are worse plotting the first and third columns.



Yuk! Extensive commingling. It will be a challenge using linear weights to predict with a degree of accuracy.



Step 2: Make Predictions

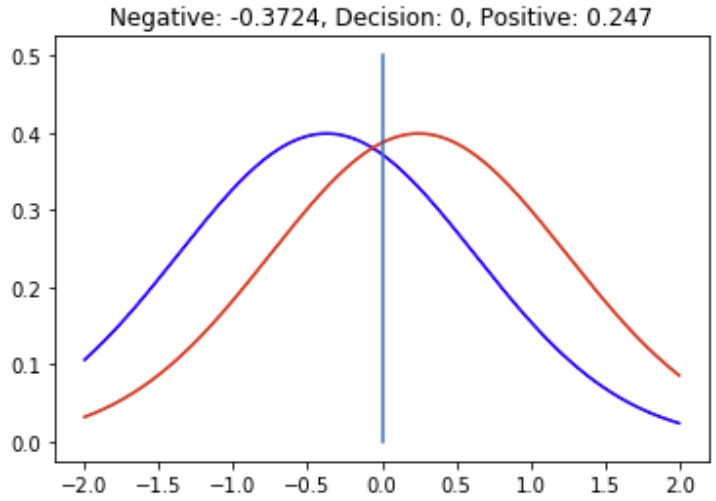
This algorithm is one of the simpler learning algorithms. The variables for predicting a label were three coordinate values. After making a prediction, the prediction is compared to the corresponding label. The differences are applied to the weights based on the previous value, a learning rate, and variable.

```
Break:  No change in weights.
Iteration: 1017 Accuracy: 0.99 Weights: [ 0.9074722  -0.7050772  -0.54085415]

Negatives:
        Mean: -0.3724
        Sigma: 0.249
        Min: -1.0974
        Max: 0.0177

Positives:
        Mean: 0.247
        Sigma: 0.1777
        Min: 0.005
        Max: 0.8446

Decision Point: 0
```



From the visual inspection of the data, it appeared that there would be some overlap of predictions. Predictions were bifurcated into two collections, one for negative values and one for positive values. They are plotted above. The best decision point was the value zero. Other decision points like the intersection of the Gaussian curves were attempted, but they were less accurate.

Step 3: Calculate Accuracy

An accuracy rate 0.99 was achieved. The iterative process constantly adjusts weights by minute amounts by the difference of the predicted value and the actual value as well as other factors. The main criteria for ending the process is when there is no further refinement of the weights. In this implementation, this occurred in approximately one thousand iterations. The required accuracy and weights are listed below. However, at iteration 100, there appears to be a better answer. In this case, the "Pocket Algorithm" could be applied to capture the better value.

```
Calculating Model
        Iterations: 10001
        Learning Rate: 0.0001
        Overfitting: 0

Iteration: 0 Accuracy: 0.6775 Weights: [ 0.00148693 -0.05227459 -0.0467224 ]
Iteration: 100 Accuracy: 0.9955 Weights: [ 0.87372373 -0.68506158 -0.5274393 ]
Iteration: 200 Accuracy: 0.99 Weights: [ 0.90631995 -0.70437419 -0.54041615]
Iteration: 300 Accuracy: 0.99 Weights: [ 0.90743284 -0.70505255 -0.54083983]
Iteration: 400 Accuracy: 0.99 Weights: [ 0.90747086 -0.70507634 -0.54085368]
Iteration: 500 Accuracy: 0.99 Weights: [ 0.90747215 -0.70507717 -0.54085413]
Iteration: 600 Accuracy: 0.99 Weights: [ 0.9074722  -0.7050772  -0.54085415]
Iteration: 700 Accuracy: 0.99 Weights: [ 0.9074722  -0.7050772  -0.54085415]
Iteration: 800 Accuracy: 0.99 Weights: [ 0.9074722  -0.7050772  -0.54085415]
Iteration: 900 Accuracy: 0.99 Weights: [ 0.9074722  -0.7050772  -0.54085415]
Iteration: 1000 Accuracy: 0.99 Weights: [ 0.9074722  -0.7050772  -0.54085415]

Break:  No change in weights.
Iteration: 1017 Accuracy: 0.99 Weights: [ 0.9074722  -0.7050772  -0.54085415]
```

Data Structures

The program made use of the *numpy* library. Vectors were used to store coordinates, labels, and weights. Vector algebra was used to calculate predictions. Other than this, the program is generic Python.

Main Routine

The program is structured as a collection of functions and one main routine, from where all functions were called.

```
# main:  All processing is controlled here.

# Print identification data.
if True:
    preamble()

# Load training data and initialize differences.
(X,L,W) = load_data()

# Check visually if the data could be bifurcated.
if True:
    title = 'Sanity Check:  There are two regions of label values.'
    plot_data_3d(X,L,title)
    plot_data_2d(X,L,0,1)
    plot_data_2d(X,L,0,2)
    plot_data_2d(X,L,1,2)

# Calculate the weights.
# X:  Training data.
# L:  Training labels.
# W:  Weights.
# I:  Maximum iterations allowed.
# R:  Learning Rate
# O:  Overfitting (Decision Points at zero, intersection, average)
I = 10001
R = 0.0001
O = 0
calculate_model(X,L,W,I,R,O)

# Run the sklearn.linear_model.Perceptron package.
if True:
    sklearn_perceptron()

print(' ')
print('Done!')
```

The "Overfitting" parameter was added to experiment with different decision points. They were the values zero, the intersection of the Gaussian curves of the predictions, and an average between Gaussian distributions.

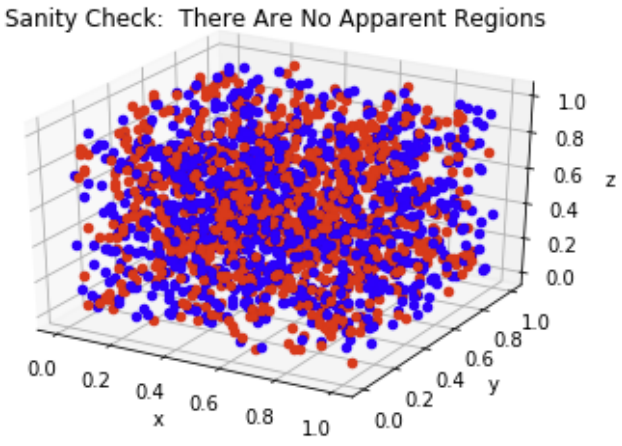
Requirement 2: Pocket Algorithm

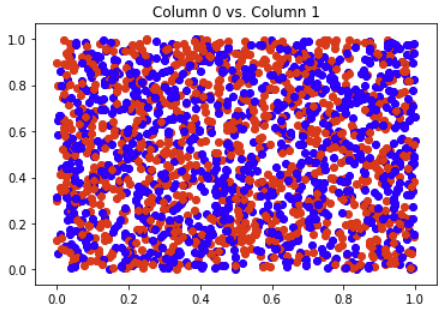
Abstract The termination of the iteration loop in the Perceptron Learning Algorithm is possible when the data can be bifurcated and termination criteria can be met. However, when data is cannot be separated into regions, then a termination criteria does not work. In this case, the "Pocket" variant is used. After a definite number of iterations, the weights with the best accuracy that is encountered is used for predictions.

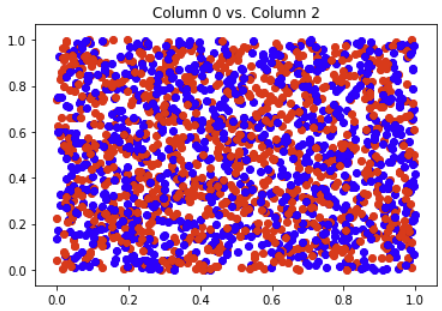
Step 1: Visualize Data

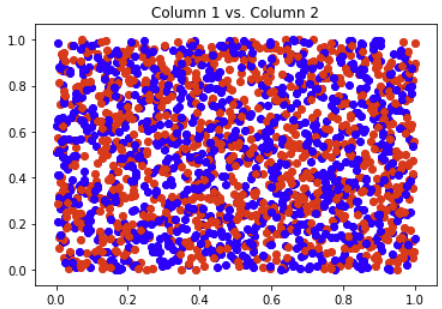
This is a challenge. Nearly all of the labels are equally divided between negative and positive values. If the model does not have an prediction accuracy of at least fifty percent, it would be more accurate just to predict everything label having a negative value. As evident below, the data appears to be evenly distributed.

```
Negative Values: 1012 0.506
Positive Values: 988 0.494
Total Values: 2000
```









Step 2: Make Predictions

The Perceptron Learning Algorithm is applied. A prediction is calculated by multiplying all of the coordinates by the corresponding weights. The results are summed and compared to the training label. The weights are then adjusted by the difference between the prediction and label, using the learning rate.

```
Iteration: 0 Accuracy: 0.506 Weights: [-2.68755032e-05 -2.47637415e-06 -5.36474516e-06]
Iteration: 1000 Accuracy: 0.514 Weights: [-0.00881903  0.0010015  -0.00019688]
Iteration: 2000 Accuracy: 0.514 Weights: [-0.00881903  0.0010015  -0.00019688]
Iteration: 3000 Accuracy: 0.514 Weights: [-0.00881903  0.0010015  -0.00019688]
Iteration: 4000 Accuracy: 0.522 Weights: [-0.03710837  0.01660915  0.00973208]
Iteration: 5000 Accuracy: 0.5235 Weights: [-0.03830144  0.01732911  0.01019529]
Iteration: 6000 Accuracy: 0.5235 Weights: [-0.03830144  0.01732911  0.01019529]
Iteration: 7000 Accuracy: 0.5235 Weights: [-0.03830144  0.01732911  0.01019529]
```

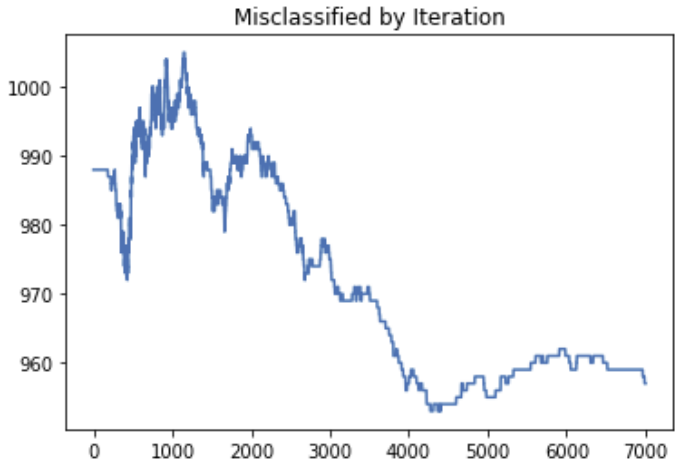
Step 3: Determine Accuracy

A accuracy rate of 0.5235 was achieved As expected, the prediction model is a bit more accurate as flipping a coin. An accuracy rate of 0.5235 was achieved after 4,271 iterations. When a higher accuracy rate is encountered, it is stored in the "pocket". If the accuracy rate encountered is the same as the one in the pocket, then the weights are examined. If the weights are less, then they too are stored in the pocket.

There is a requirement to plot misclassified versus iterations. As iteration 4,271, there were 953 misclassified predictions.

It is curious that beyond this iteration, the continued adjustment of the weights resulted in a decline in accuracy and an increased in the number of misclassified.

```
Best Accuracy: 0.5235
Weights: [-0.03830144  0.01732911  0.01019529]
Iteration: 4271
Misclassified: 953
Iterations: 7001
Learning Rate: 1e-06
```



Program Structure

The structure of the program is relatively simple. The program is a collection of functions that are called a main routine.

```
# main:  All processing is contolled from here.

# Print identification data.
if True:
    preamble()

# Load training data and initialize differences.
(X,L,W) = load_data()

# Check visually if the data could be bifurcated.
if True:
    title = 'Sanity Check:  There Are No Apparent Regions'
    plot_data_3d(X,L,title)
    plot_data_2d(X,L,0,1)
    plot_data_2d(X,L,0,2)
    plot_data_2d(X,L,1,2)

# Calculate the weights.
# X:  Training data.
# L:  Training labels.
# W:  Weights.
# I:  Maximum iterations allowed.
# R:  Learning Rate
if True:
    I = 7001
    R = 0.000001
    B = calculate_model(X,L,W,I,R)
    print(' ')
    print('Best Accuracy:', B["accuracy"])
    print('Weights:', B["weights"])
    print('Iteration:', B["iteration"])
    print('Misclassified:', B["misclassified"][B["iteration"]])
    print('Iterations:', I)
    print('Learning Rate:', R)
    print(' ')

# Plot miclassified versus iterations.
if True:
    # Limit the number of data points.
    n = 7000
    plot_missclassified(B,n)

# Run the sklearn.linear_model.Perceptron package.
if True:
    sklearn_perceptron()

print(' ')
print('Done!')
```

Requirement 3: Logistic Regression

Abstract Up to now, the Perceptron Learning Algorithm and the variant Pocket Algorithm have had a hard decision point of zero. This results in an inflexible binary prediction. However, when a sigmoid, using predictions as input, the probability of a correct prediction can be made. The decision point now is if the probability is greater than fifty percent.

Code

The code and data are the same as for the Pocket Algorithm except for one change in the code. Instead of comparing a prediction to a label to determine accuracy, the prediction is feed into a sigmoid, which in turn yielded a probability. If the probability is less than the decision point of 0.5, then the prediction is -1. Otherwise, it is +1. Again, this prediction is compared to the label.

```
#  Make predictions and compare to labels.
count = 0
correct = 0
i = 0
for x in X:
    count += 1
    y = x @ W
    sigmoid = np.exp(y)/(1 + np.exp(y))
    if sigmoid < decision_point:
        prediction = -1
    else:
        prediction = +1
    if prediction == L[i]:
        correct += 1
    i += 1
```

As expected, the results were the same as for the Pocket Algorithm.

Each *x* is a vector of coordinates. *X* is the container that is equivalent to the coordinates found in the file.

W is a vector of weights.

The equation *y = x @ W* is the dot product of the coordinates and the weight. Each coordinate is multiplied by the corresponding weight. Then, all the products are summed and returned in variable *y*.

The variable *y* is feed into the sigmoid and the output is compared to the label.

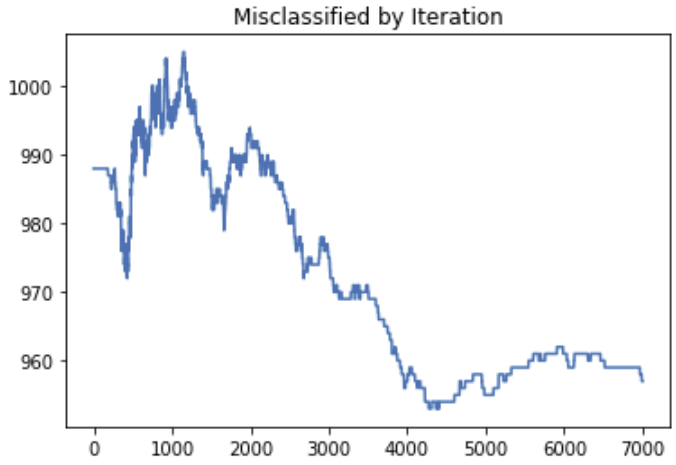
L is a vector containing labels that correspond to the corrdinates. It is loaded from the fifth column of the input file.

The decision point is the probability 0.5.

Results

```
Iteration: 0 Accuracy: 0.506 Weights: [-2.68755032e-05 -2.47637415e-06 -5.36474516e-06]
Iteration: 1000 Accuracy: 0.514 Weights: [-0.00881903  0.0010015  -0.00019688]
Iteration: 2000 Accuracy: 0.514 Weights: [-0.00881903  0.0010015  -0.00019688]
Iteration: 3000 Accuracy: 0.514 Weights: [-0.00881903  0.0010015  -0.00019688]
Iteration: 4000 Accuracy: 0.522 Weights: [-0.03710837  0.01660915  0.00973208]
Iteration: 5000 Accuracy: 0.5235 Weights: [-0.03830144  0.01732911  0.01019529]
Iteration: 6000 Accuracy: 0.5235 Weights: [-0.03830144  0.01732911  0.01019529]
Iteration: 7000 Accuracy: 0.5235 Weights: [-0.03830144  0.01732911  0.01019529]

Best Accuracy: 0.5235
Weights: [-0.03830144  0.01732911  0.01019529]
Iteration: 4271
Misclassified: 953
Iterations: 7001
Learning Rate: 1e-06
```



The results are the same as for the Pocket Algorithm. The best accuracy rate of 0.5235 was achieved at iteration 4,271. This seems better than chance of an equally distributed population of samples.

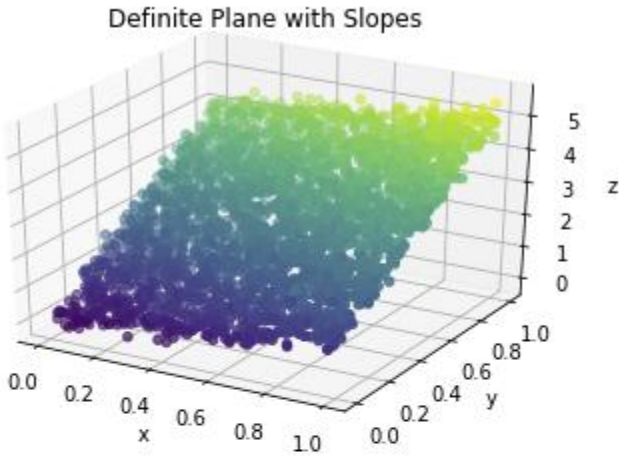
Requirement 4: Linear Regression

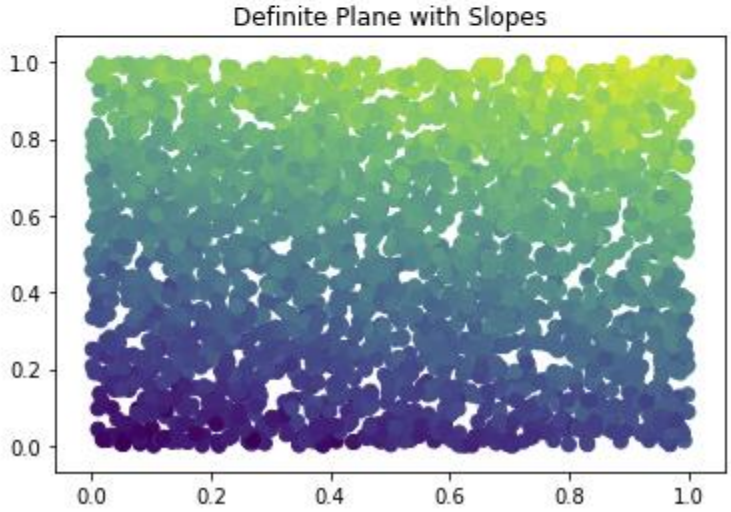
The implementation is the same as for the Perceptron Learning Algorithm with one exception. In stead of counting Boolean right or wrong answer, the average error was tracked. However, the termination logic is that when there is no further changes to the weights, the process stops.

Step 1: Visualize Input

The input data appears to be a cohesive plane, with a major slope on the vertical axis and on the horizontal slope consisting a more gentle slope.

```
Rows Loaded: 3000
```





Step 2: Calculate Model

The weights for the two independent variables are continually adjusted until there is no further changes. The average error is tracked, but it is not involved in the termination logic.

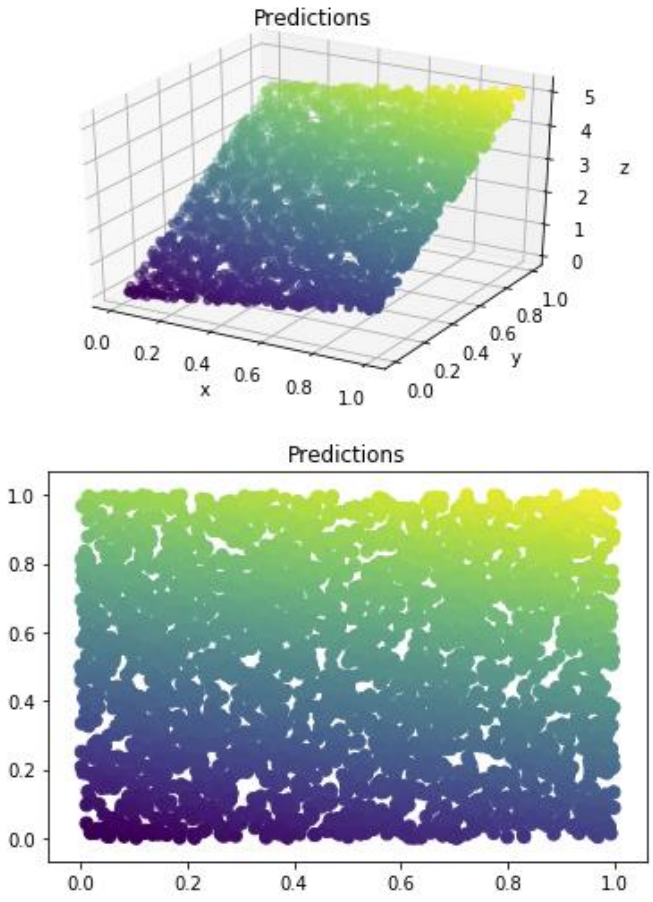
```
Calculating Model
        Iterations: 10001
        Learning Rate: 2e-05

Iteration: 0 Weights: [0.07930705 0.09405147] Average Error: 2.502176201443116
Iteration: 1000 Weights: [1.10704278 3.99541913] Average Error: 0.0021659920849672575
Iteration: 2000 Weights: [1.0983936  4.00407519] Average Error: 0.0021378856147888457
Iteration: 3000 Weights: [1.09834156 4.00412727] Average Error: 0.0021377165041323166
Iteration: 4000 Weights: [1.09834125 4.00412758] Average Error: 0.0021377154866218094
Iteration: 5000 Weights: [1.09834125 4.00412758] Average Error: 0.0021377154804391343
Iteration: 6000 Weights: [1.09834125 4.00412758] Average Error: 0.00213771548052454
```

Step 3: Visually Verify Accuracy

The predicted values are nearly identical to the original data.

```
Break:  No change in weights.
Iteration: 6046 Weights: [1.09834125 4.00412758] Average Error: 0.0021377154805266294
```



Code

Like all other programs, this program is a collection of functions called from the main routine.

```
# main:  All processing is controlled here.

# Print identification data.
if True:
    preamble()

# Load training data and initialize differences.
(X,L,W) = load_data()

# Check visually if the data could be bifurcated.
if True:
    plot_data_3d(X,L)
    plot_data_2d(X,L)

# Calculate the weights.
# X:  Training data.
# L:  Training labels.
# W:  Weights.
# I:  Maximum iterations allowed.
# R:  Learning Rate
if True:
    I = 10001
    R = 0.00002
    calculate_model(X,L,W,I,R)

plot_predictions_3d(X,W)
plot_predictions_2d(X,W)

print(' ')
print('Done!')
```

Part 2: Software Familiarization

Abstract The classification, logistic, and linear regression tasks were run using the sklearn modules. The numerical results were very close. However, the sklearn modules ran in a fraction of the time. This time difference could be attributed to the overfitting done by the implementations in order to get the maximum accuracy and fit.

sklearn

The following are the common steps used in each of the following *sklearn* executions.

1. Chose the model that corresponded to the task (classification, logistic, linear regression).
2. Train the model with training and label datasets.
3. Loop through the training data, calculating predictions and then comparing the predictions to the labels.

Perceptron

The implementation was slightly more accurate than *sklearn*. This is likely due to overfitting and adjusting parameters whereas *sklearn* was executed with default hyperparameters and execution parameters.

```
===== Perceptron =====
Negative Values: 1355 0.68
Positive Values: 645 0.32
Total Values: 2000

Perceptron using Sklearn:
        Accuracy: 0.98

Perceptron from Scratch:
        Accuracy: 0.99
```

Logistics

In this case, *sklearn* was slightly better.

```
===== Logistic =====
Negative Values: 1012 0.51
Positive Values: 988 0.49
Total Values: 2000

Logistic Classification using Sklearn:
        Accuracy: 0.53

Logistic Classification from Scratch:
        Accuracy: 0.5235
```

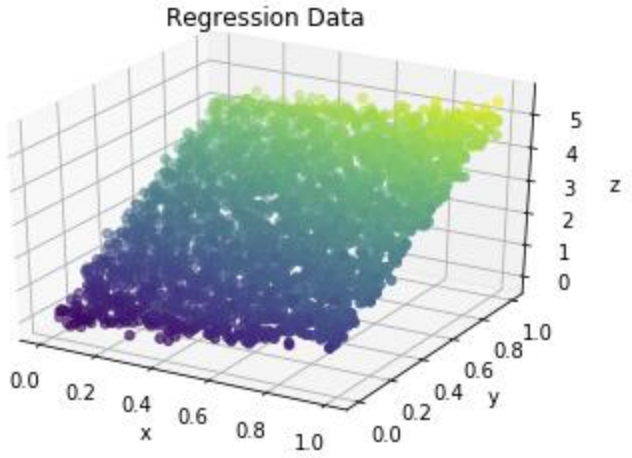
Linear Regression

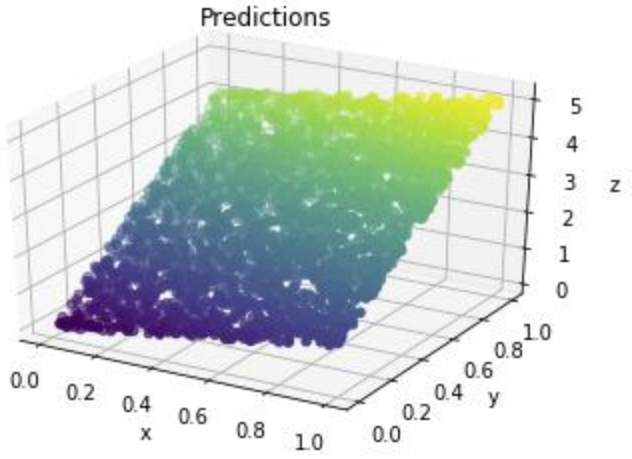
Numerically, the implementation version was much more accurate than the *sklearn* version. However, visually the difference is imperceptible. Again, the implementation parameters were adjusted whereas the *sklearn* version used defaults.

```
===== Linear Regression =====
Rows Loaded: 3000

Linear Regression Using Sklearn:
        Total Error: [476.23904691]
        Average Error: [0.15874635]

Linear Regession from Scratch:
        Total Error: 6.4131464415798884
        Average Error: 0.0021377154805266294
```





Part 3: Applications

Any application with sufficient training data can make use of these algorithms.

1. **Medical Diagnosis** There was a case study where pattern recognition using a neural network was faster and in most cases more accurate than humans. This is true in detecting skin cancer and x-ray interpretation.
2. **Media Restoration** of color to monochrome photographs and videos. There is a software package known as "Let There Be Color" that does this.
3. **Pixel Enhancing Developed** at Google, "Pixel Recursive Super Resolution" predicts how a human face would appear from a highly pixelated picture.
4. **Image Generation** "PIX2PIX" generates images based on input. Some applications include making a map from an aerial photograph or a photo-realistic image for a sales catalog from an outline.
5. **Lip Reading** "LipNet: provides a lip-reading service. It can watch a silent movie and convert mouth movements directly to text.