Minh Tran – HW4: Perceptron – Pocket – Logistic Algorithm – Linear Regression

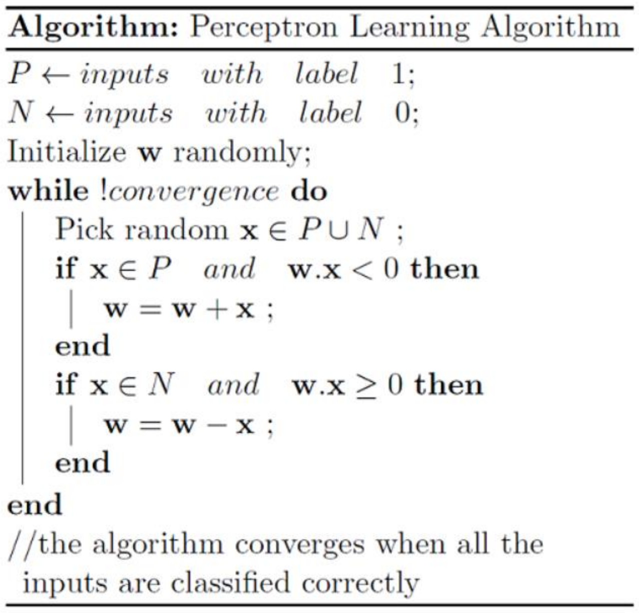
1. **<u>Part 1: Implementation:</u>**

a. **Perceptron Algorithm:**

- <u>Libraries</u>:

i. Pycharm IDE is used with Python 3.6 as the programming language. Basic libraries such as numpy and matplotlib are used to facilitate the program

- <u>Algorithm</u>:



```
Algorithm: Perceptron Learning Algorithm
$P \leftarrow inputs \quad with \quad label \quad 1;$
$N \leftarrow inputs \quad with \quad label \quad 0;$
Initialize $\mathbf{w}$ randomly;
while $!convergence$ do
    Pick random $\mathbf{x} \in P \cup N$ ;
    if $\mathbf{x} \in P \quad and \quad \mathbf{w}.\mathbf{x} < 0$ then
        $\mathbf{w} = \mathbf{w} + \mathbf{x}$ ;
    end
    if $\mathbf{x} \in N \quad and \quad \mathbf{w}.\mathbf{x} \geq 0$ then
        $\mathbf{w} = \mathbf{w} - \mathbf{x}$ ;
    end
end
//the algorithm converges when all the
 inputs are classified correctly
```

- <u>Data structure</u>:

i. Input data is imported by np.loadtxt as this gives the flexibility of which data columns to import

ii. A class called Perceptron is built with the following features:

✓ Initialization: weight, learning rate and maximum number of iterations

✓ Method “train”: given input data X and y, perform the weight update for each encounter of misclassification.

✓ Method “predict”: given an instance X, perform prediction to get y_predicted

iii. The following metrics are tracked to report at the end:

✓ numError: tracking the error at each weight update round to observe the trend of error

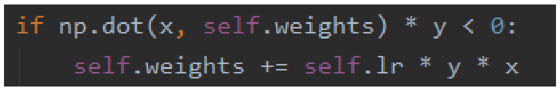
✓ iter: tracking which iteration is taking place

iv. Visualization includes the behavior of error count with the number of iterations as well as the apparent hyperplane to divide the input space.

v. Files included:

- ✓ perceptronMT.py - Contains the implementation of perceptron from scratch
- ✓ classification.txt – Input data.
- Code-level optimization:

i. Instead of performing a weight update step like shown in the previous figure, I use a one-liner:



```
if np.dot(x, self.weights) * y < 0:
    self.weights += self.lr * y * x
```

ii. After each misclassification instance, a weight update is performed and that instance is not visited again until new iteration commences. After all misclassification examples are visited, the updated weights are checked by tracking the number of instances correctly classified.

iii. Np.sign is used to quickly recognize the class (+1 or -1) in the "predict" method.

iv. Np.random.seed is used to control the randomization to obtain repeatability of the algorithm.

- Challenges:

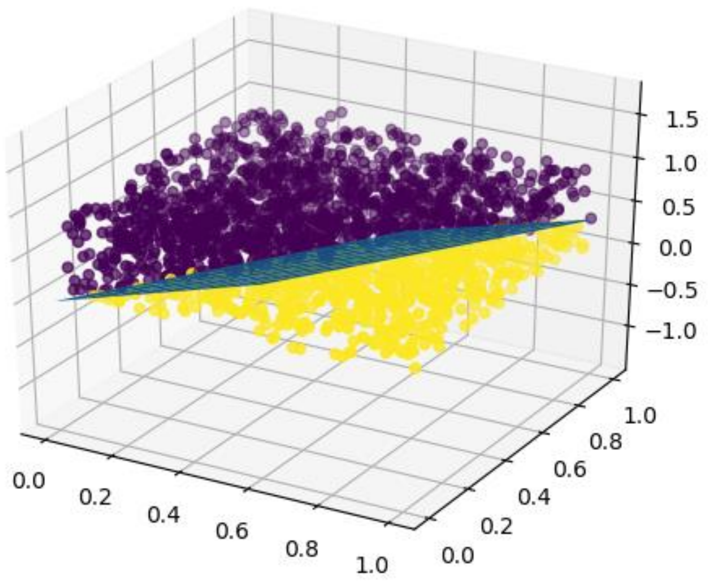
i. Tuning the learning rate may take a lot of time.

- Results:

i. Using the learning rate of 0.01 and max number of iteration at 1000, the final perceptron weight is:

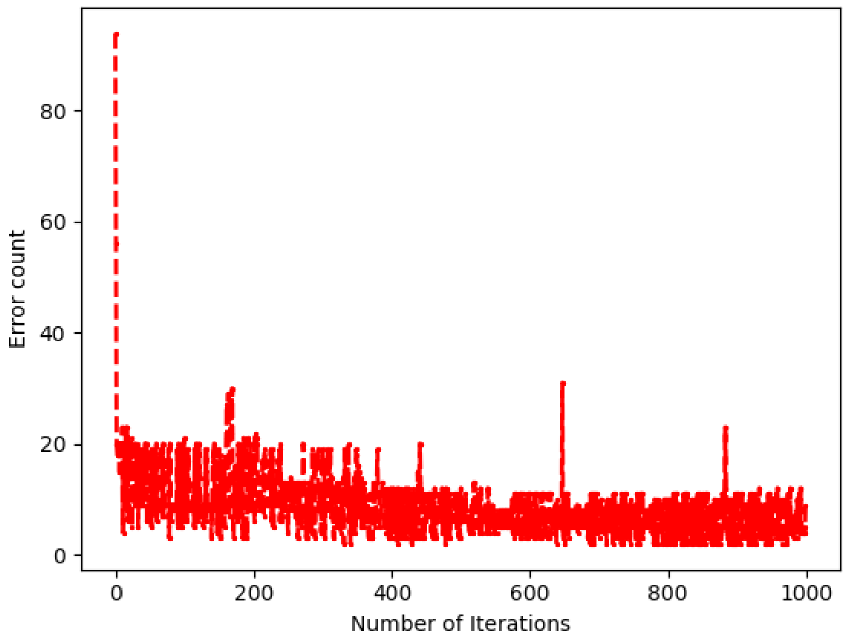
- ✓ W0 = -0.00369171
- ✓ W1 = 0.90260008
- ✓ W2 = -0.72107283
- ✓ W3 = -0.54593389

ii. The final hyperplane to dissect the input space is plotted:



iii. The accuracy on the train dataset is at 99.55%, which means the final error count is 9 out of 2000 instances

iv. The error count is plotted against number of iterations:



b. Pocket Algorithm:

- Algorithm:

i. The only difference of pocket algorithm from perceptron is to keep track of the iteration with the least error count, only updating the weights if it improves/lowers the error count.

- Data structure: (similar to Perceptron Learning Algorithm)

i. A class called Pocket is built with the features similar to Perceptron

ii. The following metrics are tracked to report at the end:

✓ bestErrorCount: tracking the best/lowest error count

✓ bestWeights: tracking corresponding weights

✓ best_iteration_number: tracking the iteration number

iii. Files included:

✓ pocketMT.py - Contains the implementation of pocket algorithm from scratch

- Results:

i. Using the learning rate of 0.01 and max number of iteration at 7000, the final perceptron weight is: (happening at iteration number 4420)

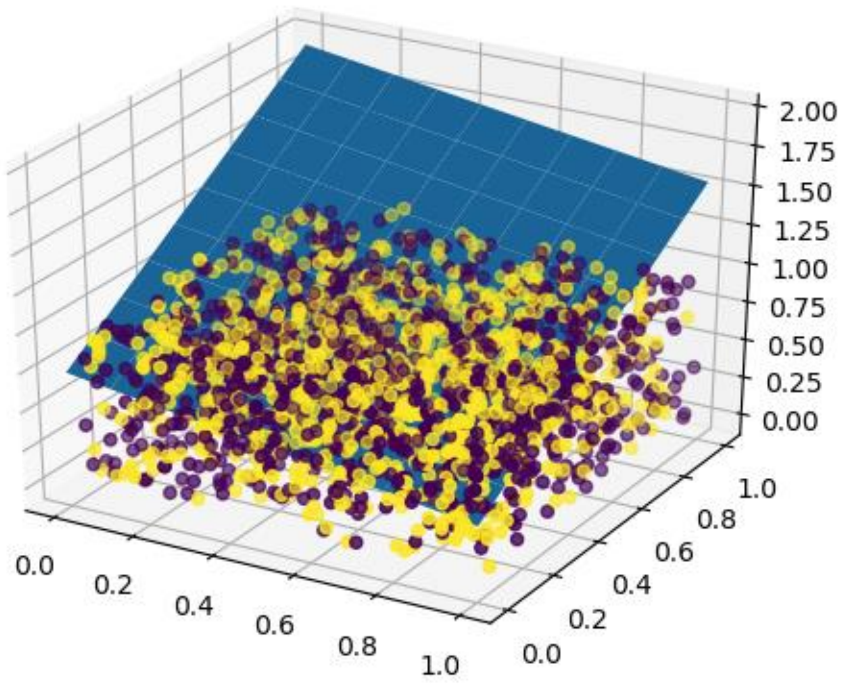
✓ W0 = -0.00630829

✓ W1 = -0.00601208

✓ W2 = 0.00175455

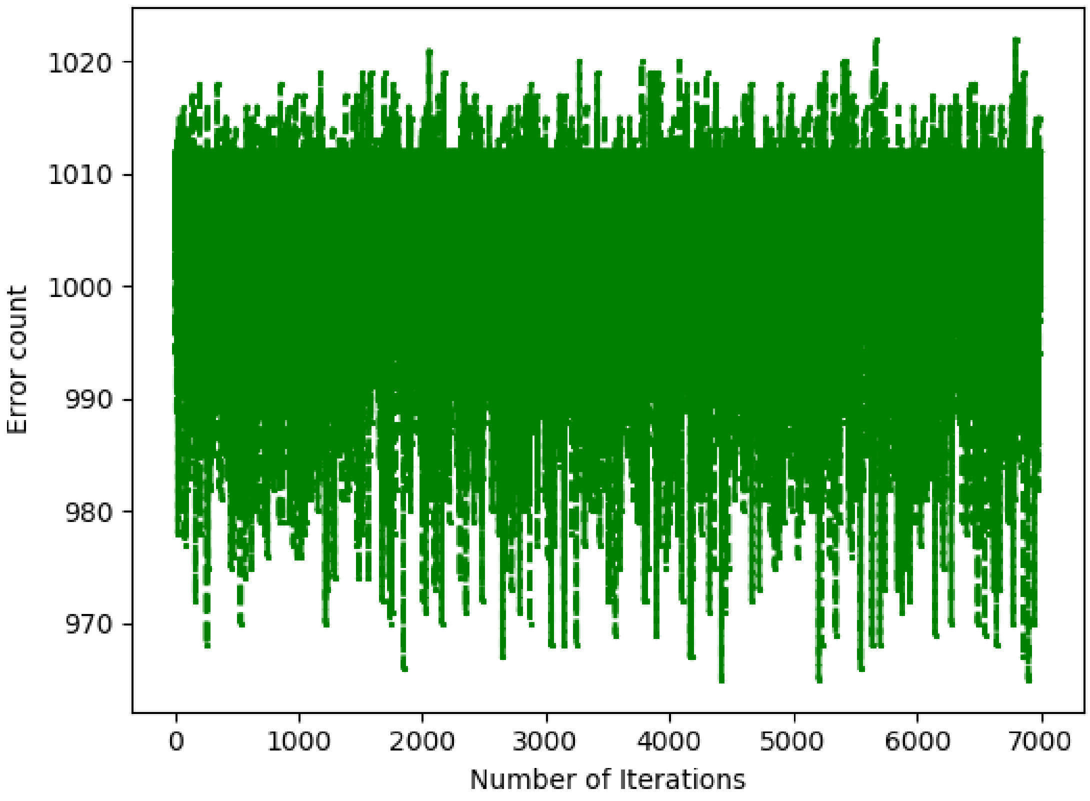
✓ W3 = -0.00635604

ii. The final hyperplane to dissect the input space is plotted:



iii. The accuracy on the train dataset is at 51.75%. The accuracy is very low because dataset is not linearly separable.

iv. The error count is plotted against number of iterations:


1020
1010
1000
990
980
970
Error count
0
1000
2000
3000
4000
5000
6000
7000
Number of Iterations

c. Logistic Regression Algorithm:

- Algorithm:

find the minimizer of

$$F(w) = \sum_{n=1}^{N} \ell_{\text{logistic}}(y_n w^{\mathrm{T}} x_n)$$
$$= \sum_{n=1}^{N} \ln(1 + e^{-y_n w^{\mathrm{T}} x_n})$$

Weight update is performed by:

$$w \leftarrow w - \lambda \nabla F(w)$$

$$: w + \lambda \, \mathbb{P}(-y_n \mid x_n; w) y_n x_n$$

$$\mathbb{P}(y \mid x; w) = \sigma(y w^{\mathrm{T}} x) = \frac{1}{1 + e^{-y w^{\mathrm{T}} x}}$$

- Data structure:

i. A global function called calc_sigmoid is built to calculate the sigmoid function given the input.

ii. A class called LogisticRegression is built with the following features:

✓ Initialization: weight, learning rate, error count and maximum number of iterations

✓ Method "train": given input data X and y, perform the weight update based on the gradient of the loss function.

✓ Method "predict": given an instance X, perform prediction to get y_predicted

iii. The following metrics are tracked to report at the end:

✓ numError: tracking the error at each weight update round to observe the trend of error

iv. Files included:

✓ logisticRegressionMT.py - Contains the implementation of perceptron from scratch

- Code-level optimization:

i. The gradient descent is cumulatively added after visiting each training instance then average out

- Challenges:

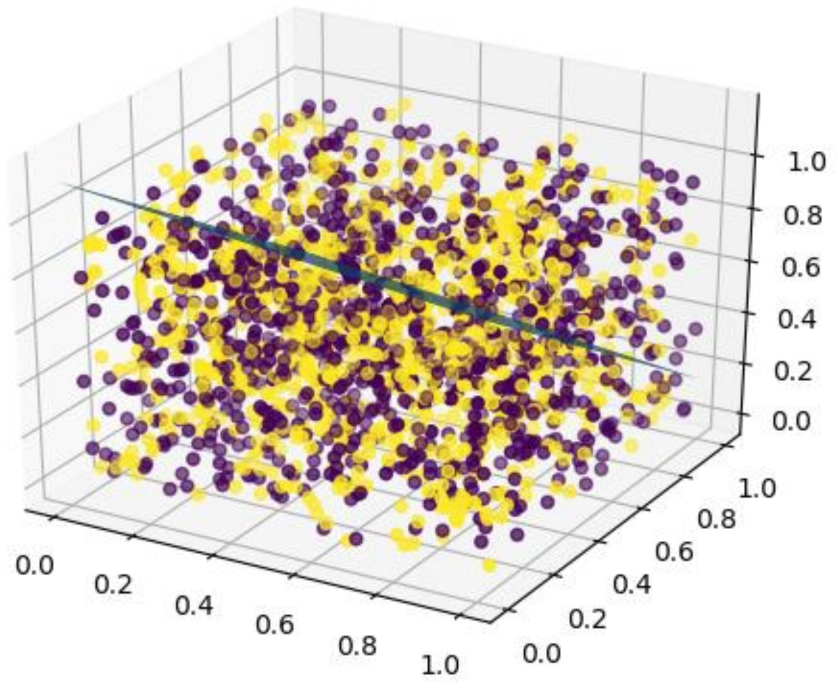
i. Using the 5th column as input data results in the very low accuracy because as seen in the 3d plot, data is not linearly separable.

- Results:

i. Using the learning rate of 0.01 and max number of iteration at 7000, the final perceptron weight is:

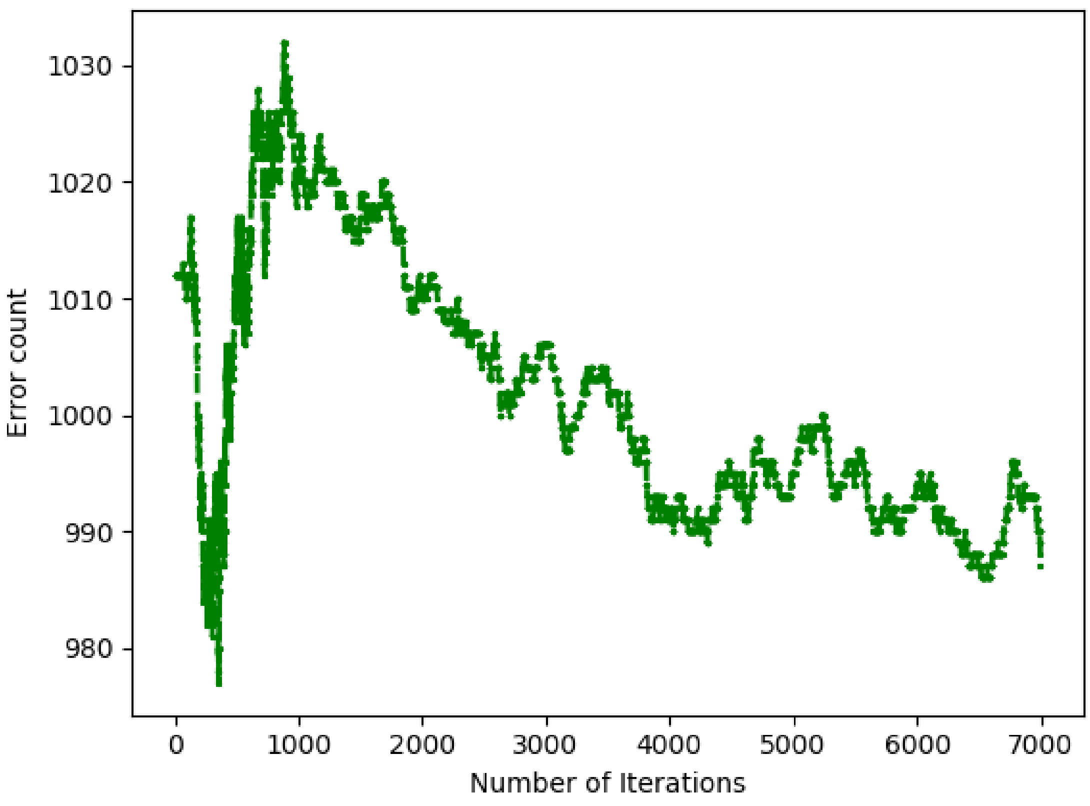
- ✓ W0 = -0.25817479
- ✓ W1 = 0.03925513
- ✓ W2 = 0.18347739
- ✓ W3 = 0.22611178

ii. The final hyperplane to dissect the input space is plotted:



iii. The accuracy on the train dataset is at 50.65%, which means the final error count is 987 out of 2000 instances

iv. The error count is plotted against number of iterations:


Error count
1030
1020
1010
1000
990
980
0
1000
2000
3000
4000
5000
6000
7000
Number of Iterations

d. Linear Regression Algorithm:

- Algorithm:

$$w^* = (X^{\mathrm{T}}X)^{-1}X^{\mathrm{T}}y$$

- Data structure:

i. Input data is imported by np.loadtxt as this gives the flexibility of which data columns to import

ii. A class called LinearRegression is built with the following features:

✓ Initialization: weight

✓ Method "train": given input data X and y, calculate the weights

iii. Files included:

✓ linearRegressionMT.py - Contains the implementation of perceptron from scratch

✓ linear-regression.txt – Input data.

- Code-level optimization:

i. Correct dimensions are required: W = (X.T * X)^-1 * X.T * Y

- Challenges:

i. If the covariance matrix is invertible, then we need to add a regularization term

$$w^* = \left(X^{\mathrm{T}}X + \lambda I\right)^{-1}X^{\mathrm{T}}y$$

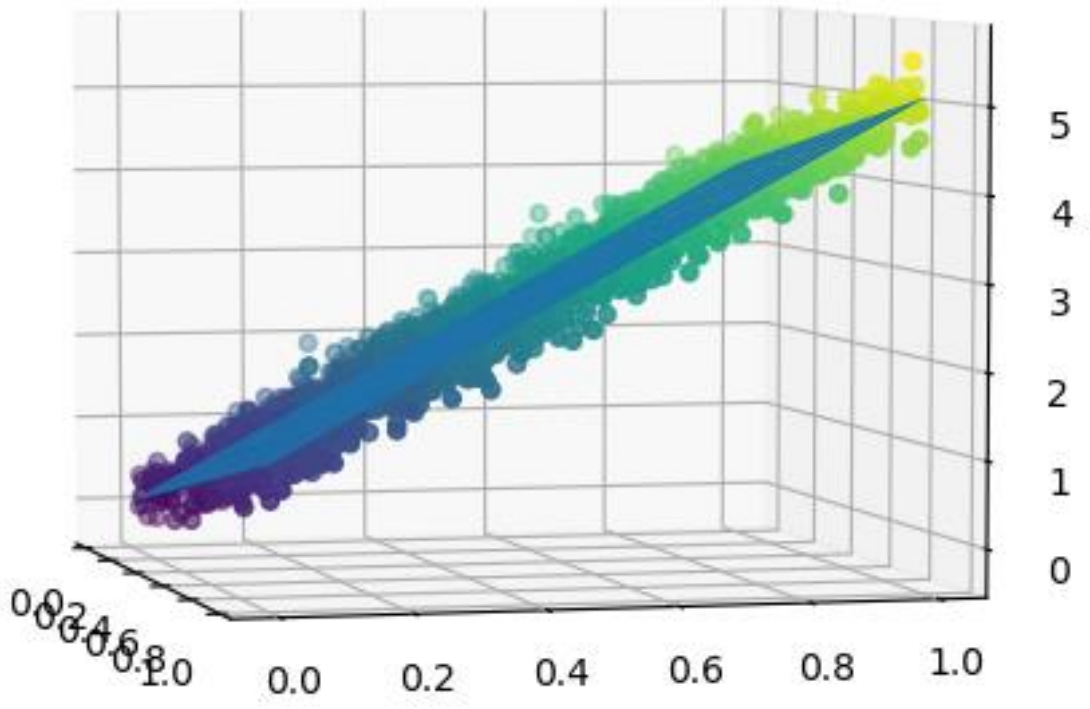
- Results:

i. The final linear regression weights:

✓ W1 = 1.09833772

✓ W2 = 4.00413136

ii. The final regression plane



0.0
0.2
0.4
0.6
0.8
1.0
0.0
0.2
0.4
0.6
0.8
1.0
0
1
2
3
4
5

2. **<u>Part 2: Software familiarization</u>**

a. **Sklearn for Perceptron, Logistic Regression and Linear Regression:**

✓ Scikit-Learn offers useful built-in packages to implement perceptron, logistic regression and linear regression.

✓ 3 simple steps are needed:

- Import data
- Create model and fit to data:
- Run command: python <script> <data file>

✓ Output: the number of iteration is significantly lower because the stopping criteria are enforced.

Method	Number of iterations	Accuracy
Perceptron	9	98.2%
Logistic Regression	6	99.2%
Linear Regression	N/A	N/A

Method	W0	W1	W2	W3
Perceptron	1	16.96657342	-12.7546922	-10.10797002
Logistic Regression	1	10.72693617	-8.2837886	-6.32984014
Linear Regression	N/A	1.08546357	3.99068855	N/A

3D data is shown below:

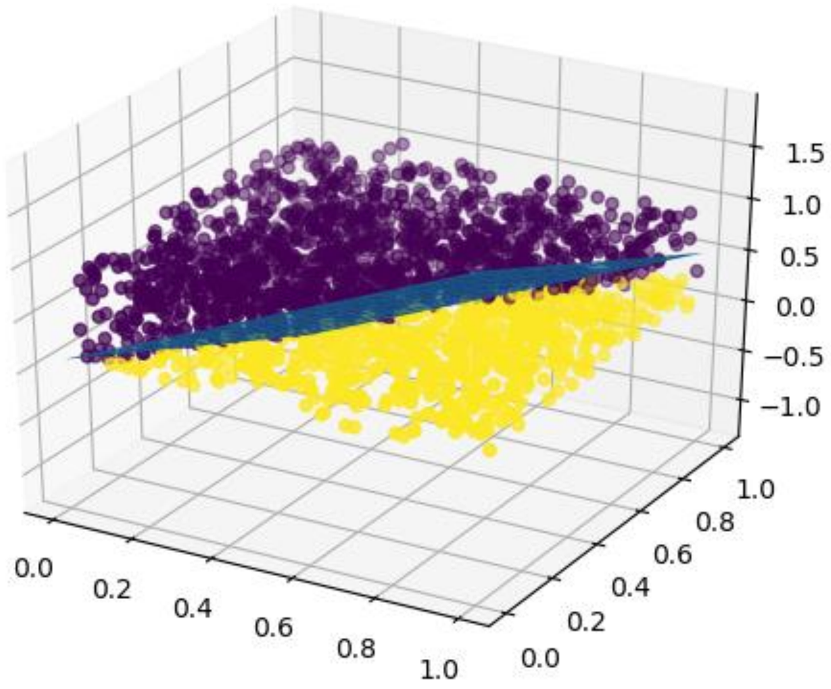


Figure 1: SKlearn implementation of Perceptron

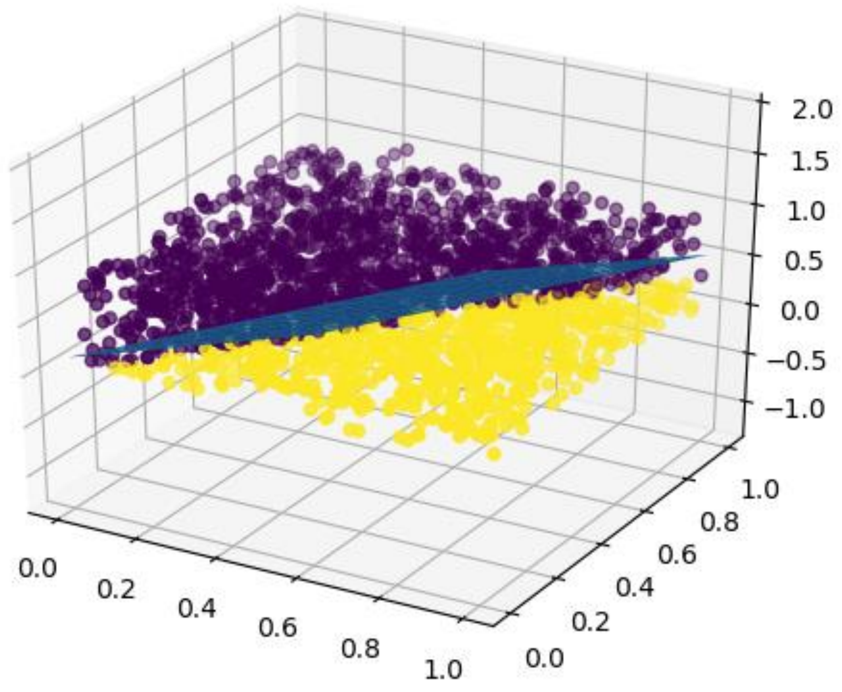


Figure 2: SKlearn implementation of Logistic Regression

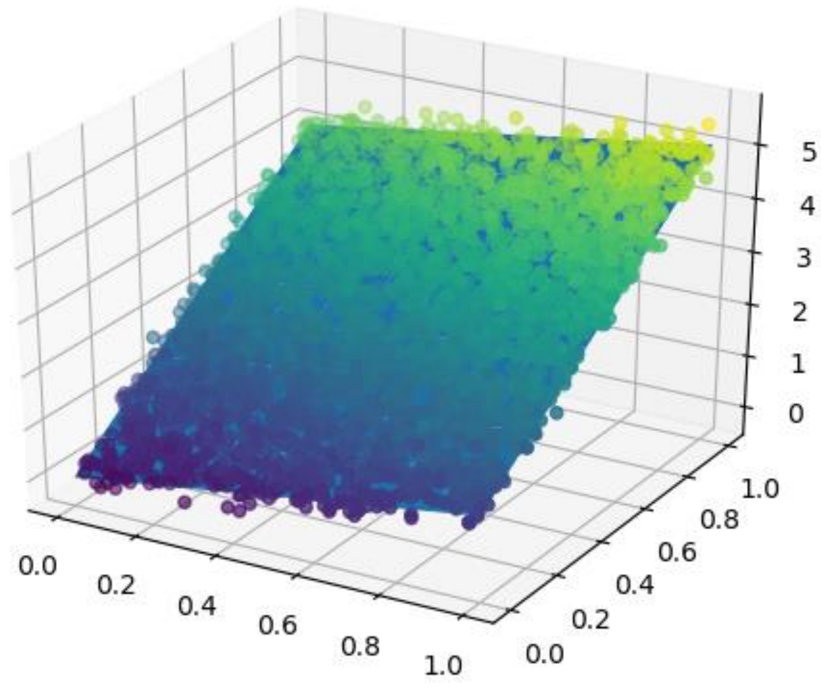


Figure 3: SKlearn implementation of linear regression

3. Applications

Perceptron is the basic form of a more complicated artificial neural network, which can be used for vision processing, object identification and classification, photo tagging, search engine, targeted marketing.

Applications of Logistic regression can be found in various fields, including machine learning, medical fields, and social sciences. The examples include mortality prediction, patient severity in treatment, disease-prone probability, political district preference. In engineering, logistic regression can be used for failure prediction of a given process, system or product. It is also used in marketing applications such as user's propensity in a subscription. In economics it can be used to predict the worker's habits, credit evaluation.

Applications of Linear regression can be found in finance (capital asset pricing model), economics (consumption spending, fixed investment, inventory investment), epidemiology (tobacco smoking mortality), environmental science (ecosystem change), machine learning.