CMPE 462 Assignment 1

Department of Computer Engineering

Bogazici University

Spring 2024

Yigit Sekerci - 2019400042

Hakan Emre Aktas - 2020400351

Mehmet Suzer – 2019400213

The implementation of all parts can be found in the following repository.

https://github.com/hakanaktas0/Basic-ML

Perceptron Learning Algorithm

How to Run

To run the PLA code use the following command,

- python PLA.py [options]

The options that can be used can be seen using -h or --help.

The arguments are,

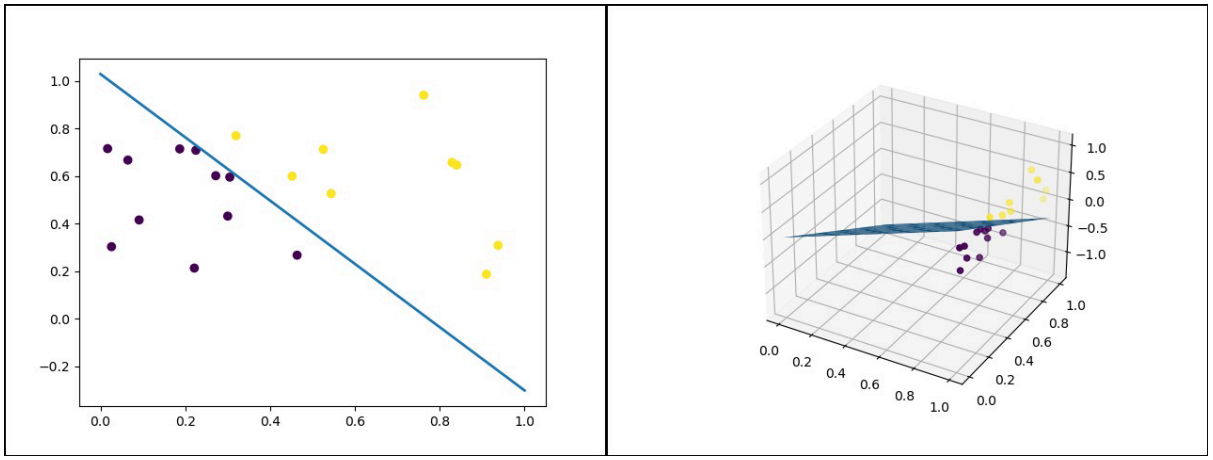
- --data_location : specifies where the data is. **required**.
- --data_size : specifies which dataset to use. *small* or *large* can be used. Default is small
- --initialization : specifies how to initialize the PLA. zero or random can be used
- --plot : if used, plots
- --visualization3D : if used, plots 3D
- --savefig : if used, saves the plot

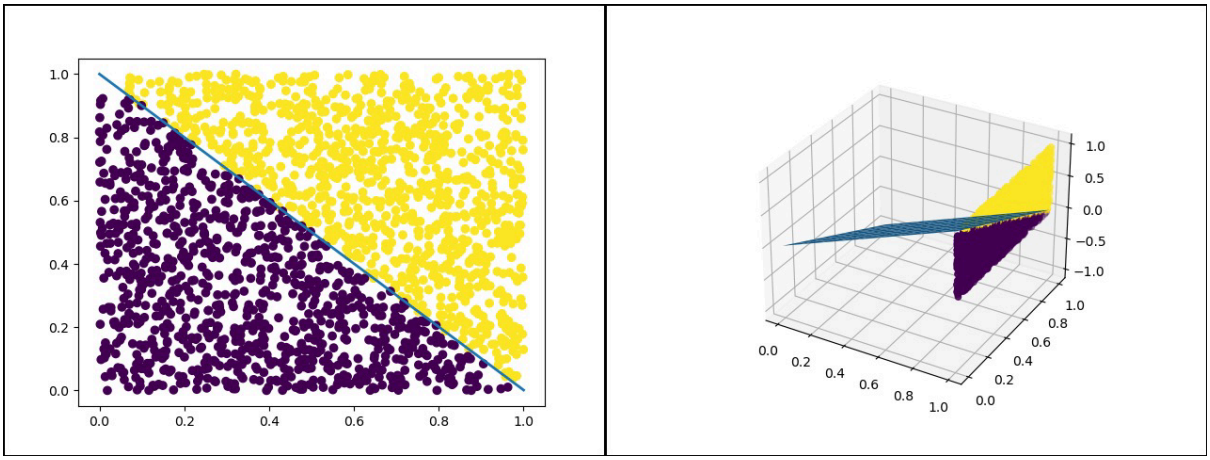
When the weights are initialized to 0s, the number of iterations to converge is

- 21 for the small dataset
- 4294 for the large dataset

Plots

The data were 3 dimensional, but one of the dimensions were always the same, so I include both 3D and 2D plots. The decision boundaries are drawn in blue. There are some intersections but they are caused by the size of the line/dots. Plots for the small and the large datasets can be seen below.

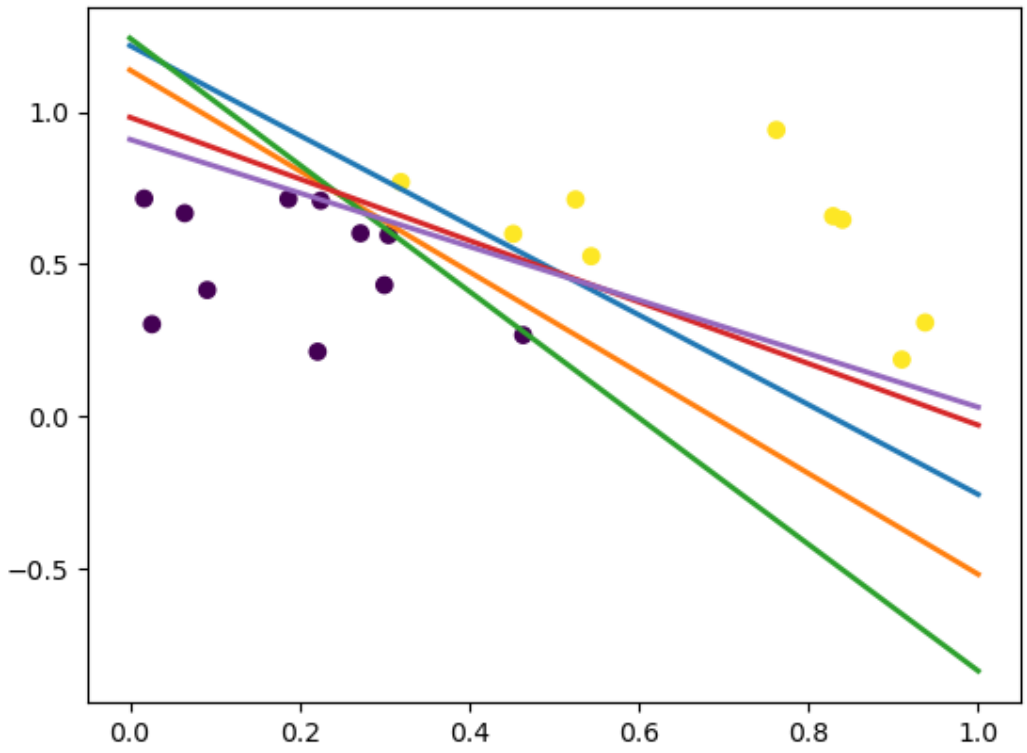




Initialization

Since the parameters of the perceptron are adjusted gradually, in other words, learned, their initial values are important and affect the end result. There is more than one solution to the linear boundary problem and in the case of PLA the found solution depends on the initial parameters.

Several different found decision boundaries for the small dataset can be seen below.



Different solutions for random initialization can be visualized by running:

- python PLA-decision.py [options]

The options can be seen by using -h or –help.

Not only the initialization affect the solution, but also affects the number of iteration the algorithm takes to find solutions. When run 100 times with different random initializations, the mean number of iterations and the standard deviations are

- 20.06 and 8.79 for the small dataset
- 4289.28 and 293.87 for the large dataset

The code used to obtain these results can be executed using the following,

- python PLA-random-test.py [options]

The options can be seen by using -h or –help. For both PLA-random-test and PLA-decision, the options are,

- --data_location : specifies where the data is. **required**.
- --number_of_runs : specifies the number of runs. Default is 5 for the PLA-decision and 100 for the PLA-random-test.

Logistic Regression

Preparation

In Rice (Cammeo and Osmancik) dataset, morphological features of 3810 different rice grains are listed. The recorded features are as follows:
1- Area
2- Perimeter
3- Major Axis Length
4- Minor Axis Length
5- Eccentricity
6- Convex Area
7- Extent

The target of the dataset is the class of the grains, which are either Cammeo or Osmancik. Since some of features like Area have a magnitude of 10^4 and some features like Eccentricity are between 0 and 1, we cannot directly use these values in our logistic regression model. Those which have significantly higher values will overshadow those having smaller values. In order to overcome this issue, we need to normalize the dataset. There are many normalization techniques such as linear normalization and Gaussian normalization. We preferred to use linear one since it is easier to implement. With linear normalization, all values are mapped to the range [0,1].

There are other benefits of normalization, as well. Firstly, normalization solved a runtime warning that we got while calculating the large powers of numbers, which prevented getting NaN or INFINITY for our weights. Secondly, calculating large powers of a number is a costly operation. Therefore, normalization helped us to improve the performance of our model.

In addition to normalization, we shuffled the data in order to get results depending on the order of the instances in the dataset. Also, the we added the bias term to the weight vector.

For 1000 epochs, we get the following results:

Normalized...
Shuffled...
Bias is added...
GD train time: 7.69 s
GD accuracy: 93.04 %
SGD train time: 13.96 s
SGD accuracy: 93.04 %

Shuffled...
Bias is added...
GD train time: 9.87 s
GD accuracy: 42.78 %
SGD train time: 16.14 s
SGD accuracy: 49.34 %

How to Run

Command the run the code: python3 logreg.py
With epoch number: python3 logreg.py <number_epochs>
And normalization: python3 logreg.py <number_epochs> norm
And shuffling: python3 logreg.py <number_epochs> norm shuffle
And bias: python3 logreg.py <number_epochs> norm shuffle bias

The order of “norm”, “shuffle”, and “bias” is not important, while <number_epochs> must be the first argument after logreg.py. In order to test the code, uncomment the last sections.

5-fold Cross Validation

In the code, there is function named **get_best_regularization_term(x, y, terms)**. The function splits the data into equally large blocks for each regularization term given in **terms**. The list of terms that we used is [0, e^{-18}, e^{-9}, e^{-6}, e^{-3}] . Each regularization terms is trained with 4 blocks, 80 % of the data, while tested with 1 block. The term that gives the best accuracy is returned. This procedure is done for both gradient descent (GD) and stochastic gradient descent (SGD). We get the following results for running GD and SGD for 500 epochs for each regularization term:

GD average accuracies [92.86, 92.88, 92.88, 92.83, 92.80]
SGD average accuracies [92.80, 92.86, 92.78, 92.25, 89.84]
GD regularization term = e^{-18}
SGD regularization term = e^{-18}

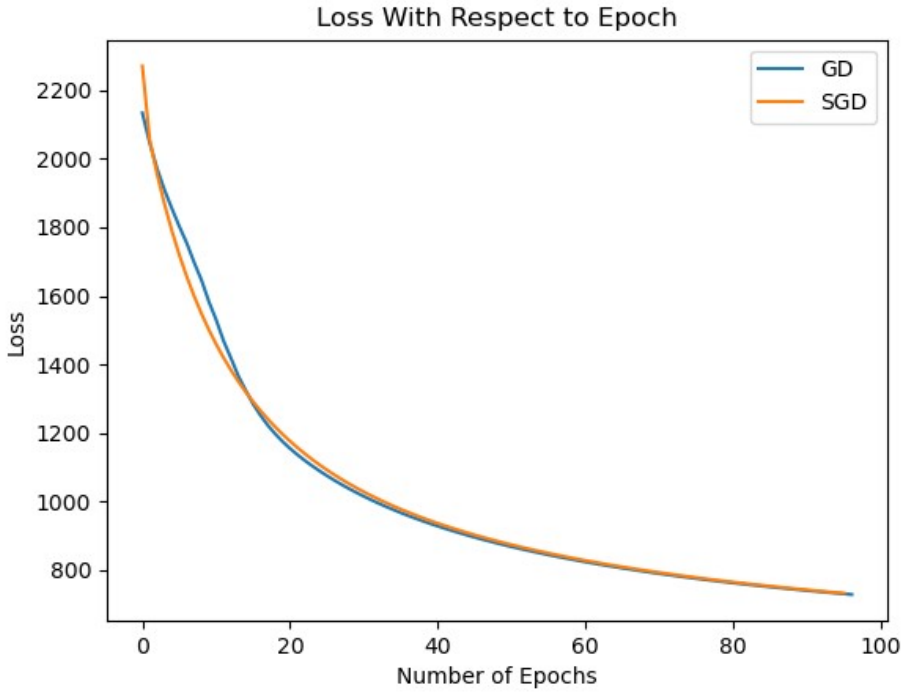
As a result, the best regularization term for GD and SGD is e^{-18}.

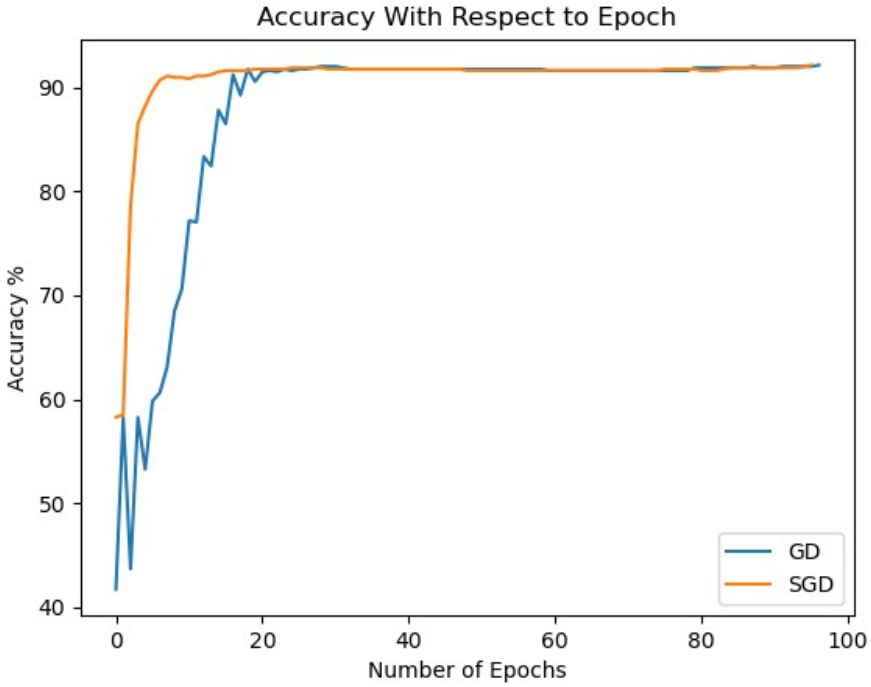
Comparison of GD (λ=0), GD (λ=e^{-18}), SGD (λ=0), and SGD (λ=e^{-18})

Model	Train Accuracy	Test Accuracy	Train Duration
GD (λ=0)	92.81 %	91.47 %	7.83 s
GD (λ=e^{-18})	92.85 %	91.47 %	7.64 s
SGD (λ=0)	92.85 %	91.34 %	13.58 s
SGD (λ=e^{-18})	92.85 %	91.34 %	13.75 s

Compare Training Times

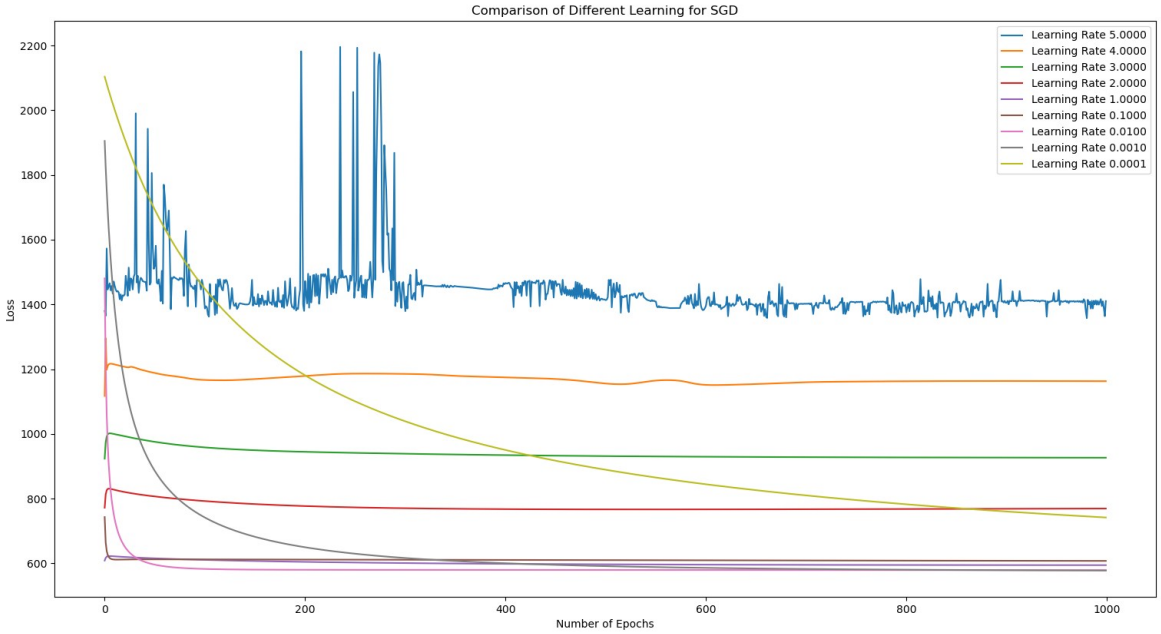
We set the target accuracy 92.0%. We selected this accuracy after a couple of tries. We saw that the accuracy of a model does not go beyond 92% most of the time; sometimes, it is as low as 89%. If you want to test the code for a target accuracy, it may stuck in an infinite loop because it may never go above 92.0 % accuracy. In such a case, try lowering the target accuracy or rerunning the program. The losses and accuracy plots of GD and SGD are as follows:

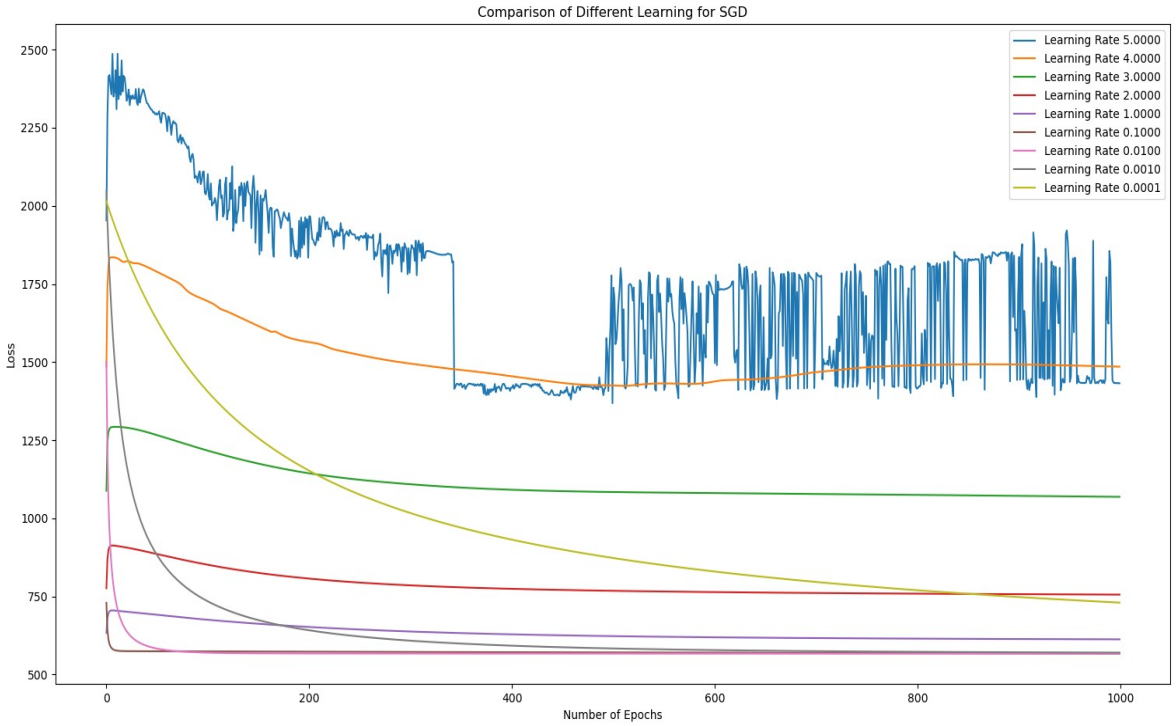




Compare Learning Rates for SGD

Our default learning rate for our models is 0.001. In order to compare different learning rates, we ran our SGD implementation with rates [5.0, 4.0, 3.0, 2.0, 1.0, 0.1, 0.01, 0.001, 0.0001]. Loss values during training of 2 runs can be seen in the following plot:





We saw that using a learning rate 0.1 or 0.01 yields a better result than other rates. The smaller learning rates require more time to reach lower loss values, while larger learning rates result in anomalies such as not training at all. Probably, the models oscillate around the optimum points with large steps; therefore, they miss the optimum weight.

Naive Bayes

How to Run

To run the Naïve Bayes code use the following command,

- Python naive_bayes.py [options]

The valid options can be seen using -h or --help.

The options are,

- --data_path : Path to the data file (**Required**)
- --train_proportion: Proportion of the dataset to include in the training set. Default is 0.8.
- --seed: Seed for the random number generator.
- --calculate_parameter_count: If present code also calculates the required parameter count.
- --repeat: Number of times to repeat the experiment. Default is 1.

The data_path given in options must be the path of the wdbc.data file. It is like a csv file without headers.

Accuracies

For the accuracy calculations, we have used the following options,

- Train proportion: 0.8
- Seed: 42
- Repeat count: 100.

With these options we have observed following accuracies (Mean of 100 repeat),

- Train accuracy: 93.84 %
- Test accuracy: 93.48 %

While we can get higher accuracies with certain seeds and splits, we chose to stick with one seed and repeat the experiment as much as we can.

Number of parameters

- Number of features: 30
- Number of classes: 2

Without conditional independence

We need covariance matrix for features and mean for every class. Since covariance matrix is symmetric, we only need one side and diagonal.

Number of parameters = Class count * (Mean count + (Covariance Diagonal and One side))

Number of parameters = 2 * (30 + (30 * 31 / 2)) = 990

Number of parameters required without conditional independence is 990.

With conditional independence

We need only mean and deviation of features for every class.

Number of parameters = Class count * (Mean count + Deviation count)

Number of parameters = 2 * (30 + 30) = 120

Number of parameters required with conditional independence is 120.

Comparison with Logistic Regression

Slightly updated version of logistic regression from logistic regression part in project is used. Training for logistic regression is done with following parameters:

- 1000 Epoch
- GD
- Regularization term: e^-18
- Seed: 0

Following accuracies observed with logistic regression,

- Train accuracy: 91.45 %
- Test accuracy: 91.15 %

Fundamental differences between models,

- While naïve bayes assumes feature are independent logistic regression does not. Thus, logistic regression model can capture relationships between features.
- Since logistic regression also tries to capture relationships between features it requires more data to estimate the parameters accurately.
- Naïve bayes has fewer parameters to estimate that's why it is very efficient in high dimensional data. On the other hand, with right epoch and regularization logistic regression can also generalize high dimensional data but at the cost of computational resources.
- Performance of Naïve Bayes heavily depends on the accuracy of the assumed distribution of features (Normal etc.).
- Logistic regression is more sensitive to outlier that's why it requires larger datasets.

Even if we assume that our data is normally distributed in Naïve Bayes and take conditional independence into account, Naïve Bayes' accuracies are slightly better than those of logistic regression. Thus, we can say that relationships between the features are not that important in this case. On the other hand, the sample count is very low (570) relative to the feature count (30), so logistic regression may not be able to estimate the parameters accurately. However, I believe the accuracies from both models are very good, and the difference between the models is very low. It is very hard to determine the reason behind the 2% accuracy difference.

Codes for Naïve Bayes Related Questions

https://github.com/hakanaktas0/Basic-ML/tree/main/NaiveBayes