Introduction

In this project I implement three algorithms: **Pocket PLA**, **Linear Regression** and **Logistic Regression**. We test these algorithms in two different datasets:

• Breast Cancer	- 2 Classes, 30 Features, 569 Samples
• Iris	- 3 Classes, 4 Features, 150 Samples

In first dataset, *Breast Cancer* I use Pocket PLA since it only has two classes and binary classification fits perfectly fine with it. I initialize the **W** weight using two different methods, in first method I take the values of features of first sample, in the second method I use Linear Regression algorithm to find the optimal **W**.

In second dataset, *Iris* I use Logistic Regression, for multiclass classification. Same as with pervious dataset, I use Linear Regression and values of first sample for initialization of the **W** weights.

Pocket PLA

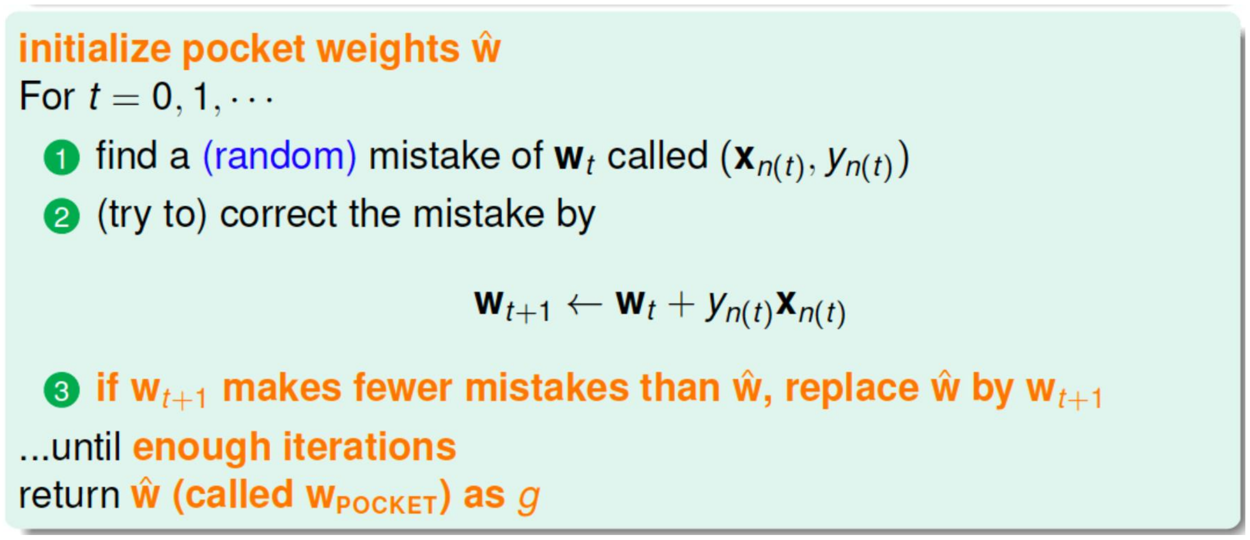
In this project we implement Pocket PLA. The reason for using Pocket PLA instead PLA is that it might be that the data is not linearly separable, in that case PLA will not halt, and that's a problem. To tackle this problem we implemented a modified version of PLA, which keeps the best **W** 'in the pocket', and it halts in a pre-defined number of iterations. This is the reason why it's called Pocket PLA. The linear model for classifying data in two classes uses a hypothesis set of linear classifiers where each h has the form:

$$h(x) = sign(w^T x)$$

where weight vector $w \in R^{d+1}$, and d is the dimensionality of the input space. This extra dimension includes bias, and it is usually set $x_0 = 1$. PLA algorithms has few steps:

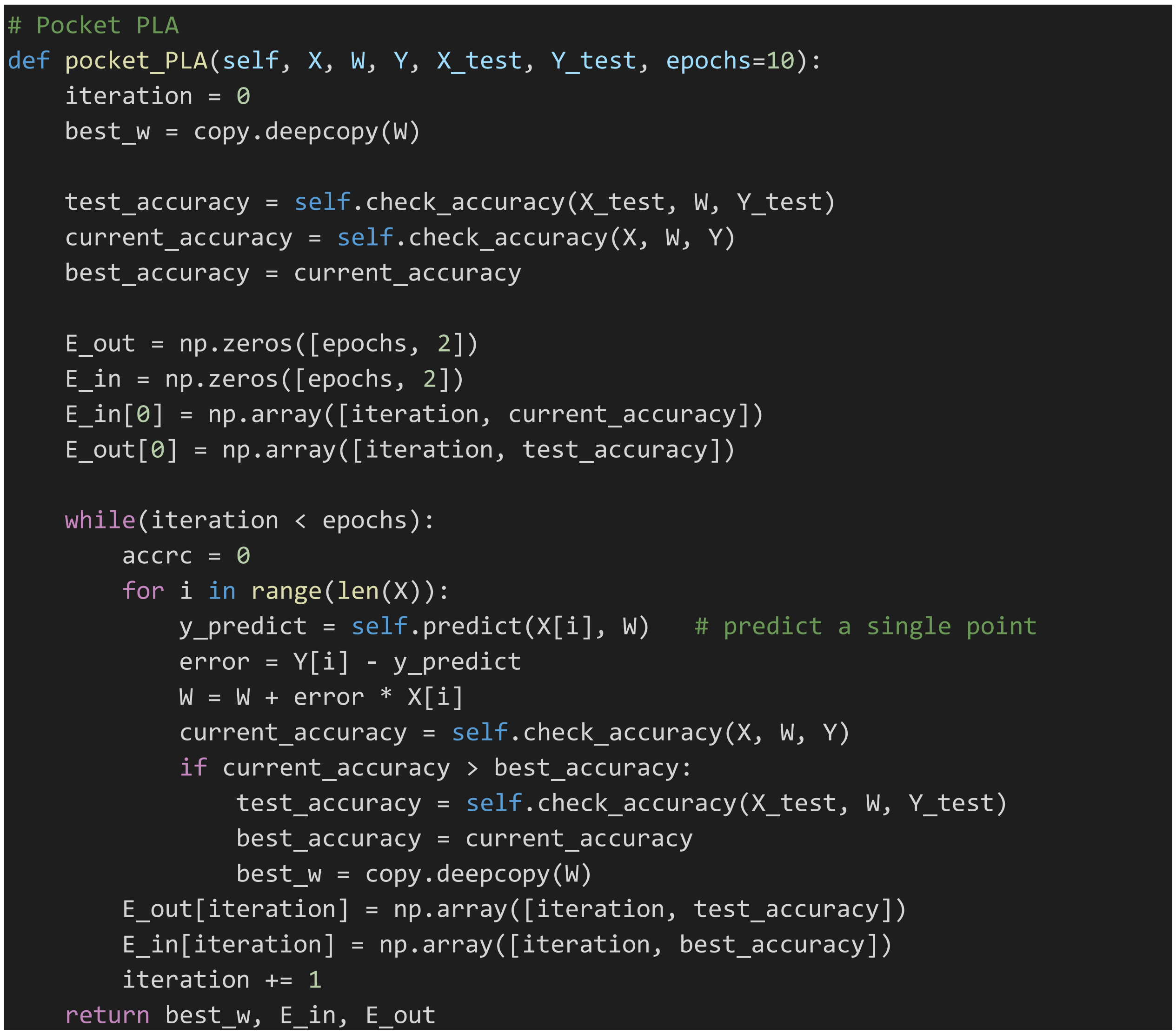
1. Initialize the initial weight **w**
2. Pick an example of the form (x_i, y_i) which is misclassified with the current **w**
3. Since the example is misclassified $y(t) \neq sign(\boldsymbol{w}(t)^T x(t))$
4. Repeat steps 2-3 until there are no misclassified examples.

Pocket PLA is almost identical with PLA, as we mentioned the only difference is that algorithm will not to correctly classify all the examples in the training dataset, but it will stall if the accuracy is 100% or the specified number of iterations are exceeded. The figure 1 describes the work flow of Pocket PLA. This picture is taken from class notes. Mathematically expressed, Pocket PLA will update the best $\hat{w} \leftarrow w_{t+1}$ only when $E_{in}(w_{t+1}) < E_{in}(\hat{w})$, where E_{in} is in-sample error.



```
initialize pocket weights $\hat{\mathbf{w}}$
For $t = 0, 1, \cdots$
    1 find a (random) mistake of $\mathbf{w}_t$ called $(\mathbf{x}_{n(t)}, y_{n(t)})$
    2 (try to) correct the mistake by
          $\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t + y_{n(t)}\mathbf{x}_{n(t)}$
    3 if $\mathbf{w}_{t+1}$ makes fewer mistakes than $\hat{\mathbf{w}}$, replace $\hat{\mathbf{w}}$ by $\mathbf{w}_{t+1}$
...until enough iterations
return $\hat{\mathbf{w}}$ (called $\mathbf{w}_{\text{POCKET}}$) as $g$
```

Fig 1. *Pocket Perceptron Learning Algorithm*



```python
# Pocket PLA
def pocket_PLA(self, X, W, Y, X_test, Y_test, epochs=10):
    iteration = 0
    best_w = copy.deepcopy(W)

    test_accuracy = self.check_accuracy(X_test, W, Y_test)
    current_accuracy = self.check_accuracy(X, W, Y)
    best_accuracy = current_accuracy

    E_out = np.zeros([epochs, 2])
    E_in = np.zeros([epochs, 2])
    E_in[0] = np.array([iteration, current_accuracy])
    E_out[0] = np.array([iteration, test_accuracy])

    while(iteration < epochs):
        accrc = 0
        for i in range(len(X)):
            y_predict = self.predict(X[i], W)   # predict a single point
            error = Y[i] - y_predict
            W = W + error * X[i]
            current_accuracy = self.check_accuracy(X, W, Y)
            if current_accuracy > best_accuracy:
                test_accuracy = self.check_accuracy(X_test, W, Y_test)
                best_accuracy = current_accuracy
                best_w = copy.deepcopy(W)
        E_out[iteration] = np.array([iteration, test_accuracy])
        E_in[iteration] = np.array([iteration, best_accuracy])
        iteration += 1
    return best_w, E_in, E_out
```

The above function returns E_{in} and E_{out} and that's the reason why as input it requires the test data also. Test data are used just to calculate E_{out}.

Linear Regression

In this project we use Linear Regression to initialize the weight vector, with a hope that it will improve the accuracy or make the algorithm converge faster. Linear Regression Algorithm starts by constructing X data matrix and Y output:

$$X = \begin{bmatrix} -- & x_1^T & -- \\ -- & x_1^T & -- \\ -- & \vdots & -- \\ -- & x_N^T & -- \end{bmatrix} \qquad \text{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix}$$

The LR is based on minimizing the squared error between hypothesis and output:

$$E_{out} = E[(h(x) - y)^2]$$

Where expected value is taken w.r.t. the joint PDF P(x,y). The goal of this algorithm is to find a good hypothesis that has a small error. PDF P(x,y) is unknown, we cannot use the formula above to measure error, instead we use the in-sample version of error:

$$E_{in}(h) = \frac{1}{N}\sum_{n=1}^{N}(h(x_n) - y_n)^2]$$

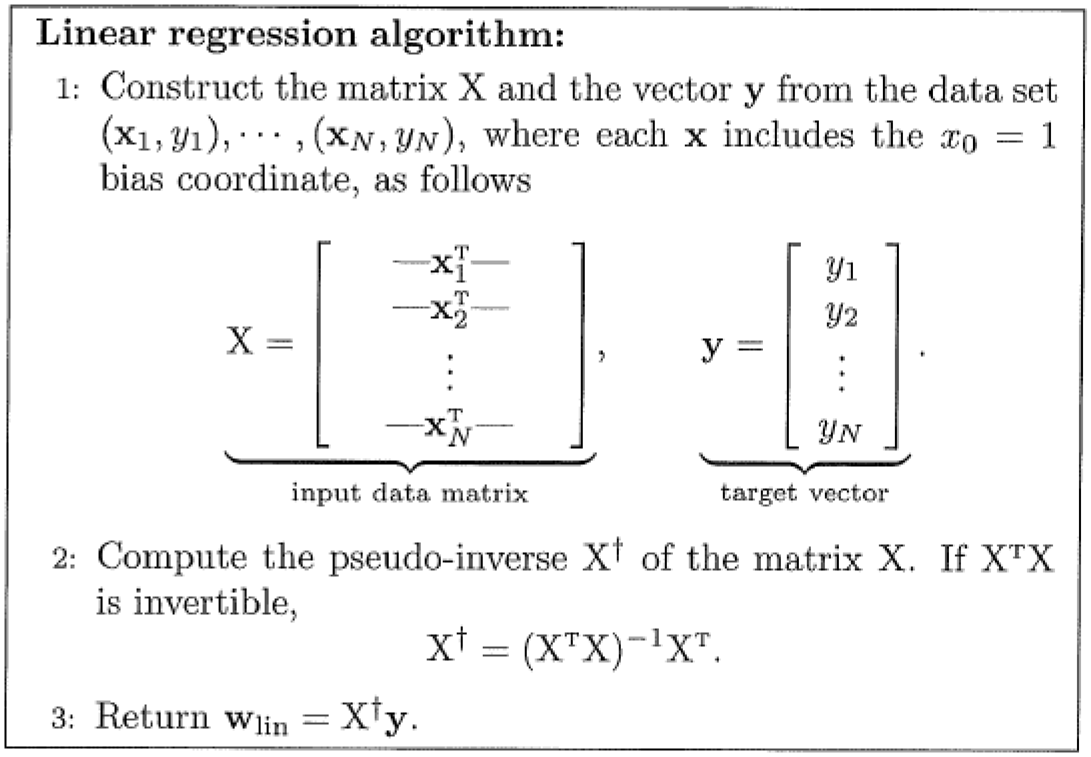
From elementary math we know how to find the minimum of this function, so we find the gradient of the below expression w.r.t. **w** vector and equal it to zero, so that way we can find our minimum **w**.

$$E_{in}(h) = \frac{1}{N}\sum_{n=1}^{N}(h(x_n) - y_n)^2] = \frac{1}{N}\|Xw - y\|^2$$

So this leads to:

$$X^T X w = X^T y$$

To solve this equation, we need to find the pseudo-inverse of the input matrix (with one added columns of 1s, to cover the bias), doing that we can calculate **w** vector, which has optimal values. This $X^T X$ matrix should be invertible, and it is in most of the cases, this is because of the dimensions of the matrix, which is usually very tall and thin matrix, and $N \gg d + 1$, so we most likely have d+1 linearly independent vectors x_n. Steps of this algorithm are shown in figure 2.



Linear regression algorithm:

```
1: Construct the matrix $X$ and the vector $\mathbf{y}$ from the data set
   $(\mathbf{x}_1, y_1), \cdots, (\mathbf{x}_N, y_N)$, where each $\mathbf{x}$ includes the $x_0 = 1$
   bias coordinate, as follows
```

$$\underbrace{X = \begin{bmatrix} —\mathbf{x}_1^{\mathsf{T}}— \\ —\mathbf{x}_2^{\mathsf{T}}— \\ \vdots \\ —\mathbf{x}_N^{\mathsf{T}}— \end{bmatrix}}_{\text{input data matrix}}, \qquad \underbrace{\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix}}_{\text{target vector}}.$$

```
2: Compute the pseudo-inverse $X^\dagger$ of the matrix $X$. If $X^{\mathsf{T}}X$
   is invertible,
```

$$X^\dagger = (X^{\mathsf{T}}X)^{-1}X^{\mathsf{T}}.$$

```
3: Return $\mathbf{w}_{\text{lin}} = X^\dagger\mathbf{y}$.
```

Fig. 2. *Linear Regression Algorithm.*

Implementation of this algorithm is very easy and it only takes a line of code using numpy functions to calculate the inverse matrix.

```
# Returns the weights calculated using Pseudo-Inverse
def linear_regression(self, X, Y):
    return (np.linalg.inv(X.T @ X) @ X.T) @ Y
```

Logistic Regression

This algorithm is a soft binary classification algorithm, which will return probability instead 0/1 classes. This makes this algorithm very useful for multi-class classification. When we use this algorithm for prediction we look for the highest probability instead the class, that's how we are able to classify an input. Logistic regression model is given by:

$$h(x) = \theta(w^T x)$$

where:

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

which gives an output between 0 and 1. This logistic function θ is a soft threshold, that's why it is very helpful because it gives probabilities.

Formally, we are trying to learn the target function $f(x) = P[y = +1|x]$. The data we have gives us samples of this f function. The data is generated by a noisy target $P(y|x)$:

$$P(y|x) = \begin{cases} f(x) & for\ y = +1 \\ 1 - f(x) & for\ y = -1 \end{cases}$$

To learn from data we need to define an error function. The standard error measure used in logistic regression is based on the notion of likelihood:

$$P(y|x) = \theta(y\, w^T x)$$

In this work we decided to use **cross-entropy** loss function which is given with:

$$J(\theta) = \frac{1}{N}\sum_{i=1}^{N}[-y^i \log\left(h_\theta(x^i)\right) - (1 - y^i)\log(1 - h_\theta(x^i))]$$

To find the optimal solution I used the gradient descent. From loss function we calculate the gradient which is given with:

$$\frac{J(\theta)}{\partial\theta} = \frac{1}{N}\sum_{i=1}^{N}(h_\theta(x^i) - y^i)\, x^i$$

and then weight update is given with:

$$\theta \leftarrow \theta - \alpha\frac{J(\theta)}{\partial\theta}$$

where α is the learning rate.

To classify multi-classes there are few strategies. I used One Vs All strategy. This strategy will take a label at the time, and make the other labels as one label, this helps because after you separate the data in that form, you can simply use binary classification. When we want to predict, we will have a set of weights, in order to know the output we need to use all generated weights and choose the output with the highest probability. The implementation of Logistic Regression is shown below:

```python
class LogisticRegression(object):

def __init__(self):
    None

def sigmoid(self, z):
    return 1/(1+np.exp(-z))

def hypothesis(self, X, W):
    z = np.dot(X, W)
    return self.sigmoid(z)
```

```
def cost_function(self, X, W, Y):
    predictions = self.sigmoid(X @ W)
    predictions[predictions == 1] = 0.9999999   # log(1) = 0
    gradient = X.T @ (predictions - Y) / len(Y)
    return gradient

def fit_model(self, X, W, Y, learn_rate=0.05, epochs=10):
    W_model = []    # save the weights for each class in this array
    classes = np.unique(Y)
    for c_class in classes:    # iterate through all unique classes in Y
        Y_c = np.where(Y == c_class, 1, 0)      #
        W_i= copy.deepcopy(W)        # Always use this as init weight
        for epoch in range(epochs):
            gradient = self.cost_function(X, W_i, Y_c)
            W_i = W_i - learn_rate*gradient
        W_model.append(W_i)
    return W_model

def predict_one_vs_all(self, X, W):
    max_probability = 0.0
    c_class = 99999
    for i in range(len(W)):
        curr_prob = self.hypothesis(X, W[i])
        if max_probability < curr_prob:
            max_probability = curr_prob
            c_class = i
    return c_class

def check_accuracy(self, X, W, Y):
    accuracy = 0.0
    tot_hits = 0
    for i in range(len(X)):
        if self.predict_one_vs_all(X[i], W) == Y[i]:
            tot_hits += 1
    accuracy = tot_hits/len(Y)
    # print("Accuracy: {0}%".format(accuracy*100.0))
    return accuracy

# Returns the weights calculated using Pseudo-Inverse
def linear_regression(self, X, Y):
    return (np.linalg.inv(X.T @ X) @ X.T) @ Y
```

TRAINING AND RESULTS

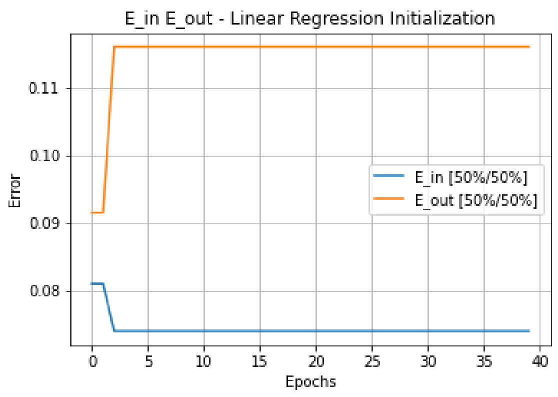
BREAST CANCER DATA

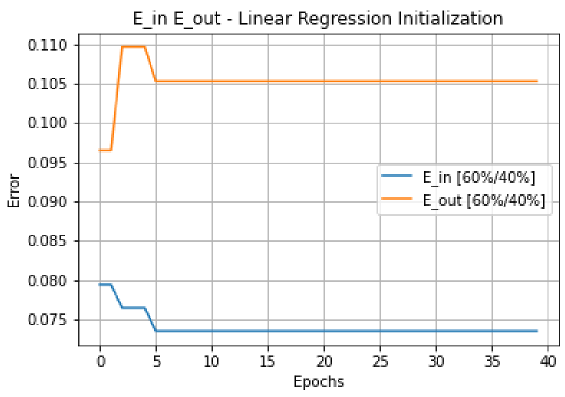
For breast cancer data as mentioned we used Pocket PLA and Linear Regression. As it was asked, I separated the dataset in 5 different ways:

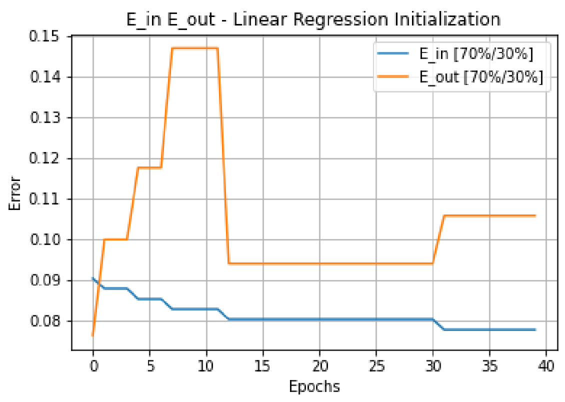
- 90 % training, 10% testing
- 80 % training, 20% testing
- 70 % training, 30% testing
- 60 % training, 40% testing
- 50 % training, 50% testing

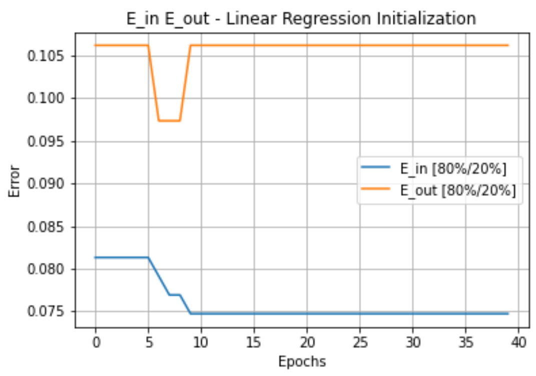
Before using the data for training I added the bias column. In the first iteration of experiments with these 5 data samples I used Linear Regression for initializing the weights. I have to mention that for initialization I used the whole batch of data. In-sample and out-sample errors ended up being very close for all the data samples above. To get a sense of the process I also ploted live-error changes of in-sample and out-sample (I keep doing the tests each iteration, just for the sake of understanding what's happening).

Experiment 1: Initialization using Linear Regression, Epochs=40, Algorithm: Pocket PLA









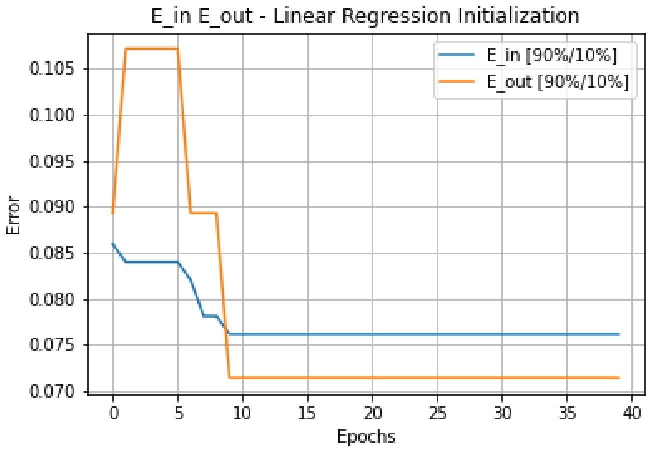
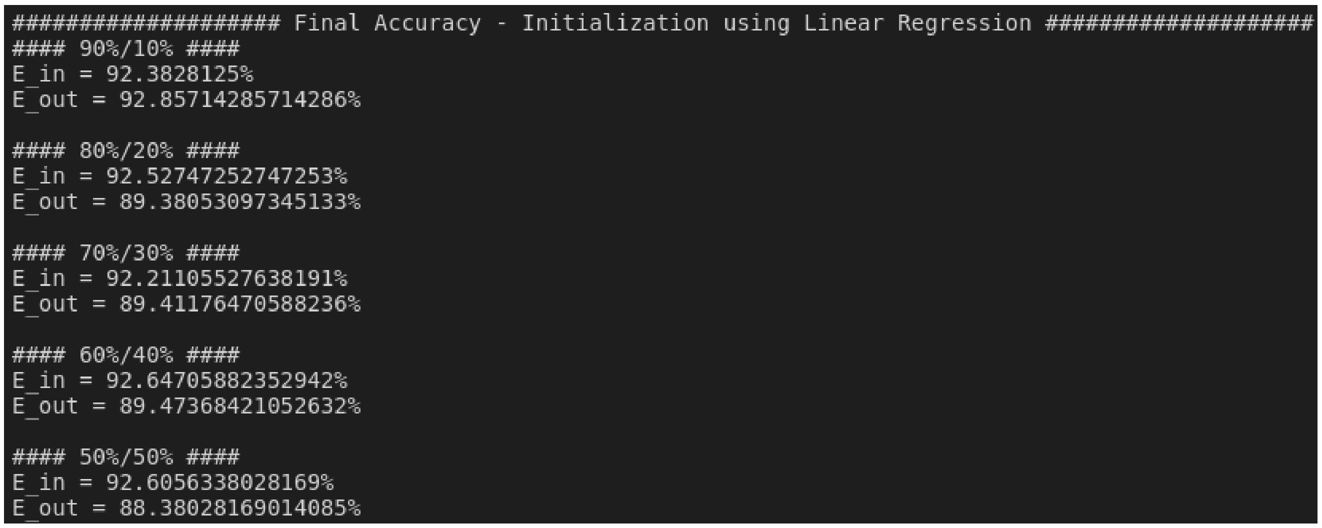


Fig. 3. *E_in and E_out plots for Breast Cancer Dataset – Initialization using LR.*

As we can see from figure 3 using LR for initialization it is very helpful, because the error as seen it starts very low already. Also, as expected the error most-likely decreases with the iterations. Finally, from the figure 4 we can see that all training results and testing results stay very close to each other in most of the data samples.



```
#################### Final Accuracy - Initialization using Linear Regression ####################
#### 90%/10% ####
E_in = 92.3828125%
E_out = 92.85714285714286%

#### 80%/20% ####
E_in = 92.52747252747253%
E_out = 89.38053097345133%

#### 70%/30% ####
E_in = 92.21105527638191%
E_out = 89.41176470588236%

#### 60%/40% ####
E_in = 92.64705882352942%
E_out = 89.47368421052632%

#### 50%/50% ####
E_in = 92.6056338028169%
E_out = 88.38028169014085%
```

Fig 4. *Accuracy in training and testing data – LR initialization.*

Experiment 2: Initialization using first data point, Epochs=40, Algorithm: Pocket PLA

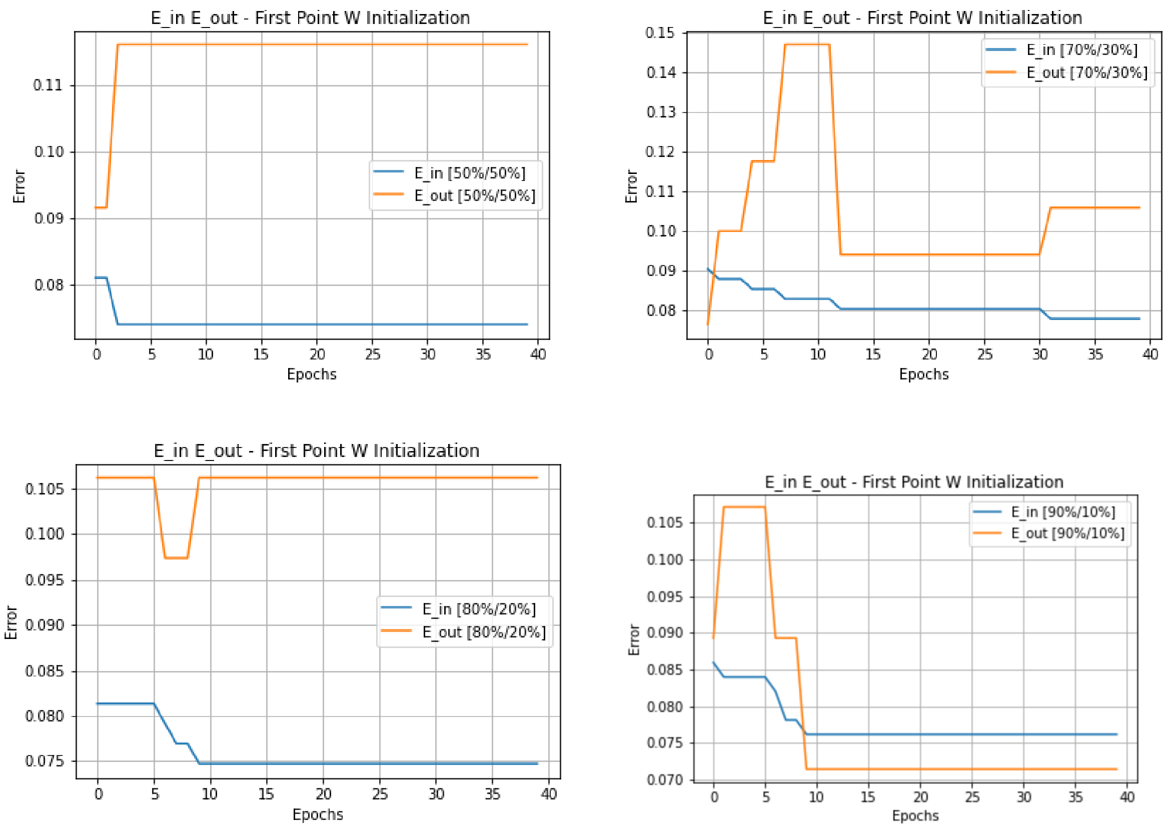
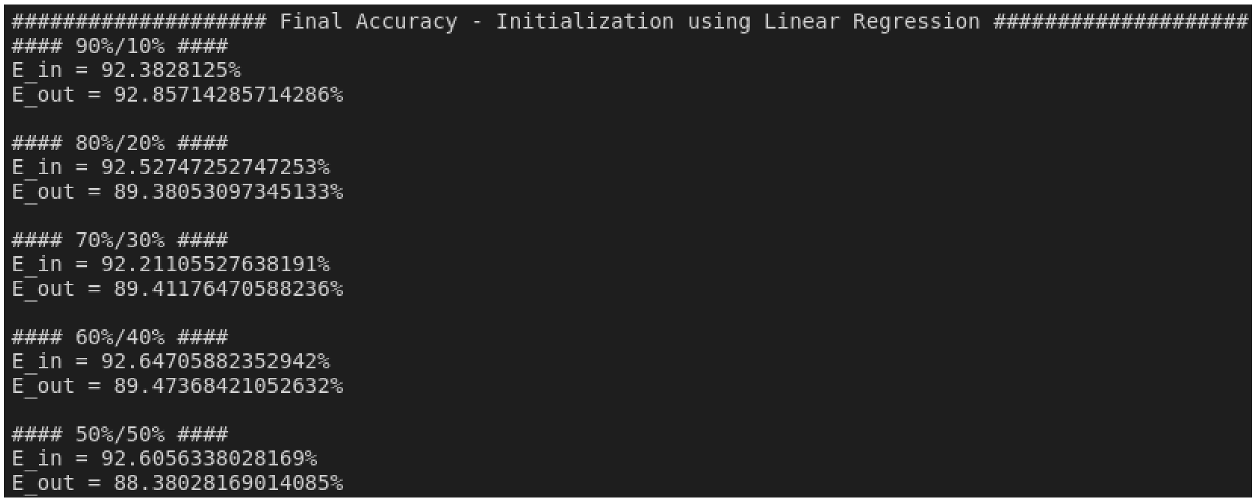


Fig 5. *E_in and E_out plots for Breast Cancer Dataset – Initialization using first point.*

As seen from the Fig. 5 the in-sample and out-sample errors stay very close to each other and we can say that algorithm performed well (be careful, before you judge the plots, check the scale of the y-axis). Finally, from figure 6 we see the accuracy from the best solution.



```
#################### Final Accuracy - Initialization using Linear Regression ####################
#### 90%/10% ####
E_in = 92.3828125%
E_out = 92.85714285714286%

#### 80%/20% ####
E_in = 92.52747252747253%
E_out = 89.38053097345133%

#### 70%/30% ####
E_in = 92.21105527638191%
E_out = 89.41176470588236%

#### 60%/40% ####
E_in = 92.64705882352942%
E_out = 89.47368421052632%

#### 50%/50% ####
E_in = 92.6056338028169%
E_out = 88.38028169014085%
```

Fig 6. *Accuracy in training and testing data – first data point initialization.*

I also kept track of **live-error** on testing and training data. I thought it is informative to understand the process, but probably not useful because it really slows down the learning process, because we do accuracy calculation for testing and training data each iteration. Results of this is shown in figure 7, and as we can see even for this experiments errors stay close to each other, which is an indicator that our classifier is a good representation of the model.

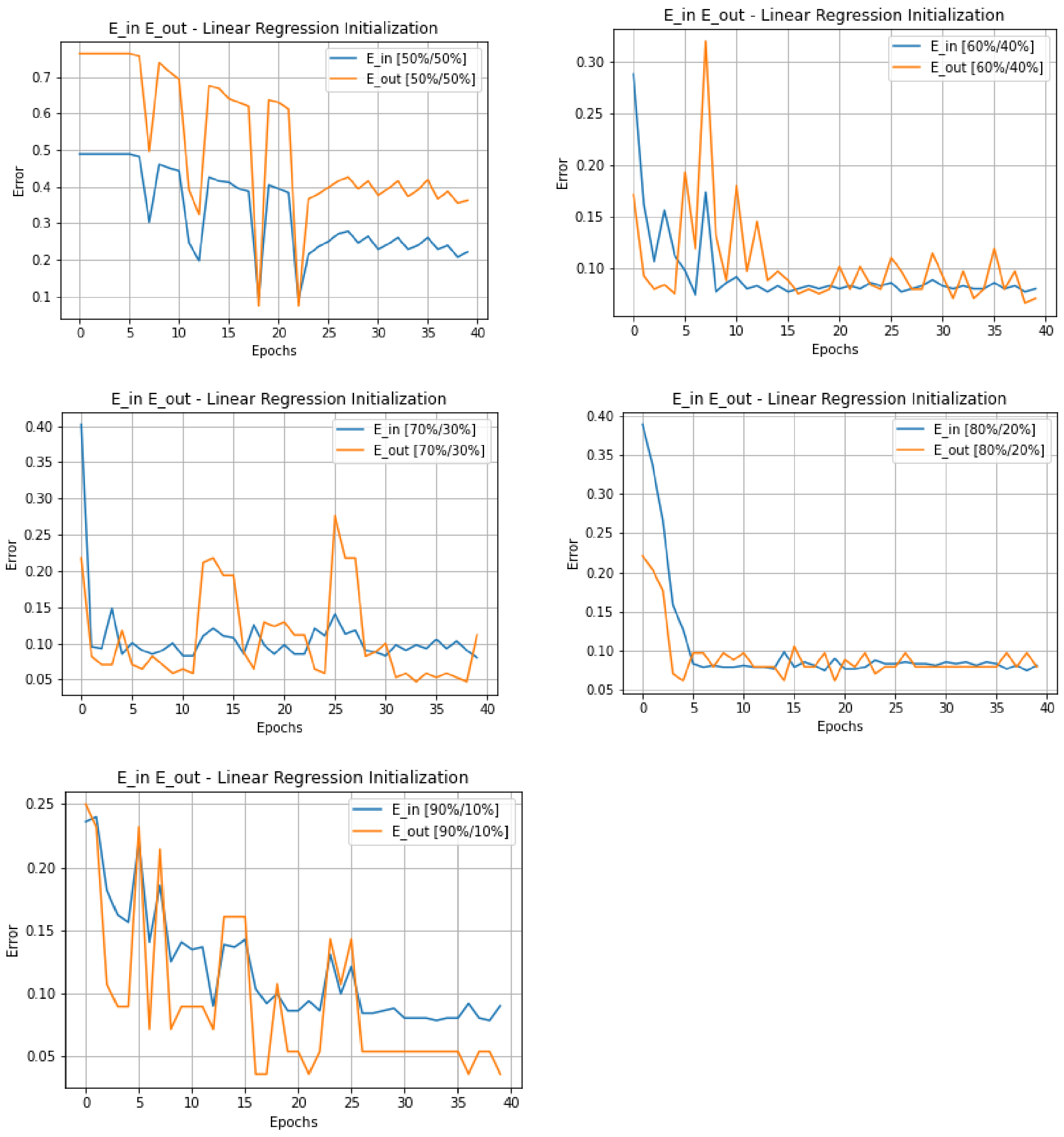


*Fig. 7. **LIVE** in-sample and out-sample error – Initialization using LR.*

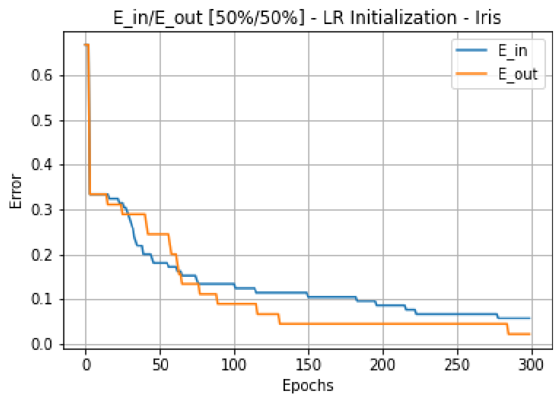
IRIS DATA

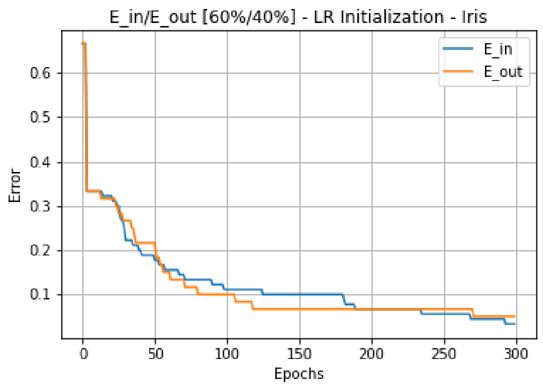
For iris data as mentioned I used Logistic regression and Linear Regression for first weight initialization. As it was asked, I separated the dataset in 5 different ways:

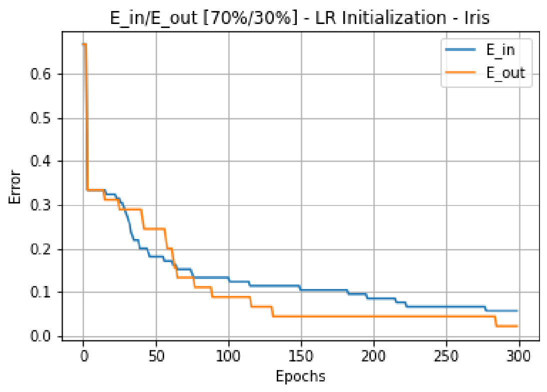
- 90 % training, 10% testing
- 80 % training, 20% testing
- 70 % training, 30% testing
- 60 % training, 40% testing
- 50 % training, 50% testing

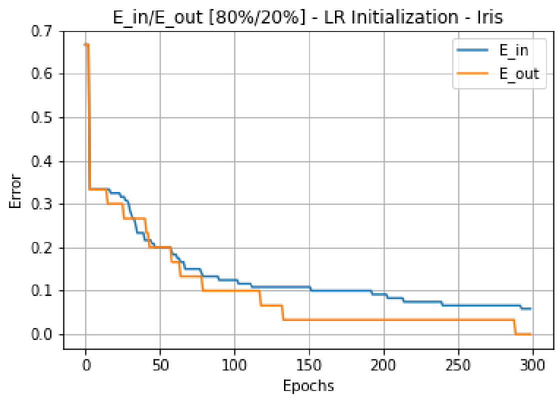
I have to mention that in this case I had to do a more complicated process because the data was ordered based on the class. For example 90% training means, 90% of the data of each class is used for training and the remaining 10% is used for testing. Please check the code for this.

Experiment 1: Initialization using Linear Regression; Epochs=300; Alg. Logistic Regression









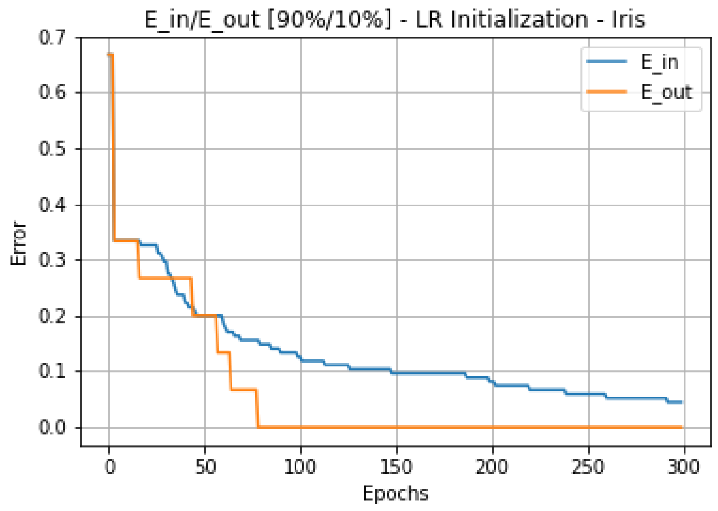
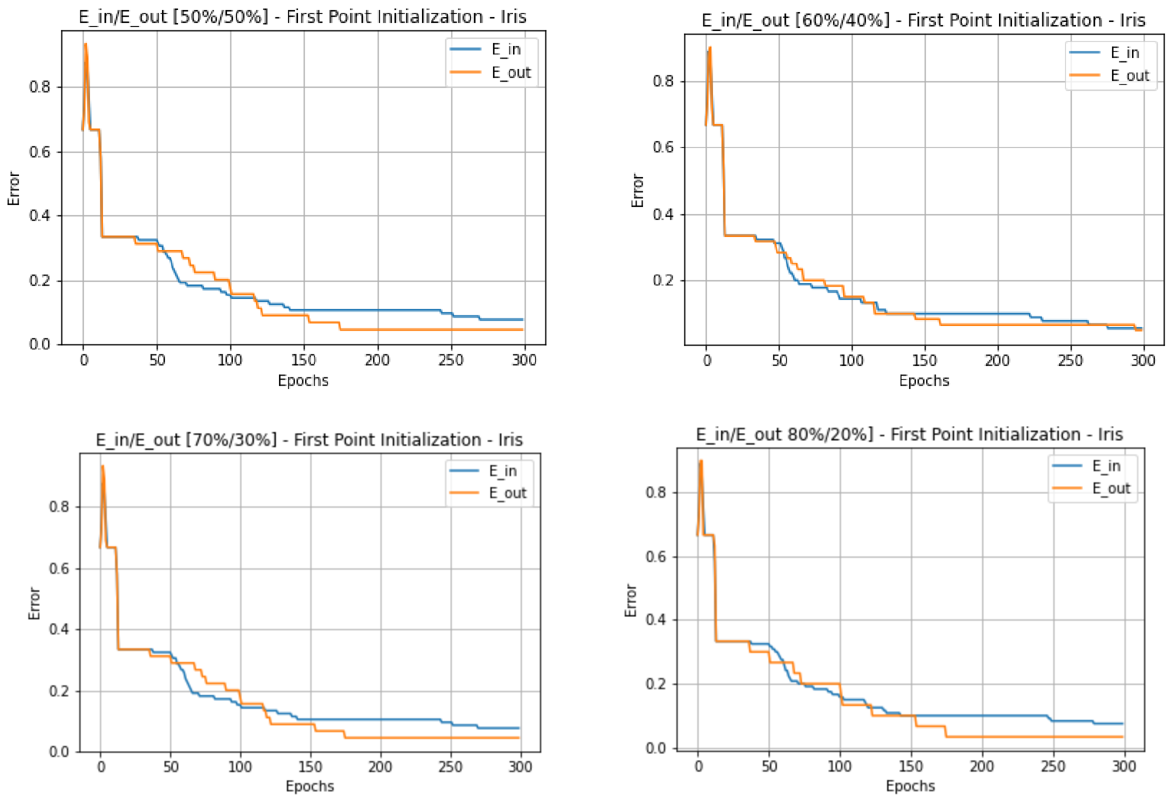


Fig. 8. *In-sample and out-sample error, Iris Dataset, LR initialization.*

Experiment 2: Initialization using First Point; Epochs=300; Alg. Logistic Regression



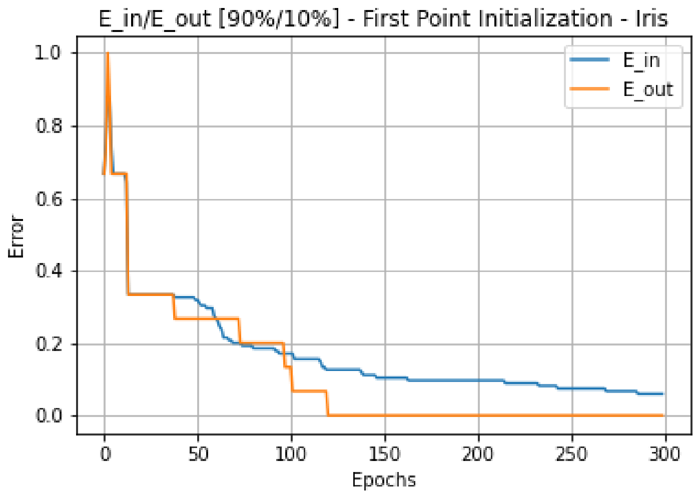
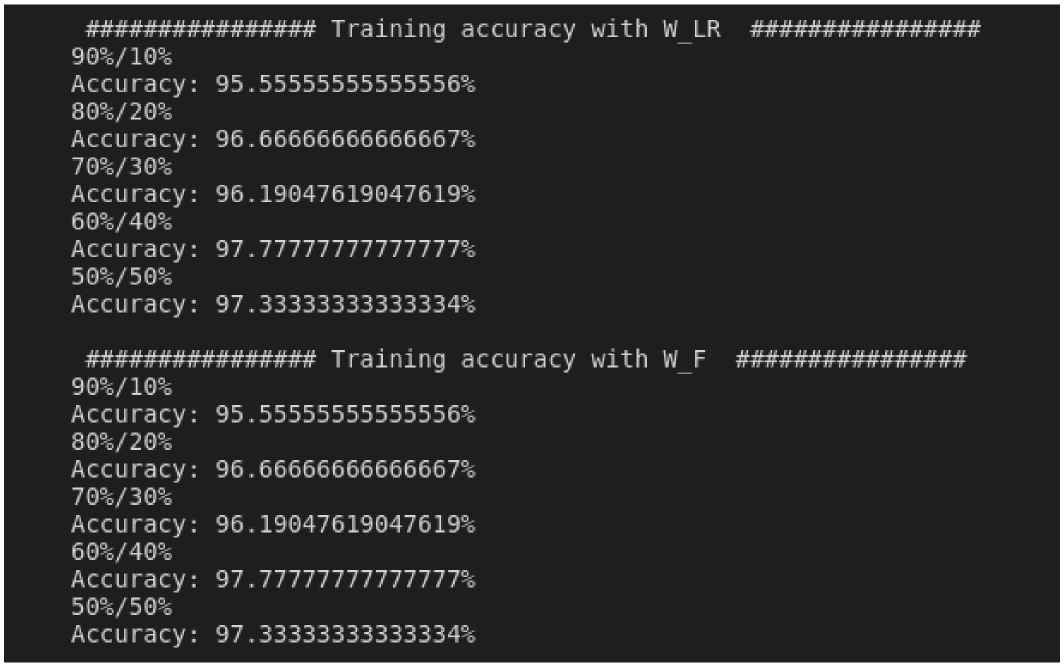


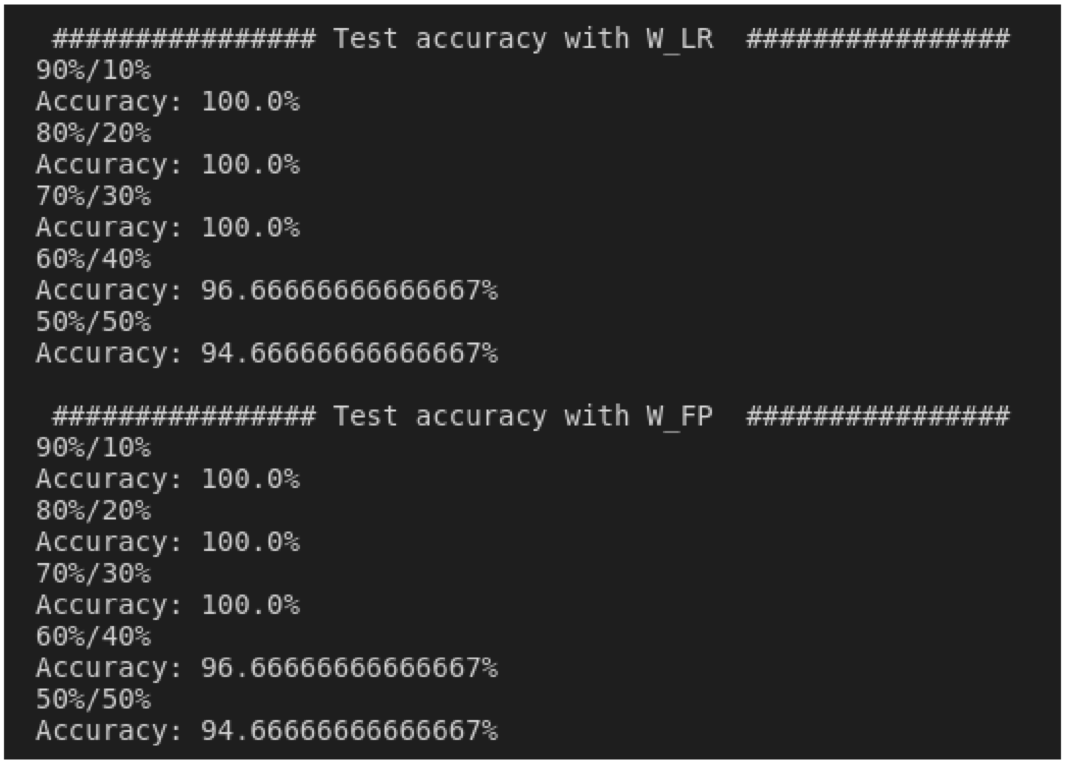
Fig 10. . *In-sample and out-sample errors, Iris Dataset, first point Initialization.*

As seen from results in figure 8 in-sample and out-sample are very close to each other, and as expected both of them decrease with the number of iterations performed. In experiment 2, we use first point initialization, and as we can see in the beginning the error is quite high, but it decreases fast. We can say that algorithm performs very well, and that can be proven by these graphs and also the final accuracy shown in figure 11.



```
 ################ Training accuracy with W_LR  ################
90%/10%
Accuracy: 95.55555555555556%
80%/20%
Accuracy: 96.66666666666667%
70%/30%
Accuracy: 96.19047619047619%
60%/40%
Accuracy: 97.77777777777777%
50%/50%
Accuracy: 97.33333333333334%

 ################ Training accuracy with W_F  ################
90%/10%
Accuracy: 95.55555555555556%
80%/20%
Accuracy: 96.66666666666667%
70%/30%
Accuracy: 96.19047619047619%
60%/40%
Accuracy: 97.77777777777777%
50%/50%
Accuracy: 97.33333333333334%
```



```
 ################ Test accuracy with W_LR  ################
90%/10%
Accuracy: 100.0%
80%/20%
Accuracy: 100.0%
70%/30%
Accuracy: 100.0%
60%/40%
Accuracy: 96.66666666666667%
50%/50%
Accuracy: 94.66666666666667%

 ################ Test accuracy with W_FP  ################
90%/10%
Accuracy: 100.0%
80%/20%
Accuracy: 100.0%
70%/30%
Accuracy: 100.0%
60%/40%
Accuracy: 96.66666666666667%
50%/50%
Accuracy: 94.66666666666667%
```

Fig. 9. *Accuracy on training and testing data – Iris dataset, epochs=3000.*

CONCLUSION

In this work we tested three different important algorithms and their performance in sklearn datasets IRIS and BREAST CANCER. Accuracy in testing and training data is very close to each other, which is a good indicator that we have a good hypothesis of the model. Linear regression is very important step, it helps the learning process to achieve the steady state faster. A surprising thing is when you get better accuracy in testing data than training data, and that means that training data contains either more details of the model or noise. It is worth to mention that Pocket PLA is slow. Another important thing is how I chose the data separation in the second problem, I did a manual data-shuffling in order to keep the good representation of the model in both training and testing samples. Finally, when datasets are large we can see that it is most likely that in-sample and out-sample errors are very close to each other, this proves in practice Hoeffding's Inequality.