# STATISTICAL METHODS FOR MACHINE LEARNING COURSE PROJECT

Date: 19/06/2024

Instructor: Nicolo Cesa-Bianchi

Student: Hasan Kuşpınar

Project Type: Kernelized Linear Regression

TABLE OF CONTENTS PAGE

## A. INTRODUCTION

In recent years with the enhancements in the field of Machine Learning, complex models have become available for usage in most areas. However, the main goal of this project is rather than using the already available models implementing some machine learning algorithms from scratch. Firstly, a given dataset is explored and preprocessed so that it can be used by the algorithms. Secondly, the algorithms are implemented from scratch as python classes. Moreover, the algorithms are trained using the training set and cross validation is performed to tune hyperparameters. After the cross validation the training error and test errors are compared. Zero one loss is used to check the training and test errors. Based on this comparison the existence of overfitting or underfitting is checked.

The algorithms implemented in the project are Perceptron, Pegasos SVM and Logistic Pegasos. After these models are evaluated, polynomial feature expansion is done on the dataset to check how it affects the results. Lastly, Kernel trick is performed on the Perceptron and Pegasos algorithm to see how the results are changed.

## B. THEORY

### 1. Perceptron

The Perceptron algorithm is a solution to the ERM (empirical risk minimization) problem for the linearly separable case. The main goal of Perceptron is to find a homogeneous separating hyperplane via checking every example in the training set subsequently. Each example is tested with the classifier to see if it misclassifies and if so, it updates the hyperplane. Finally, if the algorithm terminates that hyperplane, is the separating hyperplane (Cesa-Bianchi 2024).

### 2. Pegasos SVM

The SVM is an algorithm to find the classifier with the maximum margin separating hyperplane, given a linearly separable data. What Pegasos does is that it performs stochastic gradient descent on an objective function to minimize it. Unlike the Perceptron it uses the hinge loss. It is run through examples that are randomly drawn from the training dataset. Then their weights are updated based on the gradient (Cesa-Bianchi 2024).

### 3. Polynomial Feature Expansion

The polynomial feature expansion helps decrease the error for linear predictors since linear predictors usually have large errors due to their number of coefficients being smaller than number of features. A possible way we can fix is the polynomial feature expansion which increases the number of features with nonlinear combinations of the original features. This helps in the case of non-linearly separable data. After the data with the expanded features is used to train the linear predictor a more complex nonlinear predictor can be achieved. This helps reduce the error and increase the stability of the model (Cesa-Bianchi 2024).

### 4. Kernel Methods

Another possible solution to overcome the problem of large errors of linear classifiers is the kernel trick. The kernel trick helps SVM's use linear classifiers with non-linearly separable data. Since better separation could be done on these data, the errors of these classifiers will be reduced (Cesa-Bianchi 2024).

## C. RESULTS

### 1. Perceptron Analysis

The Perceptron algorithm is implemented using the pseudo-code from the lectures (see Appendix A). First of all, k-fold cross validation is done on the training set to find the best combination of eta and epochs. After the cross validation, the model is trained again with the found parameter values. Then the trained model is tested on the test set. Looking at the results from the cross validation (training) phase and the test phase, it is observed that both errors are not very low (see Appendix E). Therefore, it implies underfitting since both training and test errors are high. As a result, it can be deduced that the dataset used is not linearly separable since perceptron tries to find the separating hyperplane that minimizes the risk.

## 2. Pegasos Analysis

The Pegasos algorithm is also based on the pseudo-code from the lecture (see Appendix B). Cross validation is performed on the training set to determine the ideal values for lambda and epoch. After the best values for the parameters have been found, they are used to train the model again. And the model is tested with the test set. Subsequently, checking the results from these steps makes it clear that both the training and test errors are high which implies underfitting (see Appendix F). Therefore, similarly with the perceptron case it can be seen that the dataset is not linearly separable. This is as a result of Pegasos being an algorithm to find the maximum margin separating hyperplane for a linearly separable dataset.

## 3. Logistic Pegasos Analysis

Another algorithm that is implemented in this project is the logistic pegasos. In this algorithm instead of hinge loss, logistic loss is used. Firstly, like the previous algorithms cross validation is performed on the training set and best set of parameters have been found. Then the model is trained on these parameters and tested with the test set. The first thing that can be observed after looking at the results is that it has pretty big training and test error compared to the original Pegasos (see Appendix G). Similarly, with the previous algorithms a possible implication of this high error is the fact that the dataset is not linearly separable. Another, possible reason for the high error might be the usage of logistic loss instead of hinge loss. Since the logistic loss is more sensitive to outliers that are the points far from the boundary, it possibly has higher error than the original one with hinge loss.

## 4. Polynomial Feature Expansion Analysis

After the evaluation of the previous model, polynomial feature expansion of degree 2 is performed on the dataset. Then, Perceptron and Pegasos are tested with this new expanded set to see if it has improved the results. Firstly, the results of the Perceptron on the expanded set have a significantly lower error than the normal one (see Appendix H). Likewise, Pegasos' results after it is fitted on the expanded set are lower than the normal (see Appendix H). Both of these results are expected since the polynomial feature expansion results in a more complex non-linear predictor.

### 5. Kernelized Perceptron Analysis

Another possible way to improve the performance of the algorithms is the Kernel trick. First of all, polynomial and gaussian kernels are defined before the class definitions. Then Kernel version of Perceptron is implemented based on the lecture notes as well (see Appendix D). Kernel Perceptron is firstly trained with the gaussian kernel with 0.1 gamma and evaluated with both the training and the test sets. The results observed have a lot less error compared to the original Perceptron. Then it is trained with the polynomial perceptron with degree 2 and evaluated with also the training and the test sets. Again these results show a significantly lower error than the original version (see Appendix E). The results from the gaussian and the polynomial kernels are expected since the aim of using kernels is to be able to separate non-linearly separable data by having higher degree separating curves.

### 6. Kernelized Pegasos Analysis

Final implementation is the Kernelized Pegasos which is based on the lecture as well (see Appendix C). Firstly, the algorithm is trained on the training set with the gaussian kernel with gamma 0.01. Then it is tested with both the test and training set and lower errors are observed compared to the original pegasos (see Appendix K). Furthermore, it is trained on the trained with the polynomial kernel with degree 4 and evaluated with the test and training sets. The results from it have a similar error value with the original Pegasos (see Appendix L). The gaussian kernel helped separating the data more than the polynomial kernel which means for the Pegasos more complex separating curves for the data can be achieved with the gaussian kernel.

## D. CONCLUSION

In conclusion, it can be induced that machine learning algorithms can be implemented efficiently without using libraries. However, there are some crucial things to be taken into account such as; using a proper and preprocessed dataset, tuning right hyperparameters, using the correct kernel methods and also using feature expansion is an option. After these steps are carefully processed, the implementations of the algorithms can be successfully done.

## E. REFERENCES

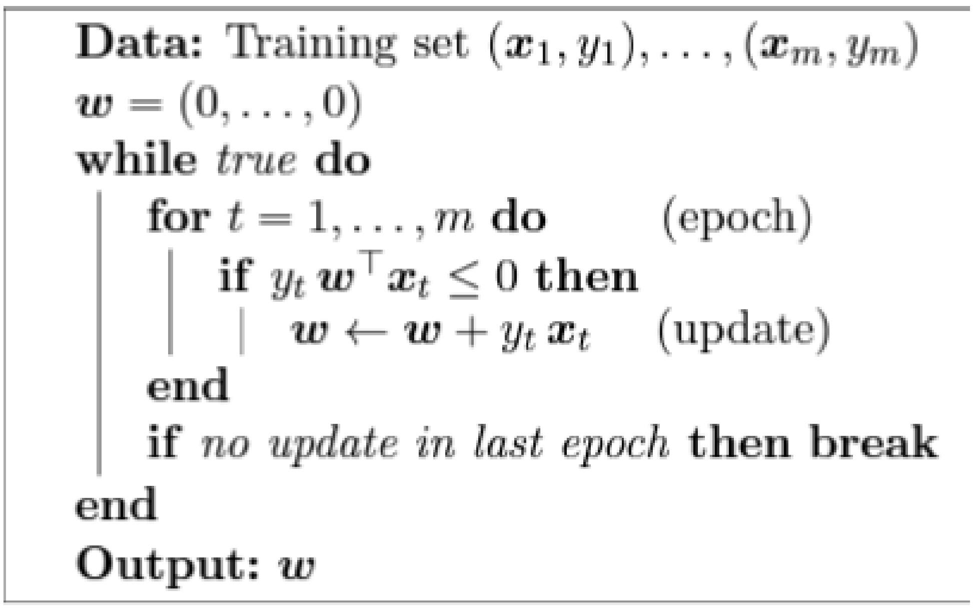
Cesa-Bianchi, Nicolo. (2024, May). *Linear Predictors* [PDF]. University of Milan.

Cesa-Bianchi, Nicolo. (2024, May). *Kernel Functions* [PDF]. University of Milan.

Cesa-Bianchi, Nicolo. (2024, May). *Support Vector Machines* [PDF]. University of Milan.

F. APPENDICES

Appendix A



```
Data: Training set $(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_m, y_m)$
$\boldsymbol{w} = (0, \ldots, 0)$
while *true* do
    for $t = 1, \ldots, m$ do        (epoch)
        if $y_t\, \boldsymbol{w}^\top \boldsymbol{x}_t \leq 0$ then
            $\boldsymbol{w} \leftarrow \boldsymbol{w} + y_t\, \boldsymbol{x}_t$     (update)
    end
    if *no update in last epoch* then break
end
Output: $\boldsymbol{w}$
```

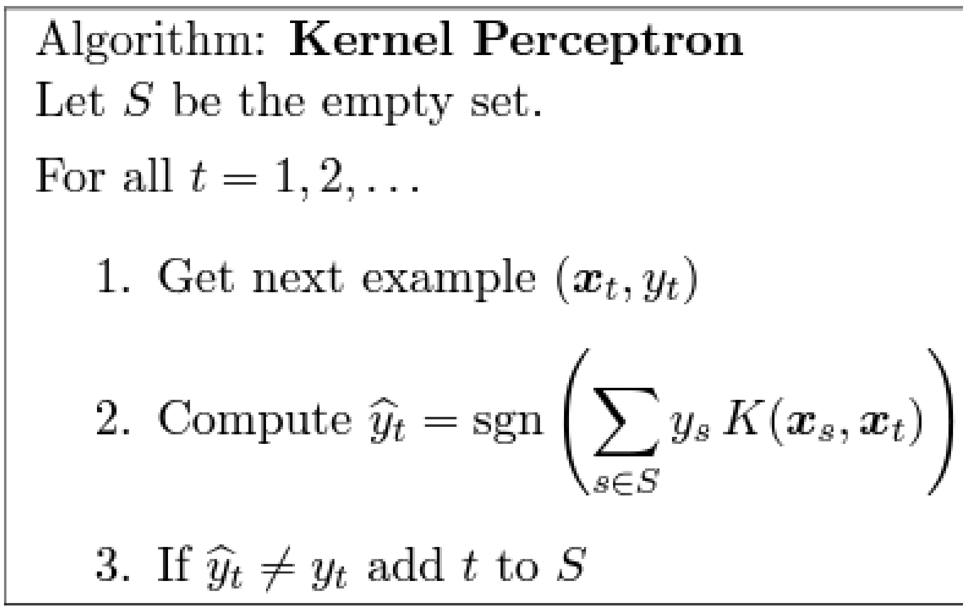
## Appendix B

```
**Parameters:** number $T$ of rounds, regularization coefficient $\lambda > 0$
**Input:** Training set $(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_m, y_m) \in \mathbb{R}^d \times \{-1, 1\}$
Set $\boldsymbol{w}_1 = \mathbf{0}$
For $t = 1, \ldots, T$
    1. Draw uniformly at random an element $(\boldsymbol{x}_{Z_t}, y_{Z_t})$ from the training set
    2. Set $\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \eta_t \nabla \ell_{Z_t}(\boldsymbol{w}_t)$
Output: $\overline{\boldsymbol{w}} = \frac{1}{T}(\boldsymbol{w}_1 + \cdots + \boldsymbol{w}_T)$.
```

## Appendix C

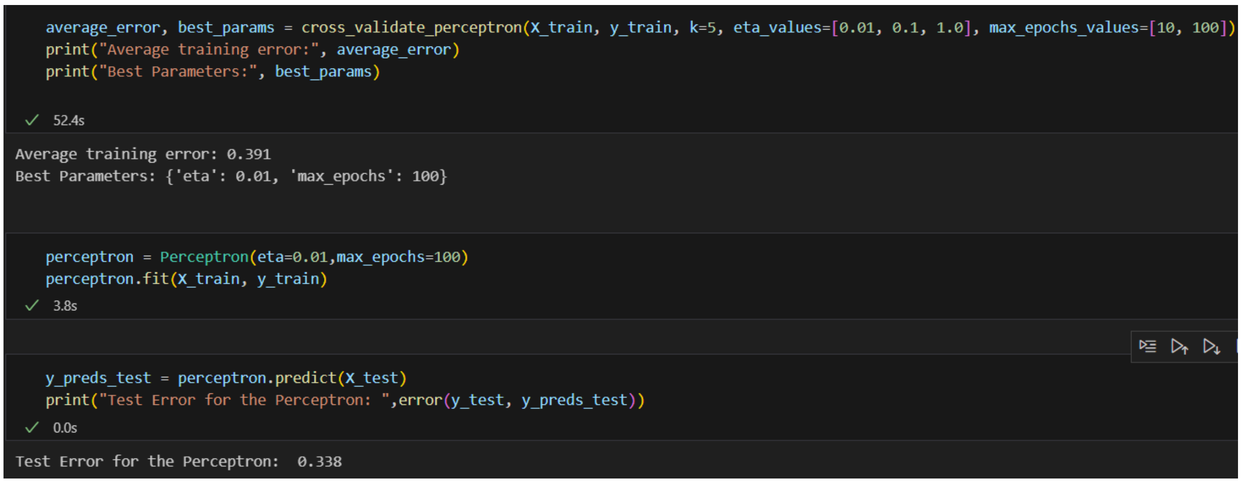
```
INPUT: $S, \lambda, T$
INITIALIZE: Set $\boldsymbol{\alpha}_1 = 0$
FOR $t = 1, 2, \ldots, T$
    Choose $i_t \in \{0, \ldots, |S|\}$ uniformly at random.
    For all $j \neq i_t$, set $\alpha_{t+1}[j] = \alpha_t[j]$
    If $y_{i_t} \frac{1}{\lambda t} \sum_j \alpha_t[j] y_{i_t} K(\mathbf{x}_{i_t}, \mathbf{x}_j) < 1$, then:
        Set $\alpha_{t+1}[i_t] = \alpha_t[i_t] + 1$
    Else:
        Set $\alpha_{t+1}[i_t] = \alpha_t[i_t]$
OUTPUT: $\boldsymbol{\alpha}_{T+1}$
```

Appendix D



```
Algorithm: **Kernel Perceptron**
Let $S$ be the empty set.
For all $t = 1, 2, \ldots$
    1. Get next example $(\boldsymbol{x}_t, y_t)$
    2. Compute $\widehat{y}_t = \operatorname{sgn}\left(\sum_{s \in S} y_s \, K(\boldsymbol{x}_s, \boldsymbol{x}_t)\right)$
    3. If $\widehat{y}_t \neq y_t$ add $t$ to $S$
```

## Appendix E



```
average_error, best_params = cross_validate_perceptron(X_train, y_train, k=5, eta_values=[0.01, 0.1, 1.0], max_epochs_values=[10, 100])
print("Average training error:", average_error)
print("Best Parameters:", best_params)
```

52.4s

```
Average training error: 0.391
Best Parameters: {'eta': 0.01, 'max_epochs': 100}
```

```
perceptron = Perceptron(eta=0.01,max_epochs=100)
perceptron.fit(X_train, y_train)
```

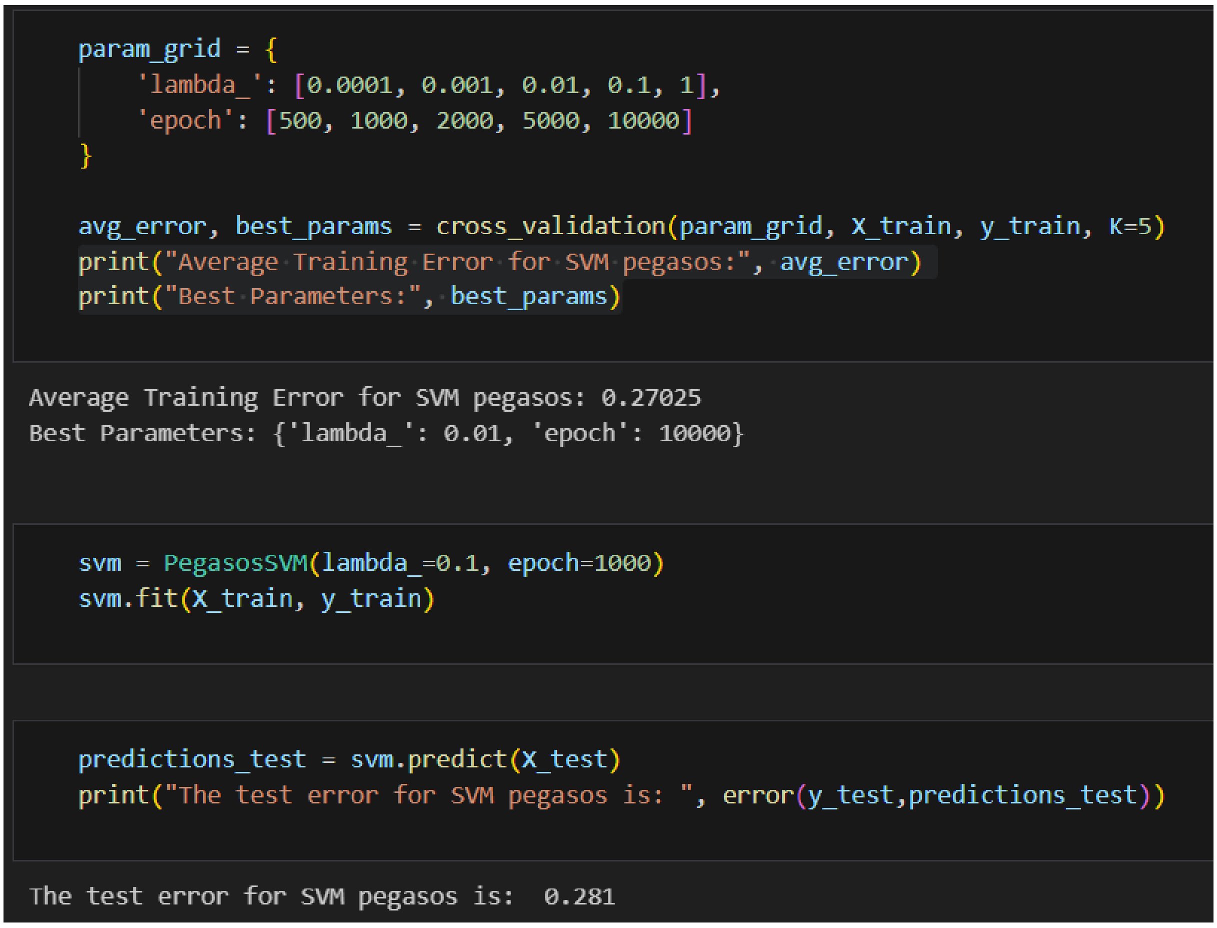
3.8s

```
y_preds_test = perceptron.predict(X_test)
print("Test Error for the Perceptron: ",error(y_test, y_preds_test))
```

0.0s

```
Test Error for the Perceptron:  0.338
```

## Appendix F



```
param_grid = {
    'lambda_': [0.0001, 0.001, 0.01, 0.1, 1],
    'epoch': [500, 1000, 2000, 5000, 10000]
}

avg_error, best_params = cross_validation(param_grid, X_train, y_train, K=5)
print("Average Training Error for SVM pegasos:", avg_error)
print("Best Parameters:", best_params)
```

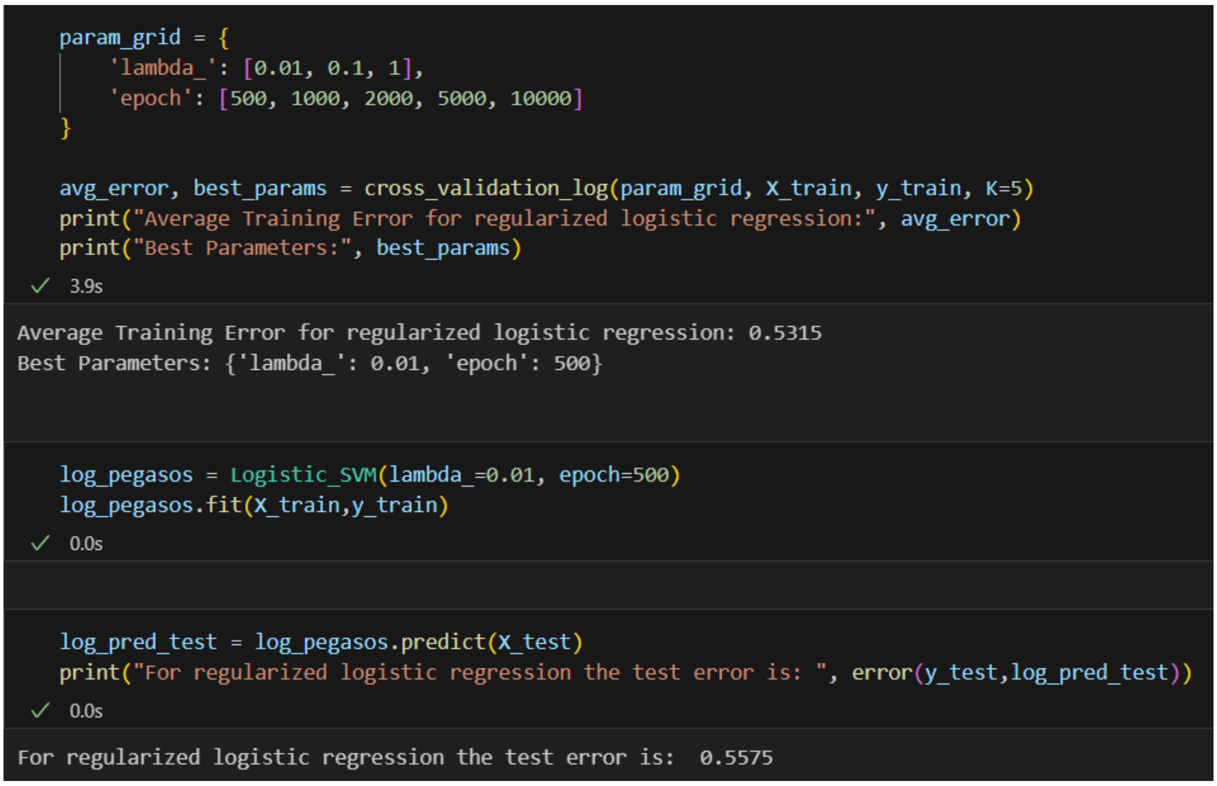
```
Average Training Error for SVM pegasos: 0.27025
Best Parameters: {'lambda_': 0.01, 'epoch': 10000}
```

```
svm = PegasosSVM(lambda_=0.1, epoch=1000)
svm.fit(X_train, y_train)
```

```
predictions_test = svm.predict(X_test)
print("The test error for SVM pegasos is: ", error(y_test,predictions_test))
```

```
The test error for SVM pegasos is:  0.281
```

# Appendix G



```
param_grid = {
    'lambda_': [0.01, 0.1, 1],
    'epoch': [500, 1000, 2000, 5000, 10000]
}

avg_error, best_params = cross_validation_log(param_grid, X_train, y_train, K=5)
print("Average Training Error for regularized logistic regression:", avg_error)
print("Best Parameters:", best_params)
```
✓ 3.9s

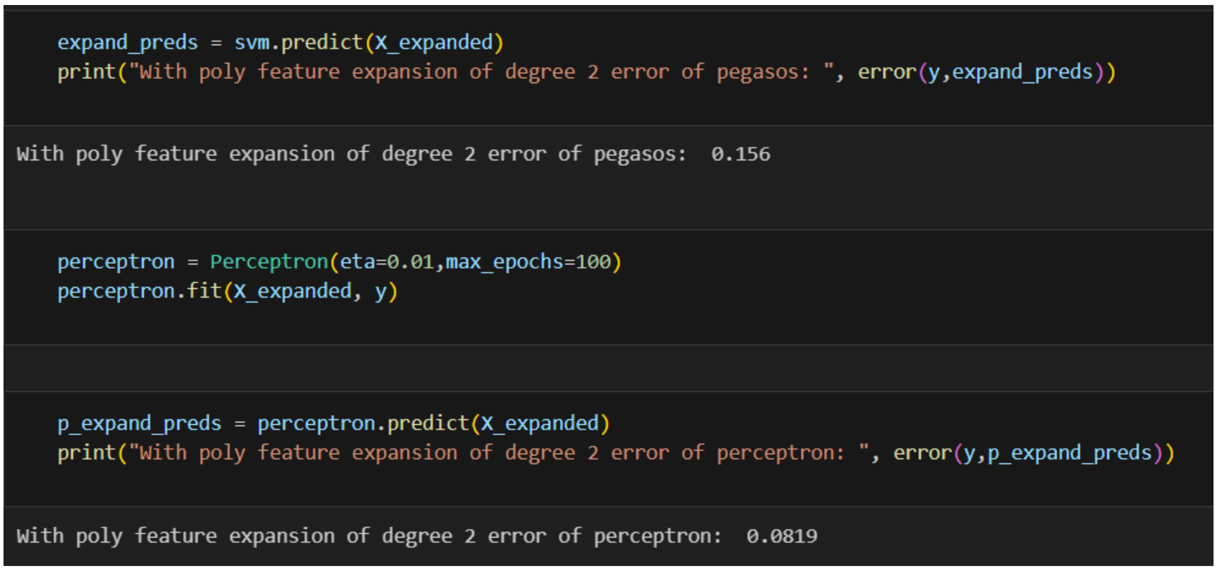
```
Average Training Error for regularized logistic regression: 0.5315
Best Parameters: {'lambda_': 0.01, 'epoch': 500}
```

```
log_pegasos = Logistic_SVM(lambda_=0.01, epoch=500)
log_pegasos.fit(X_train,y_train)
```
✓ 0.0s

```
log_pred_test = log_pegasos.predict(X_test)
print("For regularized logistic regression the test error is: ", error(y_test,log_pred_test))
```
✓ 0.0s

```
For regularized logistic regression the test error is:  0.5575
```

## Appendix H



```
expand_preds = svm.predict(X_expanded)
print("With poly feature expansion of degree 2 error of pegasos: ", error(y,expand_preds))
```

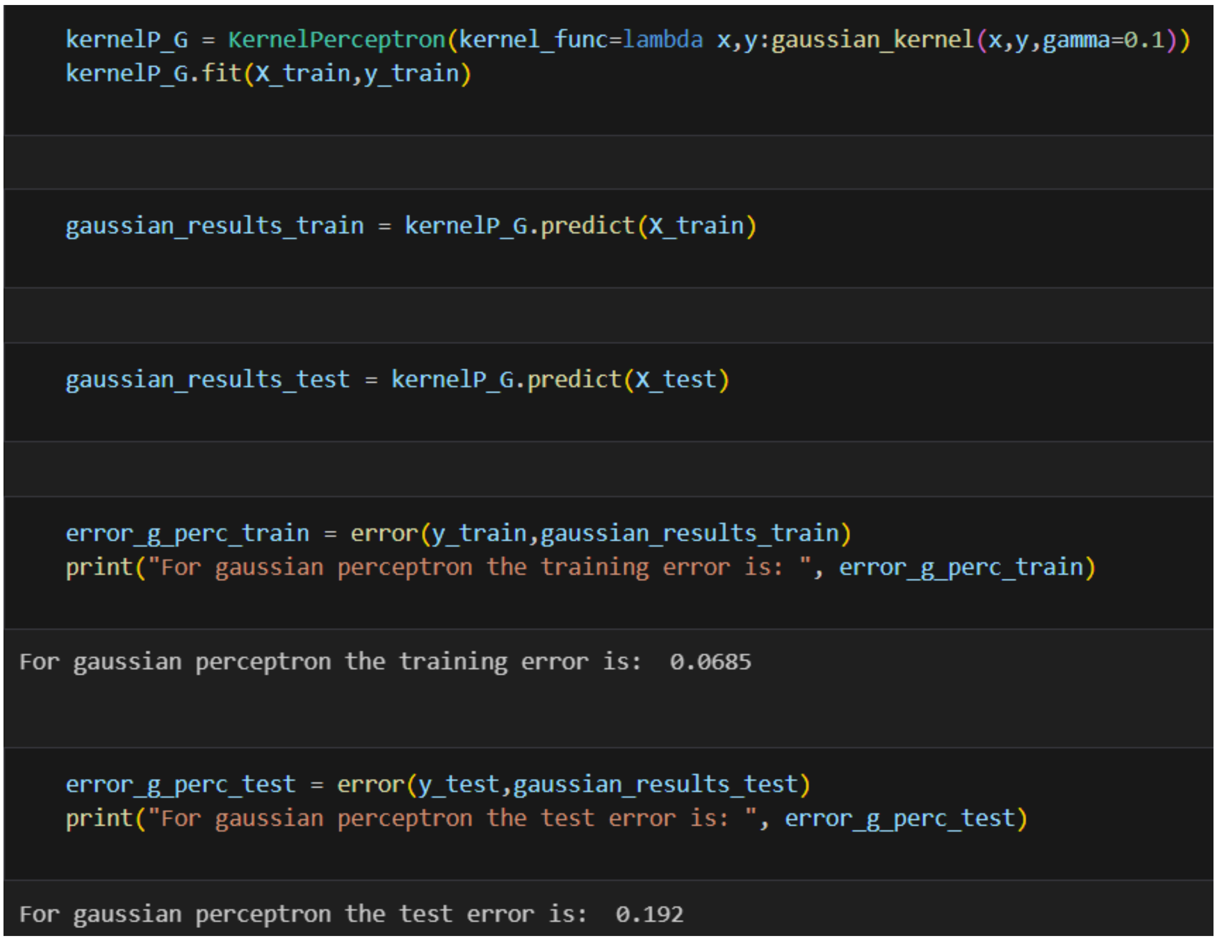
```
With poly feature expansion of degree 2 error of pegasos:  0.156
```

```
perceptron = Perceptron(eta=0.01,max_epochs=100)
perceptron.fit(X_expanded, y)
```

```
p_expand_preds = perceptron.predict(X_expanded)
print("With poly feature expansion of degree 2 error of perceptron: ", error(y,p_expand_preds))
```

```
With poly feature expansion of degree 2 error of perceptron:  0.0819
```

Appendix I



```
kernelP_G = KernelPerceptron(kernel_func=lambda x,y:gaussian_kernel(x,y,gamma=0.1))
kernelP_G.fit(X_train,y_train)
```

```
gaussian_results_train = kernelP_G.predict(X_train)
```

```
gaussian_results_test = kernelP_G.predict(X_test)
```

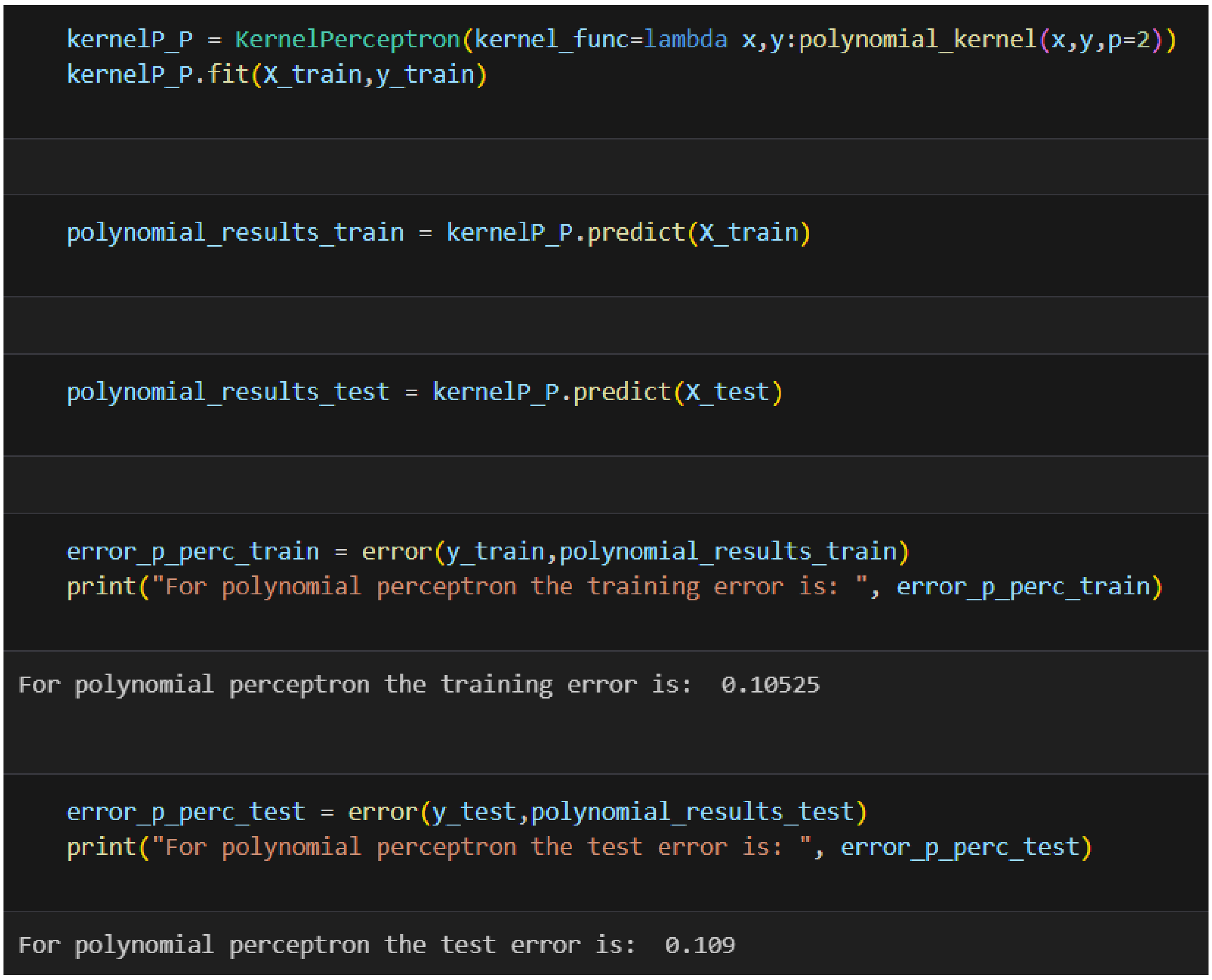
```
error_g_perc_train = error(y_train,gaussian_results_train)
print("For gaussian perceptron the training error is: ", error_g_perc_train)
```

```
For gaussian perceptron the training error is:  0.0685
```

```
error_g_perc_test = error(y_test,gaussian_results_test)
print("For gaussian perceptron the test error is: ", error_g_perc_test)
```

```
For gaussian perceptron the test error is:  0.192
```

## Appendix J



```
kernelP_P = KernelPerceptron(kernel_func=lambda x,y:polynomial_kernel(x,y,p=2))
kernelP_P.fit(X_train,y_train)
```

```
polynomial_results_train = kernelP_P.predict(X_train)
```

```
polynomial_results_test = kernelP_P.predict(X_test)
```

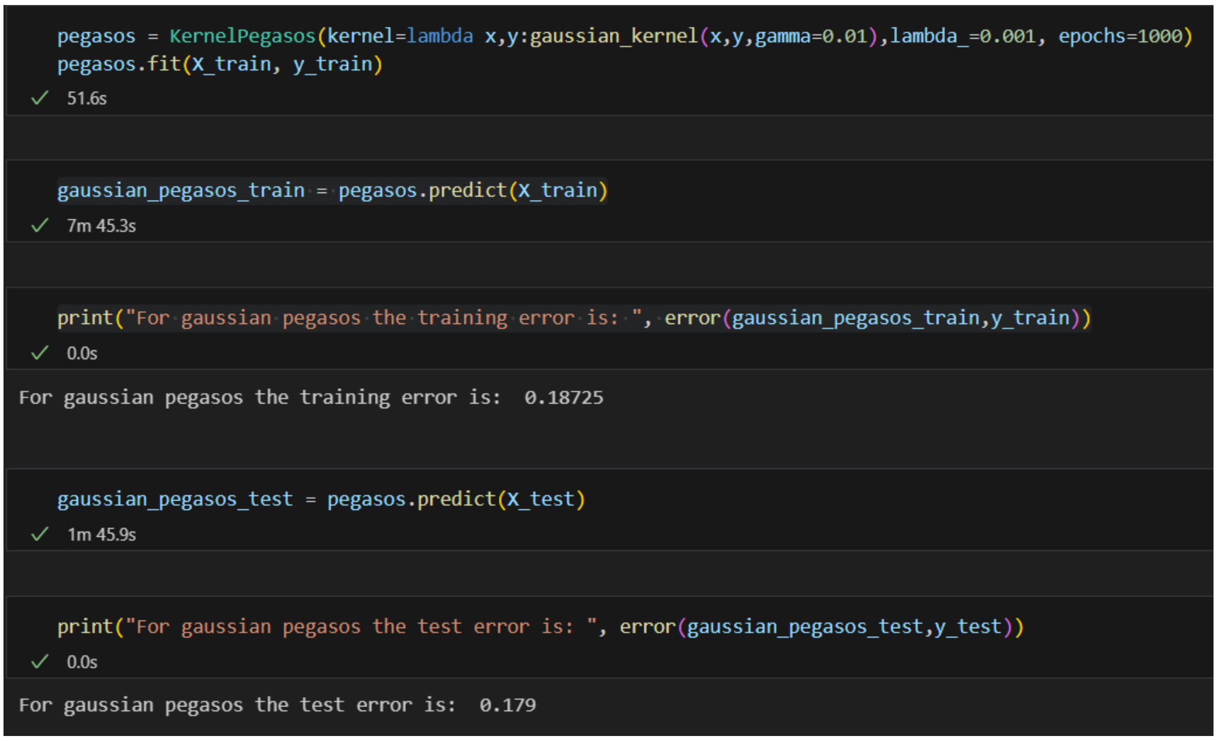
```
error_p_perc_train = error(y_train,polynomial_results_train)
print("For polynomial perceptron the training error is: ", error_p_perc_train)
```

```
For polynomial perceptron the training error is:  0.10525
```

```
error_p_perc_test = error(y_test,polynomial_results_test)
print("For polynomial perceptron the test error is: ", error_p_perc_test)
```

```
For polynomial perceptron the test error is:  0.109
```

## Appendix K



```
pegasos = KernelPegasos(kernel=lambda x,y:gaussian_kernel(x,y,gamma=0.01),lambda_=0.001, epochs=1000)
pegasos.fit(X_train, y_train)
```
✓ 51.6s

```
gaussian_pegasos_train = pegasos.predict(X_train)
```
✓ 7m 45.3s

```
print("For gaussian pegasos the training error is: ", error(gaussian_pegasos_train,y_train))
```
✓ 0.0s

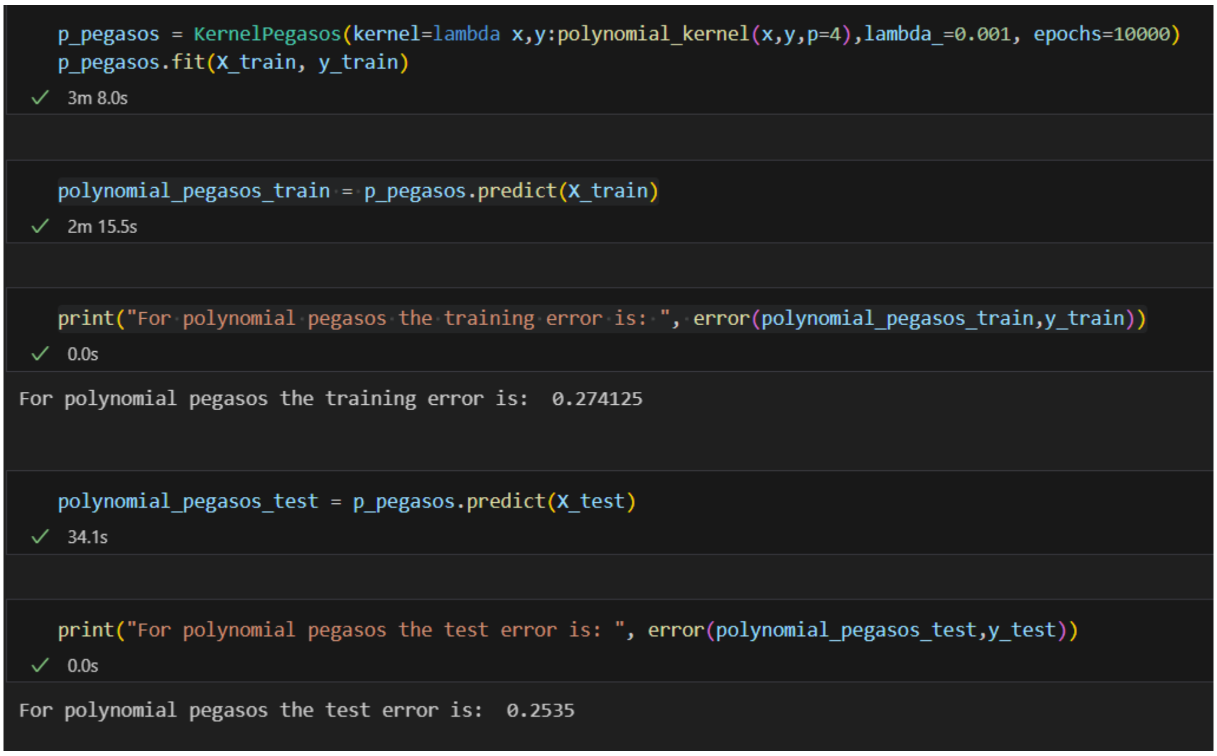
```
For gaussian pegasos the training error is:  0.18725
```

```
gaussian_pegasos_test = pegasos.predict(X_test)
```
✓ 1m 45.9s

```
print("For gaussian pegasos the test error is: ", error(gaussian_pegasos_test,y_test))
```
✓ 0.0s

```
For gaussian pegasos the test error is:  0.179
```

# Appendix L



```
p_pegasos = KernelPegasos(kernel=lambda x,y:polynomial_kernel(x,y,p=4),lambda_=0.001, epochs=10000)
p_pegasos.fit(X_train, y_train)
```
✓ 3m 8.0s

```
polynomial_pegasos_train = p_pegasos.predict(X_train)
```
✓ 2m 15.5s

```
print("For polynomial pegasos the training error is: ", error(polynomial_pegasos_train,y_train))
```
✓ 0.0s

```
For polynomial pegasos the training error is:  0.274125
```

```
polynomial_pegasos_test = p_pegasos.predict(X_test)
```
✓ 34.1s

```
print("For polynomial pegasos the test error is: ", error(polynomial_pegasos_test,y_test))
```
✓ 0.0s

```
For polynomial pegasos the test error is:  0.2535
```