Project 1:
Kernelized Linear Classification

Stefano Chiesa

Abstract

This report presents an analysis of kernelized linear classification. I implemented several machine learning algorithms from scratch, including the Perceptron, Support Vector Machines (SVMs) using the Pegasos algorithm, and Regularized Logistic Regression. I analysed the impact of polynomial feature expansion and kernel methods using Gaussian and polynomial kernels. The performance of each model was evaluated based on ℓ_{0-1} loss, variance and runtime.

1 Introduction

The goal of this project was to classify labels based on numerical features using kernelized linear classification methods, implementing and testing various machine learning algorithms from scratch to assess their performance and to understand the trade-offs between different approaches.
You can find the code and the Project 1 instructions in the KernelLinearClassificationML GitHub repository.

2 Dataset Exploration and Preprocessing

2.1 Dataset Description

The dataset consists of 10,000 instances, each characterized by 10 numerical features (x_1 to x_{10}) and a binary target label y. The target variable y takes values of 1 or -1, meaning the problem is a binary classification task.

2.2 Data Preprocessing

To manage the computational limitations of my machine, I performed a random sampling of the dataset selecting a subset of 1,000 instances using the `sample` method, as shown below:

```
data = data.sample(n=1000, random_state=45)
```

where `data` is a Pandas data frame containing the imported data.
This step was necessary because the original dataset was too large for my PC to process efficiently, and reducing the dataset size allowed the algorithms to run faster while still providing meaningful insights for model training and evaluation. The `random_state` parameter was set to 45 to ensure reproducibility of the results.
Before training and testing each algorithm, I used the function `split_data()` to split the dataset into training and testing sets. The function first introduces a bias term by adding a column of ones to the feature matrix: including this bias term is a common practice in machine learning, as it provides greater flexibility and improves the model's ability to fit the data. The data points are then randomized through shuffling: this shuffling step ensures that both the training and testing sets are representative samples of the overall dataset, mitigating any potential biases. A bias term has been added by introducing a column of ones to the feature matrix. This step improves flexibility, allowing it to not pass from the origin, and fit, allowing the decision boundary to shift up or down. Finally, the dataset is split according to a specified training ratio set to 70% for training and 30% for testing.

3 Methodology

To ensure the robustness of my results and to measure the volatility of the model performance, each model was trained and tested 50 times. This approach allowed me to measure the variability in performance and obtain a more reliable estimate of each model's effectiveness. For each model, I followed the following procedure:

- **Initialization**: Each model was initialized with the same hyperparameters;
- **Training and Testing**: The model was trained on the training dataset and evaluated on the testing dataset 50 times. The dataset was split into training and testing sets for each run according to the fixed ratio;
- **Performance Measurement**: After each run, the ℓ_2 loss was recorded while the runtime was measured on all 50 runs.

In my implementation of the algorithms, I have included the predictions obtained during the training as an additional output. This feature is not typically provided by default in standard algorithm implementations.

3.1 Implemented Algorithms

3.1.1 The Perceptron

Pseudocode
The Perceptron algorithm can be summarized with the following pseudocode:

Algorithm 1 Perceptron Learning Algorithm

Input: Feature matrix X, target labels y, learning rate η, number of epochs T
Initialize weights $w \leftarrow 0$ ▷ Initialize weights to zero
for epoch $= 1$ to T **do**
    for each sample (x_i, y_i) in (X, y) **do**
        Compute prediction: $\hat{y}_i \leftarrow \text{sign}(w \cdot x_i)$
        if $\hat{y}_i \neq y_i$ **then**
            if $\hat{y}_i = -1$ and $y_i = 1$ **then**
                Update weights: $w \leftarrow w + \eta \cdot x_i$
            else if $\hat{y}_i = 1$ and $y_i = -1$ **then**
                Update weights: $w \leftarrow w - \eta \cdot x_i$
            end if
        end if
    end for
end for
Output: Final weights w, predictions $\hat{y} \leftarrow \text{sign}(X \cdot w)$

3.1.2 Support Vector Machines (SVMs) using Pegasos

Pseudocode The Pegasos (Primal Estimated sub-Gradient Solver for SVM) algorithm for training a Support Vector Machine (SVM) can be summarized with the following pseudocode:

Algorithm 2 Pegasos Algorithm for SVM

Input: Feature matrix X, target labels y, regularization parameter λ, number of epochs T, batch size B
Initialize weight vector $w \leftarrow 0$ ▷ Initialize weights to zero
Initialize iteration counter $t \leftarrow 0$
Initialize weight sum $w_{\text{sum}} \leftarrow 0$
for epoch = 1 to T **do**
    for each batch in the dataset **do**
        Increment iteration counter: $t \leftarrow t + 1$
        Compute learning rate: $\eta \leftarrow \frac{1}{\lambda t}$
        Randomly select a batch of size B
        for each sample (x_i, y_i) in the batch **do**
            if $y_i(w \cdot x_i) < 1$ **then**
                Update weights: $w \leftarrow (1 - \eta\lambda)w + \eta y_i x_i$
            else
                Update weights: $w \leftarrow (1 - \eta\lambda)w$
            end if
        end for
        Update weight sum: $w_{\text{sum}} \leftarrow w_{\text{sum}} + w$
    end for
end for
Compute final weights: $w \leftarrow \frac{w_{\text{sum}}}{t}$
Output: Final weights w, predictions $\hat{y} \leftarrow \text{sign}(X \cdot w)$

Mathematical Formulation
The goal of the SVM is to find a hyperplane that maximizes the margin between two classes while minimizing classification errors. The optimization problem can be expressed as:

$$\min_w \frac{\lambda}{2}\|w\|^2 + \frac{1}{n}\sum_{i=1}^{n}\max(0, 1 - y_i(w \cdot x_i)) \tag{1}$$

where:

- w is the weight vector to be optimized.
- λ is the regularization parameter that controls the trade-off between margin maximization and error minimization.
- y_i is the target label for the i-th training example.
- x_i is the feature vector for the i-th training example.

The Pegasos algorithm approximates the solution to this optimization problem using stochastic gradient descent (SGD). At each iteration, a subset (batch) of the data is used to compute an estimate of the gradient, which is then used to update the weight vector. The learning rate η is adjusted over time according to the iteration count t:

$$\eta_t = \frac{1}{\lambda t} \tag{2}$$

To improve stability and reduce variance, the algorithm keeps track of the cumulative sum of the weight vectors and computes the final weight vector as the average of all weight vectors across iterations.

3.1.3 Regularized Logistic Regression

Pseudocode The Regularized Logistic Regression algorithm can be summarized with the following pseudocode:

Algorithm 3 Regularized Logistic Regression with Stochastic Gradient Descent

Input: Feature matrix X, target labels y, regularization parameter λ, number of epochs T, batch size B, initial learning rate η_0, learning rate decay γ
Initialize weights $w \leftarrow$ random small values ▷ Initialize weights randomly
Transform target labels: $y \leftarrow (y + 1)/2$ ▷ Transform labels from {-1, 1} to {0, 1}
for epoch = 1 to T **do**
    Shuffle the data
    for each batch $X_{\text{batch}}, y_{\text{batch}}$ in (X, y) **do**
        Update learning rate: $\eta \leftarrow \eta_0/(1 + \gamma \cdot t)$ ▷ Decay learning rate
        Compute logits: logits $\leftarrow X_{\text{batch}} \cdot w$
        Apply sigmoid function: $\hat{y} \leftarrow \frac{1}{1+\exp(-\text{logits})}$
        Compute gradient: gradient $\leftarrow \frac{1}{B} \cdot (X_{\text{batch}}^{\top} \cdot (\hat{y} - y_{\text{batch}})) + \lambda \cdot w$
        Update weights: $w \leftarrow w - \eta \cdot \text{gradient}$
        Increment iteration counter $t \leftarrow t + 1$
    end for
end for
Transform predictions: $\hat{y} \leftarrow 2 \cdot (\hat{y} > 0.5) - 1$ ▷ Transform {0, 1} back to {-1, 1}
Output: Final weights w, predictions $\hat{y}$

Mathematical Formulation
The Regularized Logistic Regression algorithm I implemented involves the following steps initialize the weight vector w with small random values. I did this because the algorithm was giving me performance problems. Initializing to a random small value instead of zero can cause faster convergence.
Then it convert target labels y from {-1, 1} to {0, 1} to use the logistic regression.
For each batch of data, it computes the logits using:

$$\text{logits} = X \cdot w \tag{3}$$

It applies the sigmoid function to convert logits to probabilities:

$$\hat{y} = \frac{1}{1 + \exp(-\text{logits})} \tag{4}$$

Notice that the exponential argument `logit` has been cropped with a value between $[-709, +709]$ to avoid overflow and underflow. This is necessary because Python type `float64` can't handle values smaller than e^{-709} (they are considered 0) and bigger than e^{709}.
After the Gradient Computation and the Weight Update, the algorithm makes the final predictions by converting probabilities back to class labels {-1, 1}:

$$\hat{y} = 2 \cdot (\hat{y} > 0.5) - 1 \tag{5}$$

3.2 Polynomial Feature Expansion

In polynomial feature expansion of degree 2, we transform a dataset with original features $(x_1, x_2, \ldots, x_{10})$ to include additional features that capture non-linear relationships and interactions between the original features. This is accomplished by introducing new features that are:

1. **Squared Features:**
    - For each original feature x_i, we include its square x_i^2. This allows the model to capture the quadratic effects of the features.
2. **Pairwise Product Features:**
    - For each unique pair of features (x_i, x_j) where $i \neq j$, we include the product $x_i \cdot x_j$. This enables the model to learn interactions between different features.

The expanded dataset thus has:

- 10 original features
- 10 squared features
- 45 pairwise product features

This results in a total of $10 + 10 + \frac{10 \times 9}{2} = 65$ features.

3.3 Kernel Methods

3.3.1 Kernelized Perceptron

Pseudocode The Kernelized Perceptron algorithm can be summarized with the following pseudocode:

Algorithm 4 Kernel Perceptron

Input: Feature matrix x, target labels y, learning rate η, number of epochs E, σ (for Gaussian kernel), `kernel` ('gaussian' or 'polynomial'), c (constant for polynomial kernel), d (degree for polynomial kernel)
Initialize weights (alphas) $\alpha \leftarrow \mathbf{0}$ ▷ Initialize weights to zero
Compute Kernel Matrix:
if `kernel` = 'gaussian' **then**
    for $i = 1$ **to** n **do**
        for $j = i$ **to** n **do**
            $K[i, j] \leftarrow \exp\left(-\frac{\|x[i]-x[j]\|^2}{2\sigma^2}\right)$
        end for
        $K \leftarrow K + K^T - \text{diag}(K)$ ▷ Make matrix symmetric
    end for
else if `kernel` = 'polynomial' **then**
    for $i = 1$ **to** n **do**
        for $j = i$ **to** n **do**
            $K[i, j] \leftarrow (c + x[i] \cdot x[j])^d$
        end for
        $K \leftarrow K + K^T - \text{diag}(K)$ ▷ Make matrix symmetric
    end for
end if
for epoch $= 1$ **to** E **do**
    for $i = 1$ **to** n **do**
        Compute prediction: $\hat{y}_i \leftarrow \sum_{j=1}^{n} \alpha_j y_j K_{ij}$
        if $y_i \cdot \hat{y}_i \leq 0$ **then**
            Update weight: $\alpha_i \leftarrow \alpha_i + \eta$
        end if
    end for
end for
Compute final predictions: $\hat{y} \leftarrow \text{sign}\left(\sum_{i=1}^{n} \alpha_i y_i K\right)$
Output: Weights (alphas) α, predictions $\hat{y}$

Mathematical Formulation
The Kernel Perceptron algorithm extends the basic perceptron by incorporating a kernel function to map input features into a higher-dimensional space. This allows the perceptron to learn complex decision boundaries.
The kernel matrix K measures the similarity between pairs of input samples in the feature space transformed by the kernel function.
The Kernel matrix is always symmetric, the matrix K is adjusted by:

$$K \leftarrow K + K^T - \text{diag}(K) \tag{6}$$

where K^T is the transpose of K and $\text{diag}(K)$ is a diagonal matrix containing the diagonal elements of K. This adjustment makes the computation faster.

3.3.2 Kernelized Pegasos for SVM

Pseudocode According to [1] The Kernelized Pegasos algorithm can be summarized with the following pseudocode:

Algorithm 5 Kernel Pegasos Algorithm

Input: Dataset S, regularization parameter λ, number of iterations T
Initialize dual variables (alphas) $\alpha_1 \leftarrow \mathbf{0}$ ▷ Initialize alphas to zero
for $t = 1$ **to** T **do**
  Choose $i_t \in \{1, \ldots, |S|\}$ uniformly at random ▷ Randomly select an index
  for all $j = i_t$ **do**
    Set $\alpha_{t+1}[j] \leftarrow \alpha_t[j]$
  end for
  if $y_{i_t} \left(\frac{1}{\lambda t} \sum_j \alpha_t[j] y_{i_t} K(x_{i_t}, x_j) \right) < 1$ **then**
    Update α:

$$\alpha_{t+1}[i_t] \leftarrow \alpha_t[i_t] + 1$$

  else
    No update:

$$\alpha_{t+1}[i_t] \leftarrow \alpha_t[i_t]$$

  end if
end for
Output: Dual variables α_{T+1}

However, there is a typo (red) in the pseudocode. Here's the corrected version:

Algorithm 6 Kernel Pegasos Algorithm

Input: Dataset S, regularization parameter λ, number of iterations T
Initialize dual variables (alphas) $\alpha_1 \leftarrow \mathbf{0}$ ▷ Initialize alphas to zero
for $t = 1$ **to** T **do**
    Choose $i_t \in \{1, \ldots, |S|\}$ uniformly at random ▷ Randomly select an index
    for all $j = 1$ **to** $|S|$ **do**
        Set $\alpha_{t+1}[j] \leftarrow \alpha_t[j]$
    end for
    if $y_{i_t} \left(\frac{1}{\lambda t} \sum_{j=1}^{|S|} \alpha_t[j] y_j K(x_{i_t}, x_j) \right) < 1$ **then**
        Update α:

$$\alpha_{t+1}[i_t] \leftarrow \alpha_t[i_t] + 1$$

    else
        No update:

$$\alpha_{t+1}[i_t] \leftarrow \alpha_t[i_t]$$

    end if
end for
Output: Dual variables α_{T+1}

To determine whether a row is correctly classified or not, we need to evaluate the decision function using the labels y_j of all data points and not just y_{i_t}, the batch labels.

4 Hypertuning

The goal is to find the best set of hyperparameters that lead to the best model performance.

4.1 Hypertuning Process

The hypertuning process I used involves the following steps:

- **Parameter Grid Definition:** A grid of hyperparameters to explore was defined. This grid includes various values for each hyperparameter that will be evaluated;
- **Cross-Validation Splitting:** the dataset was split into k folds. For each fold, one part was used as the validation set, and the remaining parts were combined to form the training set;
- **Model Evaluation:** For each combination of hyperparameters, the model was trained using the training folds and evaluated on the validation fold;
- **Average Loss Calculation:** The average loss across all folds for each hyperparameter combination was computed to assess its performance;
- **Best Parameter Selection:** The combination of hyperparameters that results in the lowest average loss was identified;

4.2 Results for Each Algorithm

4.2.1 Perceptron

For the Perceptron algorithm, the grid search was performed over learning rates and the number of epochs. The best hyperparameters found were:

- **Learning Rate (η):** 0.001
- **Number of Epochs:** 500

4.2.2 Pegasos

The grid search for the Pegasos algorithm involved tuning the regularization parameter, number of epochs, and batch size. The optimal hyperparameters were:

- **Regularization Parameter (λ):** 1
- **Number of Epochs:** 500
- **Batch Size:** 100

4.2.3 Logistic Regression

For Regularized Logistic Regression, the grid search focused on the regularization parameter, number of epochs, and batch size. The best hyperparameters were:

- **Regularization Parameter (λ):** 1
- **Number of Epochs:** 500
- **Batch Size:** 100

4.2.4 Perceptron (Polynomial Features)

For the Perceptron algorithm, the grid search was performed over learning rates and the number of epochs. The best hyperparameters found were:

- **Learning Rate (η): 0.001**
- **Number of Epochs: 50**

Notice that it needs fewer epochs

4.2.5 Pegasos (Polynomial Features)

The grid search for the Pegasos algorithm involved tuning the regularization parameter, number of epochs, and batch size. The optimal hyperparameters were:

- **Regularization Parameter (λ): 1**
- **Number of Epochs: 500**
- **Batch Size: 100**

4.2.6 Logistic Regression (Polynomial Features)

For Regularized Logistic Regression, the grid search focused on the regularization parameter, number of epochs, and batch size. The best hyperparameters were:

- **Regularization Parameter (λ): 0.01**
- **Number of Epochs: 500**
- **Batch Size: 500**

4.2.7 Kernel Perceptron

The hyperparameter tuning for the Kernel Perceptron algorithm involved optimizing the kernel type, learning rate, and kernel-specific parameters. The best hyperparameters for each kernel type are as follows:

Gaussian Kernel:

- **Learning Rate (η):** 0.001
- **epochs:** 50
- **Width of the Gaussian function (σ):** 2.0

Polynomial Kernel:

- **Learning Rate (η):** 0.001
- **epochs:** 500
- **Bias (c):** 1.0
- **Degree (d):** 3

4.2.8 Kernel Pegasos

The hyperparameter tuning for the Kernel Pegasos algorithm involved optimizing parameters similar to those for Kernel Perceptron, adding regularization and batch size. The optimal hyperparameters for each kernel type are as follows:

Gaussian Kernel:

- **Regularization Parameter (λ):** 0.001
- **epochs:** 100
- **Width of the Gaussian function (σ):** 2.0
- **Batch Size:** 50

Polynomial Kernel:

- **Regularization Parameter (λ):** 0.01
- **epochs:** 500
- **Batch Size:** 10
- **Bias (c):** 0.5
- **Degree (d):** 3

5 Models Performance

In this section, I'll show the performance of the models, based on:

- **Error:** measured with the 0-1 loss (ℓ_{0-1})

$$\ell_{0-1}(y_i, \hat{y}_i) = \begin{cases} 0 & \text{if } y_i = \hat{y}_i \\ 1 & \text{if } y_i \neq \hat{y}_i \end{cases} \tag{7}$$

The overall 0-1 loss L_{0-1} is then given by:

$$L_{0-1} = \frac{1}{n} \sum_{i=1}^{n} \ell_{0-1}(y_i, \hat{y}_i) \tag{8}$$

- **Error Variance:**
  Let $L_{0-1} = \{L_{0-1,1}, L_{0-1,2}, \ldots, L_{0-1,10}\}$ be a list of 0-1 loss values. The variance of these values can be computed using:
  $$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(L_{0-1,i} - \bar{L}_{0-1})^2 \tag{9}$$
  where $\bar{L}_{0-1}$ is the mean of the 0-1 loss values:
  $$\bar{L}_{0-1} = \frac{1}{n}\sum_{i=1}^{n} L_{0-1,i} \tag{10}$$

- **Runtime:** the time it takes to train and test each algorithm (T).

Every algorithm has been run **50 times**, to detect the variance of the Error.

5.1 Perceptron, Pegasos and Regularized Logistic Classification

Perceptron

Training

- L_{0-1}: 36.45%
- σ^2: 0.0063

Test

- L_{0-1}: 36.69%
- σ^2: 0.0060

T: 0m 0s 596ms

Pegasos

Training

- L_{0-1}: 45.22%
- σ^2: 0.0063

Test

- L_{0-1}: 44.61%
- σ^2: 0.0060

T: 0m 1s 378

Regularized Logistic Classification

Training

- L_{0-1}: 29.84%
- σ^2: 0.0007

Test

- L_{0-1}: 29.52%
- σ^2: 0.0013

T: 0m 0s 72ms

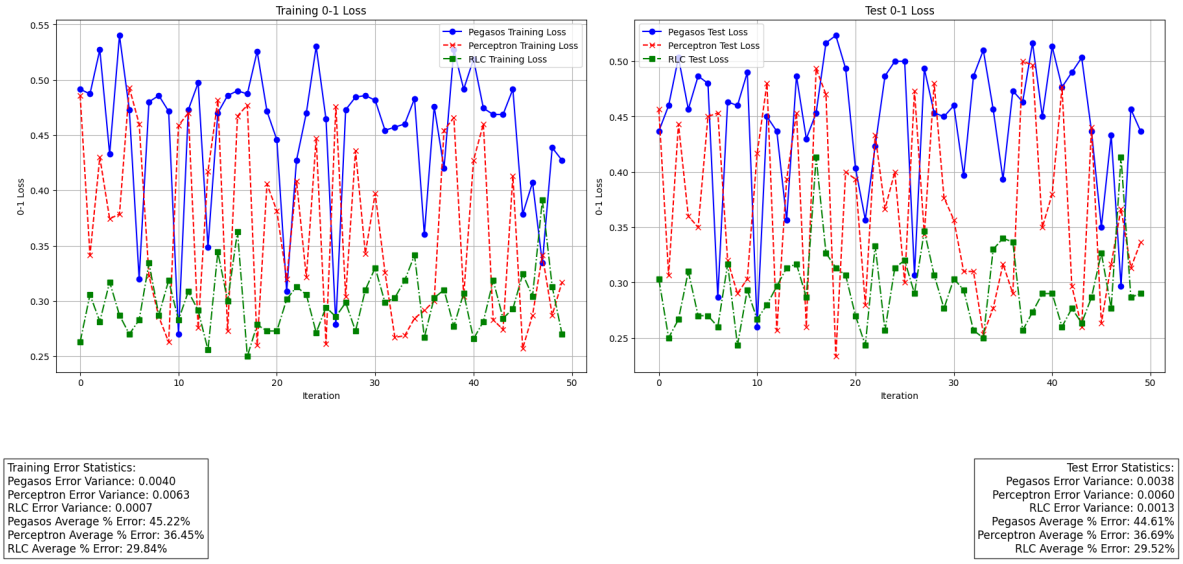

Figure 1: Perceptron, Pegasos and Regularized Logistic Classification Error Distribution

The Regularized Logistic Classification is by far the best-performing algorithm among the three. It excels in error rate, variance and runtime, in both training and test.

5.2 Perceptron, Pegasos and Regularized Logistic Classification (Polynomial Expansion)

Perceptron

Training

- L_{0-1}: 35.73%
- σ^2: 0.0057

T: 0m 0s 569ms

Test

- L_{0-1}: 35.57%
- σ^2: 0.0044

Pegasos

Training

- L_{0-1}: 44.15%
- σ^2: 0.0055

T: 0m 1s 680ms

Test

- L_{0-1}: 44.74%
- σ^2: 0.0071

Regularized Logistic Classification

Training

- L_{0-1}: 28.75%
- σ^2: 0.0007

T: 0m 0s 167ms

Test

- L_{0-1}: 29.58%
- σ^2: 0.0006

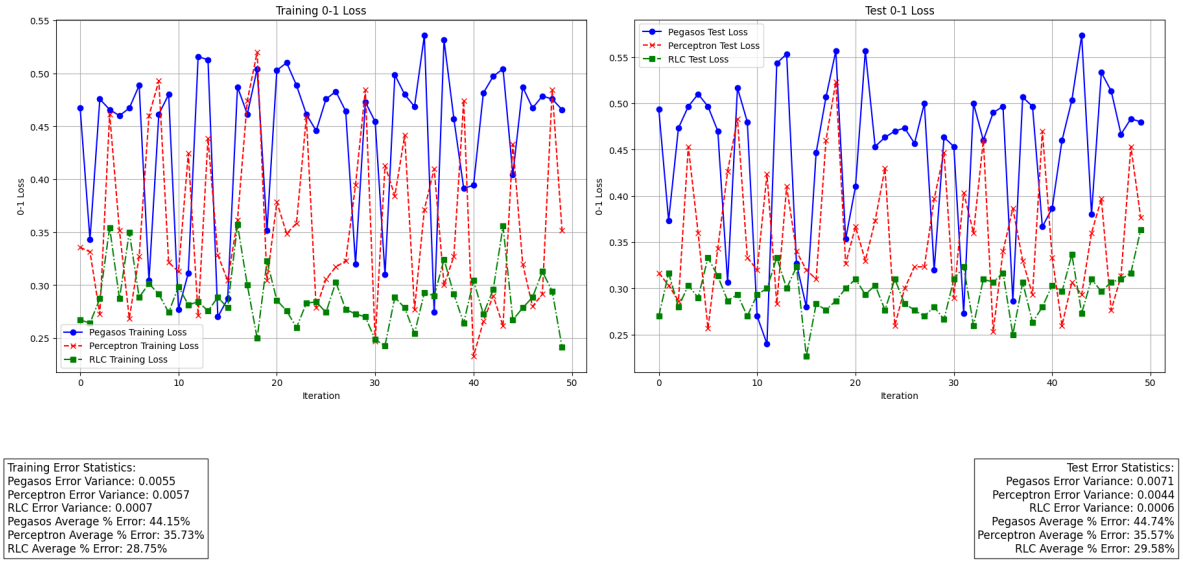

Figure 2: Perceptron, Pegasos and Regularized Logistic Classification Error Distribution after Polynomial Expansion

Perceptron shows a slight improvement in training error after polynomial expansion, but the test error remains similar. Pegasos experiences a slight improvement in training error but a slight increase in test error. Regularized Logistic Classification shows the lowest training error after polynomial the expansion. However, its test error slightly increases.
The Runtime is similar and the differences are probably due to unrelated machine processes.
In summary, **Regularized Logistic Classification** remains the best-performing algorithm in both scenarios, demonstrating better accuracy and consistency.

5.3 Kernelized Perceptron

Gaussian

Training

- L_{0-1}: 0%
- σ^2: 0

T: 0m 2s 383ms

Test

- L_{0-1}: 10.77%
- σ^2: 0.0002

Polynomial

Training

- L_{0-1}: 34.87%
- σ^2: 0.0081

T: 0m 3s 127ms

Test

- L_{0-1}: 47.09%
- σ^2: 0.0051

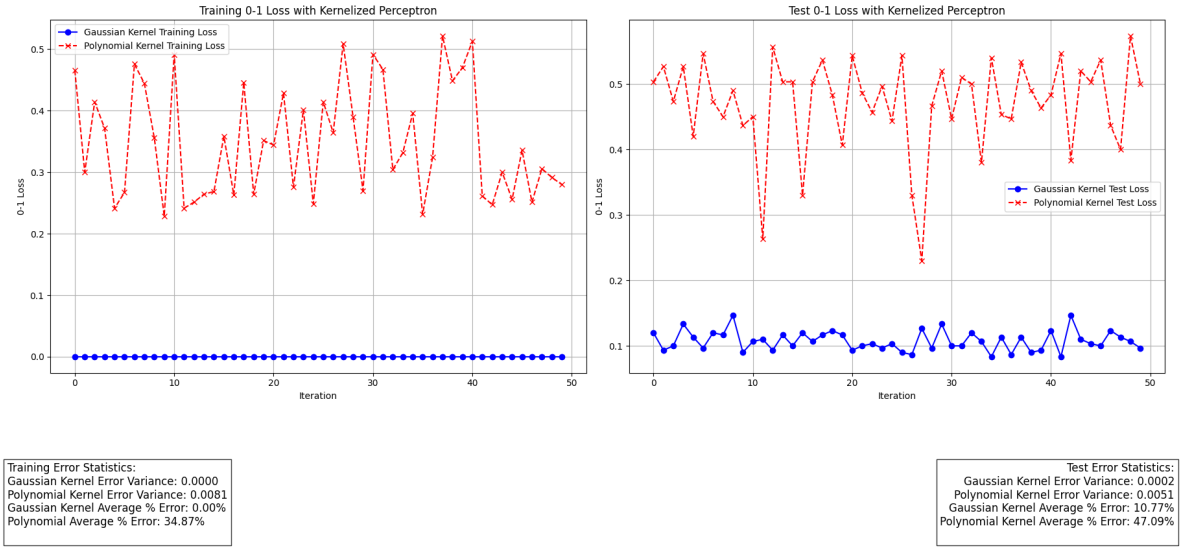

Figure 3: Kernalized Perceptron with Gaussian and Polynomial Kernel

The Kernelized Perceptron performed better with a **Gaussian Kernel** in both training and test indices. It also has a lower runtime. However, the training error of 0% suggests potential overfitting. Besides, the algorithm has the lowest test error even when comparing it to the previous algorithms.

5.4 Kernelized Pegasos

Gaussian

Training

- L_{0-1}: 0%
- σ^2: 0

T: 0m 2s 711ms

Test

- L_{0-1}: 9.65
- σ^2: 0.0003

Polynomial

Training

- L_{0-1}: 35.74%
- σ^2: 0.0069

T: 0m 3s 926ms

Test

- L_{0-1}: 48.87%
- σ^2: 0.0024

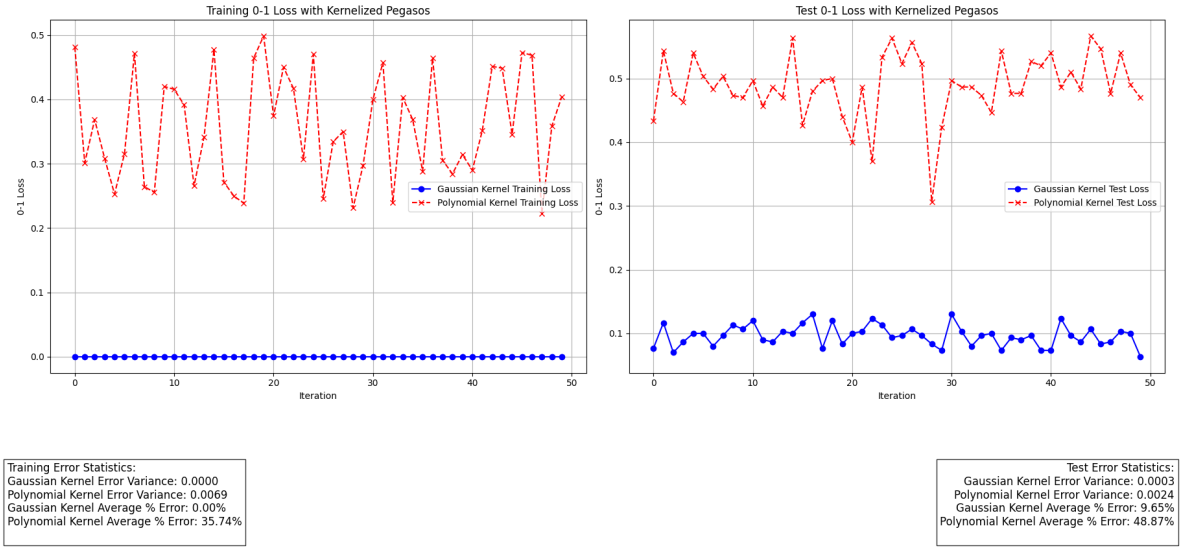

Figure 4: Kernalized Pegasos with Gaussian and Polynomial Kernel

Kernelized Pegasos and Kernelized Perceptron shows similar results: the **Gaussian kernel** achieves a perfect 0% which might suggest potential overfitting, as the model may have learned the training data too well. Nevertheless, the Gaussian kernel performs well on the test set, with a relatively low error rate of 9.65% and the tiniest variance. This means that, despite the risk of overfitting, the model generalizes effectively to unseen data. In contrast, the Polynomial kernel exhibits a much higher training error of 35.74% and a significantly higher test error of 48.87%. The gap between training and test errors combined with a test error close to random classification, indicates that the polynomial kernel doesn't perform well with this dataset.

6 Conclusion

In this report, I have evaluated the following classification algorithms: Perceptron, Pegasos, Regularized Logistic Classification, Kernelized Perceptron, and Kernelized Pegasos based on their performance in terms of error, error variance, and runtime.
The Perceptron shows decent performance in terms of error but is outperformed by the Regularized Logistic Classification. In addition, it has a relatively high variance.
Pegasos shows higher error rates compared to the other algorithms, both in the training and testing phases. This indicates that it may not be as effective at minimizing classification errors. In addition, its runtime it's higher than the other similar algorithms in terms of complexity.
The Regularized Logistic Classification algorithm consistently outperforms the Perceptron and Pegasos in terms of error and variance. It also shows the fastest runtime. Using a polynomial expanded dataset, Perceptron shows a slight improvement in training error after polynomial expansion, but the test error remains similar. Pegasos experiences a slight improvement in training error but a slight increase in test error. On the other hand, Regularized Logistic Classification shows the lowest training error after polynomial expansion. However, its test error slightly increases. The Runtime is similar for all the algorithms before and after the polynomial expansion; the differences are probably due to unrelated machine processes. **Regularized Logistic Classification still is the best-performing algorithm** in both scenarios.
The Kernelized Perceptron with the Gaussian kernel achieves a perfect training error of 0%, which suggests **overfitting**. Despite this, it performs well on the test set with a low error rate observed (10.77%). The Polynomial kernel, however, shows a suboptimal performance with high training and test errors.
Similar to Kernelized Perceptron, the Kernelized Pegasos with the Gaussian kernel performs excellently in terms of test error and variance, showing the tinest numbers (9.65% and 0.0003). Nevertheless, it shows a training error of 0% indicative of possible **overfitting**. The Polynomial kernel performs poorly,

with both high training and test errors.

6.1 Evaluation

The **Regularized Logistic Classification is the most consistent and efficient algorithm across the board**: it offers the best balance of low error rates, low variance, and efficient runtime. The Kernelized Perceptron and Kernelized Pegasos with Gaussian may suffer from overfitting due to perfect training errors. However, they show the best test performances. On the other hand, the Polynomial kernels in both Kernelized Perceptron and Pegasos, as well as Pegasos in general, are less effective, exhibiting error rates close to the random classification and high variances.

7 References

References

[1] Shalev-Shwartz, S., Singer, Y., Srebro, N. et al. Pegasos: primal estimated sub-gradient solver for SVM. *Mathematical Programming*, 127, 3–30 (2011). https://doi.org/10.1007/s10107-010-0420-4.

8 Appendix

I declare that this material, which I now submit for assessment, is entirely my own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my work. I understand that plagiarism, collusion, and copying are grave and serious offences in the university and I accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. This assignment, or any part of it, has not been previously submitted by me or any other person for assessment on this or any other course of study.