Kernelized Linear Classification

Mariasilvia Pancione - 32440A

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 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.

Contents

1 Introduction

The goal of this project is to develop and compare various algorithms for binary classification. The entire workflow was carried out from scratch, without using machine learning libraries such as Scikit-learn.

The project begins with an exploratory analysis and preprocessing phase, ensuring clean separation between training and test sets to prevent data leakage. Three linear models were then implemented: the Perceptron, the Support Vector Machines using the Pegasos algorithm, and the regularized logistic classifier based on the Pegasos optimization framework.

To enhance model performance, a polynomial feature expansion of degree 2 was applied, allowing the models to learn more complex decision boundaries. The impact of this expansion was evaluated both in terms of predictive accuracy and feature weight interpretation.

Subsequently, kernel methods were introduced as an alternative way to model non-linear relationships. The Kernelized Perceptron and Kernelized Pegasos were implemented using both Gaussian and polynomial kernels.

Hyperparameter tuning played a central role throughout the project and was performed systematically using 5-fold cross-validation, ensuring fair and consistent comparison across all models.

A final evaluation was conducted to compare the performance of all models, providing insights into the trade-offs between model complexity, accuracy, and computational cost, with the aim of identifying the most effective approach in this context.

2 Data Exploration and Preprocessing

A series of preprocessing steps was applied to the dataset in order to ensure data quality, prevent data leakage, and enable reliable model training. The dataset consists of 10,000 observations and 11 variables, including ten input features $(x_1, \ldots, x_{10})$ and a binary target variable $y \in \{-1, 1\}$.

2.1 Missing Values

A preliminary inspection was performed to detect any missing values in the dataset. No missing entries were found in any column, so no imputation or removal was required.

2.2 Variable Distributions

To gain an initial understanding of the data, the distributions of the variables were analyzed using histograms, as shown in Figure 1. Some features exhibit an approximately normal distribution (e.g., `x2`, `x7`, `x9`), while others appear skewed or multi-modal (e.g., `x1`, `x4`, `x10`), suggesting the need for transformations in subsequent modeling phases.

2.3 Train/Test Split

To prevent data leakage and ensure a fair evaluation of model performance, the dataset was split into training and test sets before any preprocessing was applied. An 80/20 split was used, with 80% of the data assigned to the training set and 20% to the test set. All preprocessing steps were performed exclusively on the training set. The parameters learned from these transformations (e.g., means and standard deviations for scaling) were then applied to the test set. This procedure guarantees that no information from the test set leaks into the training phase, which could otherwise lead to over-optimistic performance estimates and compromise the validity of the evaluation.

2.4 Outlier Detection

Boxplots of each feature in the training set were used to detect outliers (Figure 2). Several features showed extreme values beyond the whiskers, especially `x1`, `x2`, `x4`, `x7`, `x8`, and `x9`. Based on this analysis, 435 outliers were removed from the training set to reduce noise and enhance the robustness of subsequent models.

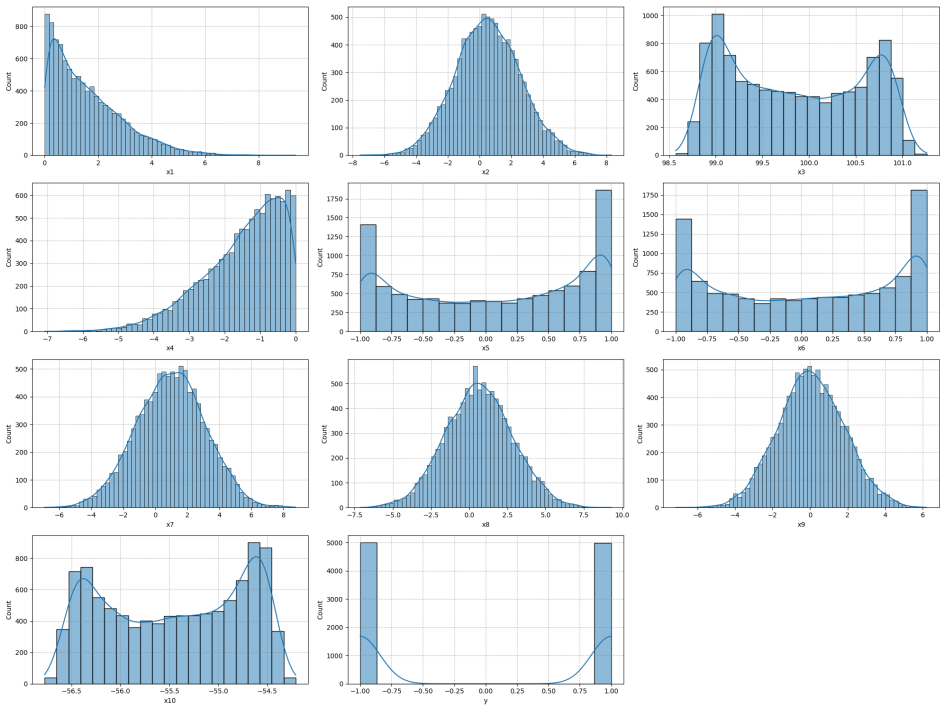

Figure 1: Distribution of the variables in the original dataset.

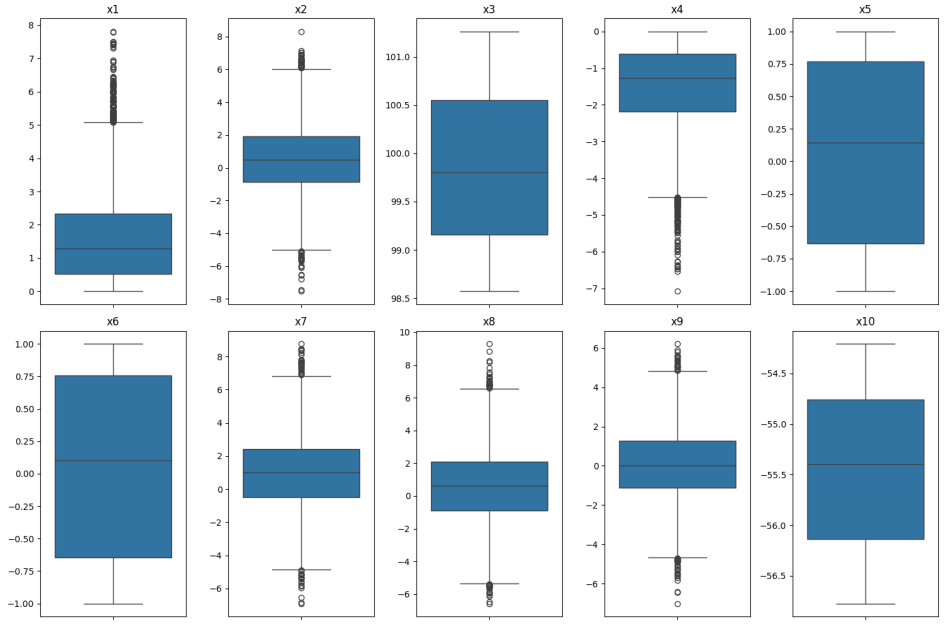

Figure 2: Boxplots of the features in the training set.

2.5 Feature Correlation

To detect multicollinearity among features, a correlation matrix was computed using Pearson coefficients on the training set. As shown in Figure 3, strong correlations were detected among `x3`, `x6`, and `x10`. In particular, `x3` is strongly negatively correlated with both `x6` ($\rho = -0.99$) and `x10` ($\rho = -0.98$), while `x6` and `x10` are positively correlated ($\rho = 0.99$). Such high correlations indicate redundancy, which can negatively affect model performance and lead to instability in parameter estimation. To address this issue, features `x6` and `x10` were removed, retaining `x3` as the representative variable.

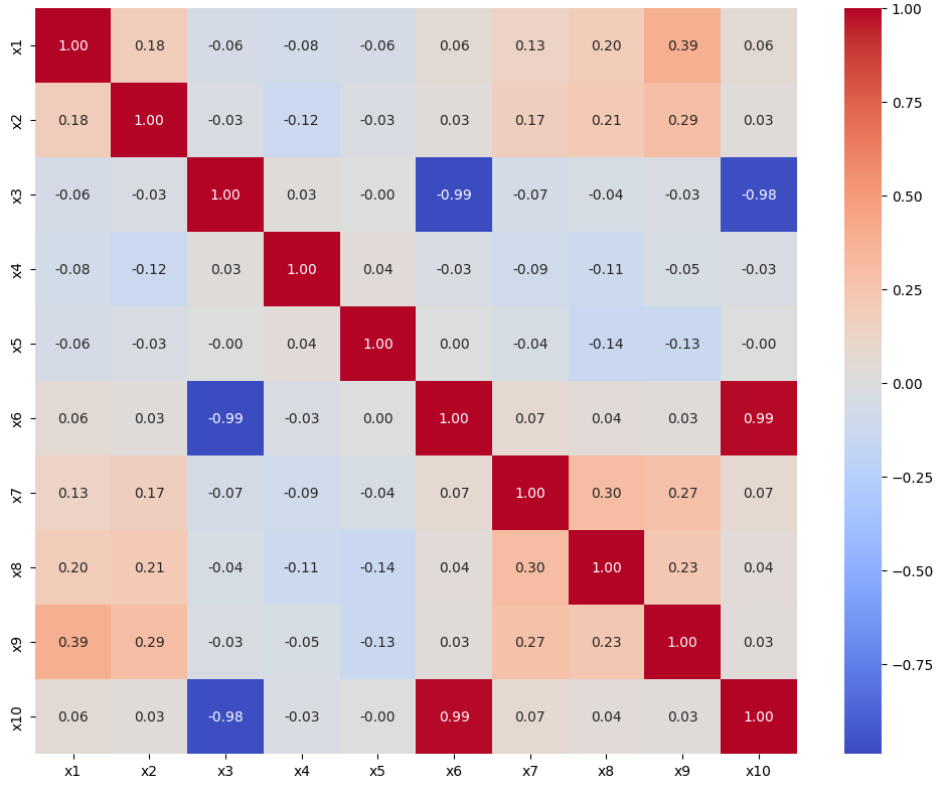

Figure 3: Correlation matrix of the features in the training set.

2.6 Feature Scaling

Standardization was applied to ensure that all features are on a comparable scale and to prevent those with larger numerical ranges from dominating the learning process. This transformation centers each variable around zero and scales it to have unit variance, making the dataset suitable for models sensitive to feature magnitudes. To prevent data leakage, the mean and standard deviation used for standardization were computed exclusively from the training set. These parameters were then applied to transform both the training and test sets, ensuring that information from the test set does not influence the training phase.

2.7 Label Distribution

Finally, the distribution of the binary target variable was evaluated in both the training and test sets to verify class balance. The proportions are summarized in Table 1.

Label	Training Set	Test Set
-1	51.80%	49.60%
1	48.20%	50.40%

Table 1: Distribution of the target variable in the training and test sets.

The label distribution is reasonably balanced in both subsets, indicating that no further adjustments (e.g., resampling) were required.

3 Algorithms Implementation

To ensure a fair and robust comparison of the models, a consistent training and evaluation procedure was adopted across all algorithms. Hyperparameter tuning was performed using k-fold cross-validation with $k = 5$ on the training set.

In k-fold cross-validation, the training data is partitioned into k equal-sized folds. For each round, one fold is used as the validation set, while the remaining $k - 1$ folds are used for training. This process is repeated k times so that each fold is used exactly once as validation. The performance across all folds is then averaged to obtain a reliable estimate of the model's generalization ability. This methodology helps reduce the variance associated with a single train/validation split and allows for effective hyperparameter selection.

Once the optimal hyperparameters were identified, each model was retrained on the full training set using these parameters. The final performance was then evaluated on the held-out test set.

Model performance was assessed using the 0-1 loss, a standard metric for classification tasks. It is defined as:

$$\ell(y, \hat{y}) = \begin{cases} 0 & \text{if } y = \hat{y} \\ 1 & \text{otherwise} \end{cases}$$

where y is the true label and $\hat{y}$ is the predicted label. This function returns 0 when the prediction is correct and 1 otherwise. To evaluate overall model performance, the 0-1 loss is averaged over all instances in the test set, resulting in the misclassification rate, which quantifies the proportion of incorrectly classified examples.

3.1 Perceptron

The Perceptron is a simple yet foundational algorithm for learning linear classifiers. It aims to find a homogeneous separating hyperplane, defined by a weight vector w, that correctly classifies all training examples. It proceeds by iteratively scanning the training set: for each input x_t, if the current classifier misclassifies the example, that is, if:

$$y_t \, \mathbf{w}^\top \mathbf{x}_t \leq 0,$$

the weight vector is updated according to the following rule:

$$\mathbf{w} \leftarrow \mathbf{w} + y_t \, \mathbf{x}_t$$

This update shifts the decision boundary in the direction that would correct the misclassification. The algorithm starts with an initial weight vector $w = 0$ and continues performing updates until no further misclassifications occur during a full pass over the training set (an epoch). If this condition is met, convergence is achieved and the resulting w defines a separating hyperplane.

It is important to note that the Perceptron is guaranteed to converge in a finite number of steps only if the data is linearly separable. In non-separable cases, the algorithm may fail to terminate, which motivates the need for modifications such as early stopping or regularization in practical applications.

Table 2 reports the results of the 5-fold cross-validation performed to select the optimal number of training epochs. Choosing an appropriate value is crucial: too few epochs may result in underfitting, while too many can lead to oscillations and poor generalization, especially in the presence of noisy or non-separable data.

Epochs	**CV 0-1 Loss**
50	0.3338
100	0.3277
500	**0.3068**
1000	0.3212

Table 2: Cross-validation results for Perceptron.

The best performance was obtained with 500 epochs, corresponding to a cross-validated misclassification rate of 0.3068. Based on this result, the Perceptron model was retrained on the entire training

set using 500 as the maximum number of epochs. However, the algorithm did not converge within the specified number of iterations, meaning that updates continued to occur throughout all epochs without reaching a fully separating hyperplane. This behavior is expected in non-linearly separable datasets, for which the Perceptron is not guaranteed to find a stable solution. The final performance evaluation yielded:

- **Training misclassification rate:** 0.3734
- **Test misclassification rate:** 0.3615

The relatively high and similar misclassification rates on both the training and test sets suggest that the model is underfitting the data. This is consistent with the limitations of the Perceptron in non-linearly separable settings, where the algorithm fails to capture more complex decision boundaries.

3.2 Support Vector Machines with the Pegasos Algorithm

Support Vector Machines (SVMs) are powerful supervised learning algorithms used for binary classification tasks. They aim to find a linear decision boundary that separates the two classes with the largest possible margin, while also minimizing classification errors through regularization. This balance between margin maximization and error minimization makes SVMs particularly effective in high-dimensional settings.

In this project, SVMs were implemented using the Pegasos algorithm (Primal Estimated sub-GrAdient SOlver for SVM), an optimization method based on stochastic sub-gradient descent. Pegasos is designed to handle large-scale datasets efficiently by updating the model parameters using one randomly selected training example at each iteration, rather than processing the entire dataset at once. This makes it computationally lighter and faster than traditional approaches that rely on solving quadratic programming problems.

The algorithm starts by initializing the weight vector $\mathbf{w}_1 = \mathbf{0}$, and proceeds for T iterations. At each step t, a training example $(\mathbf{x}_{Z_t}, y_{Z_t})$ is randomly sampled, and the weight vector is updated according to the sub-gradient of the hinge loss:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta_t \, \nabla \ell_{Z_t}(\mathbf{w}_t)$$

where η_t is the learning rate and ℓ_{Z_t} is the hinge loss evaluated on the selected example. The final output is the average of all weight vectors computed over the T rounds:

$$\mathbf{w} = \frac{1}{T} \sum_{t=1}^{T} \mathbf{w}_t$$

Two key hyperparameters influence the performance of the Pegasos algorithm: the number of iterations T and the regularization parameter λ. The number of iterations T determines how many update steps the algorithm performs and thus controls the duration and depth of the learning process. A small value may result in an insufficiently trained model, while an excessively large value can lead to unnecessary computation or even overfitting to the training data. The regularization parameter λ, on the other hand, controls the balance between maximizing the margin and minimizing the classification error. Smaller values of λ place more emphasis on fitting the training data, potentially reducing bias but increasing the risk of overfitting, whereas larger values promote simpler models with wider margins, at the cost of possibly underfitting the data.

The hyperparameters were selected via 5-fold cross-validation. As reported in Table 3, the best performance was achieved with $\lambda = 0.01$ and 30,000 iterations, corresponding to a cross-validated misclassification rate of 0.2825.

Based on this result, the Pegasos SVM was retrained on the full training set using the optimal hyperparameters. The final evaluation on both training and test sets yielded:

- **Training misclassification rate:** 0.2824
- **Test misclassification rate:** 0.2790

λ	**Iterations**	**CV 0-1 Loss**
0.0001	5000	0.3019
0.0001	10000	0.2940
0.0001	20000	0.2890
0.0001	30000	0.2921
0.001	5000	0.2999
0.001	10000	0.2896
0.001	20000	0.2915
0.001	30000	0.2854
0.01	5000	0.2841
0.01	10000	0.2838
0.01	20000	0.2851
0.01	**30000**	**0.2825**
0.1	5000	0.2853
0.1	10000	0.2826
0.1	20000	0.2826
0.1	30000	0.2830

Table 3: Cross-validation results for Pegasos.

The training and test misclassification rates are very close (0.2824 and 0.2790, respectively), suggesting that the model generalizes well to unseen data. There is no indication of overfitting, which confirms the effectiveness of the regularization introduced by the Pegasos algorithm.

Compared to the Perceptron, the Pegasos SVM achieved significantly better performance on both the training and test sets. While the Perceptron exhibited underfitting and failed to converge due to the non-separable nature of the data, Pegasos successfully addressed the same scenario by incorporating regularization through a soft-margin objective, which allows some classification errors in order to improve generalization.

3.3 Regularized Logistic Classification

Regularized logistic classification is a linear classification algorithm that combines the logistic loss with an ℓ_2 regularization term to improve generalization and prevent overfitting. This classifier was implemented using the same optimization framework as Pegasos, with one key difference: the hinge loss was replaced by the logistic loss. The objective function minimized by the algorithm is given by:

$$\frac{\lambda}{2}\|\mathbf{w}\|^2 + \frac{1}{n}\sum_{i=1}^{n} \log\left(1 + e^{-y_i \mathbf{w}^\top \mathbf{x}_i}\right)$$

where $\mathbf{w}$ is the weight vector, λ is the regularization parameter, n is the number of training examples, and $y_i \in \{-1, 1\}$ are the true labels.

The algorithm starts with an initial weight vector $\mathbf{w}_1 = \mathbf{0}$ and performs a fixed number of stochastic gradient descent updates. At each iteration, a training example is sampled uniformly at random, and the weights are updated according to the gradient of the logistic loss.

As in the Pegasos SVM, two hyperparameters must be tuned to optimize performance: the number of iterations T, which determines the number of updates, and the regularization parameter λ, which controls the trade-off between model complexity and fit to the training data.

The hyperparameters were selected using 5-fold cross-validation. As shown in Table 4, the best performance was obtained with $\lambda = 0.001$ and 30,000 iterations, achieving a cross-validated misclassification rate of 0.2835.

Based on the cross-validation results, the regularized logistic classifier was retrained on the full training set using the optimal hyperparameters. The final evaluation yielded the following performance:

- **Training misclassification rate:** 0.2838
- **Test misclassification rate:** 0.2810

Regularization parameter λ	Iterations	CV 0-1 Loss
0.0001	5000	0.3011
0.0001	10000	0.2933
0.0001	20000	0.2899
0.0001	30000	0.2886
0.001	5000	0.2954
0.001	10000	0.2882
0.001	20000	0.2917
0.001	**30000**	**0.2835**
0.01	5000	0.2857
0.01	10000	0.2861
0.01	20000	0.2861
0.01	30000	0.2839
0.1	5000	0.2851
0.1	10000	0.2854
0.1	20000	0.2847
0.1	30000	0.2868

Table 4: Cross-validation results for regularized logistic classification.

The performance is very similar to that of the Pegasos SVM, both in terms of training and test misclassification rates. This suggests that the choice between hinge and logistic loss does not significantly affect results in this case, provided that proper regularization is applied.

4 Polynomial Feature Expansion

Polynomial feature expansion is a technique used to increase the expressive power of linear models by introducing non-linear relationships between the input variables. Rather than modifying the model structure, this approach augments the input space by generating additional features based on polynomial combinations of the original ones.

In this project, a second-degree polynomial expansion was applied to the dataset. This transformation included all squared terms (e.g., x_i^2) and pairwise interaction terms (e.g., $x_i x_j$), resulting in a higher-dimensional feature space that allows linear models to capture more complex decision boundaries.

Given a feature vector $\mathbf{x} \in \mathbb{R}^d$, the expanded vector $\phi(\mathbf{x}) \in \mathbb{R}^{d'}$ includes all monomials of degree up to 2:

$$\phi(\mathbf{x}) = [x_1, x_2, \ldots, x_d, x_1^2, x_1 x_2, \ldots, x_d^2]$$

The expanded feature set was then used to retrain all three classification models—Perceptron, Pegasos SVM, and Regularized Logistic Regression—in order to evaluate whether the inclusion of non-linear features improves predictive performance.

4.1 Polynomial Perceptron

Cross-validation was performed to select the optimal number of epochs:

Epochs	CV 0-1 Loss
50	**0.0694**
100	0.0719
500	0.0702
1000	0.0726

Table 5: Cross-validation results for the Polynomial Perceptron.

The best performance was obtained with 50 epochs, corresponding to a cross-validated misclassification rate of 0.0694. Based on this result, the model was retrained on the entire training set using this setting. Final evaluation yielded the following results:

- **Training misclassification rate:** 0.0691
- **Test misclassification rate:** 0.0820

Although the algorithm did not converge, the inclusion of polynomial features led to a significant improvement in performance compared to the linear version. The low training error indicates a good fit to the training data, while the slightly higher test error suggests a mild degree of overfitting.

4.2 Polynomial Pegasos SVM

Cross-validation was performed to select the optimal combination of regularization parameter λ and number of iterations. The results are reported in Table 6.

λ	**Iterations**	**CV 0-1 Loss**
0.0001	5000	0.0937
0.0001	10000	0.0812
0.0001	20000	0.0757
0.0001	30000	0.0689
0.001	5000	0.1055
0.001	10000	0.0882
0.001	20000	0.0677
0.001	30000	0.0675
0.01	5000	0.0798
0.01	10000	0.0710
0.01	**20000**	**0.0595**
0.01	30000	0.0615
0.1	5000	0.0919
0.1	10000	0.0886
0.1	20000	0.0863
0.1	30000	0.0842

Table 6: Cross-validation results for the Polynomial Pegasos.

The best performance was obtained with $\lambda = 0.01$ and 20,000 iterations, corresponding to a cross-validated misclassification rate of 0.0595. Using these hyperparameters, the model was retrained on the entire training set. The final evaluation yielded the following results:

- **Training misclassification rate:** 0.0575
- **Test misclassification rate:** 0.0655

The model achieved very low error rates on both the training and test sets, indicating a good fit and strong generalization. Compared to the linear Pegasos version, the inclusion of polynomial features significantly improved performance, allowing the classifier to capture more complex decision boundaries.

4.3 Polynomial Logistic Classification

Cross-validation was performed to select the optimal combination of regularization parameter λ and number of iterations. The results are summarized in Table 7.

λ	**Iterations**	**CV 0-1 Loss**
0.0001	5000	0.0932
0.0001	10000	0.0826
0.0001	20000	0.0731
0.0001	30000	0.0666
0.001	5000	0.1046
0.001	10000	0.0812
0.001	20000	0.0694
0.001	30000	0.0661
0.01	5000	0.0843
0.01	10000	0.0736
0.01	**20000**	**0.0648**
0.01	30000	0.0673
0.1	5000	0.0950
0.1	10000	0.0932
0.1	20000	0.0917
0.1	30000	0.0917

Table 7: Cross-validation results for the Polynomial Regularized Logistic Classifier.

The best performance was obtained with $\lambda = 0.01$ and 20,000 iterations, corresponding to a cross-validated misclassification rate of 0.0648. Using these hyperparameters, the model was retrained on the entire training set. The final evaluation yielded the following results:

- **Training misclassification rate:** 0.0625
- **Test misclassification rate:** 0.0675

The model achieved low error rates on both the training and test sets. Compared to the linear version, the use of polynomial features resulted in a substantial improvement in generalization, confirming the value of feature expansion in capturing non-linear patterns in the data.

4.4 Model Comparison

Figure 4 compares the training and test misclassification rates for all six models: the linear and polynomial versions of Perceptron, Pegasos SVM, and Regularized Logistic Regression.

- **Linear models** (Perceptron, Pegasos, Logistic) show significantly higher misclassification rates compared to their polynomial counterparts. The Perceptron in particular exhibits poor performance (train: 0.373, test: 0.361), suggesting strong underfitting and inability to capture the structure of the data.
- **Polynomial models** benefit from the feature expansion, achieving much lower error rates. For instance, the Polynomial Perceptron reduces the training error to 0.069 and test error to 0.082, compared to 0.373 and 0.361 for the linear version. Similar improvements are observed for Pegasos (test error drops from 0.279 to 0.066) and Logistic (from 0.281 to 0.068).
- **Generalization** is preserved in all polynomial models. Although a slight increase from training to test error is visible, the gap remains small, suggesting good generalization and absence of significant overfitting.
- **Best performing model** in terms of test error is the Polynomial Pegasos (0.066), closely followed by the Polynomial Logistic model (0.068). Both combine low bias with strong generalization ability.

Overall, the results show that polynomial feature expansion significantly improved the performance of linear classifiers on non-linearly separable data, reducing underfitting without causing substantial overfitting.

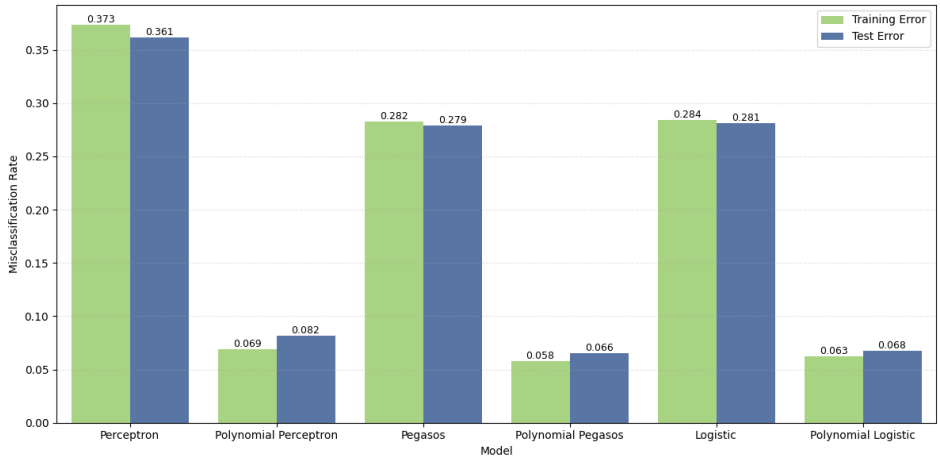

Figure 4: Training and test misclassification rates for linear and polynomial models.

4.5 Comparison of Model Weights

To better understand how different algorithms assign importance to features, we compared the learned weights across the three classifiers (Perceptron, Pegasos SVM, and Regularized Logistic Regression), both in their linear and polynomial versions.

4.5.1 Linear Models

As shown in Figure 5, the Perceptron assigns the highest weight magnitudes, particularly to features x_1, x_5, x_7, x_8, and x_9, with values ranging up to 4.25. In contrast, Pegasos and Logistic Regression assign much smaller and more balanced weights. This reflects the effect of regularization in constraining weight growth and reducing model variance.

The weight profiles of Pegasos and Logistic are similar, with consistent emphasis on features x_5, x_7, and x_8, suggesting shared identification of predictive variables. The Perceptron's large, unregularized weights indicate possible overfitting or instability in the presence of non-separable data.

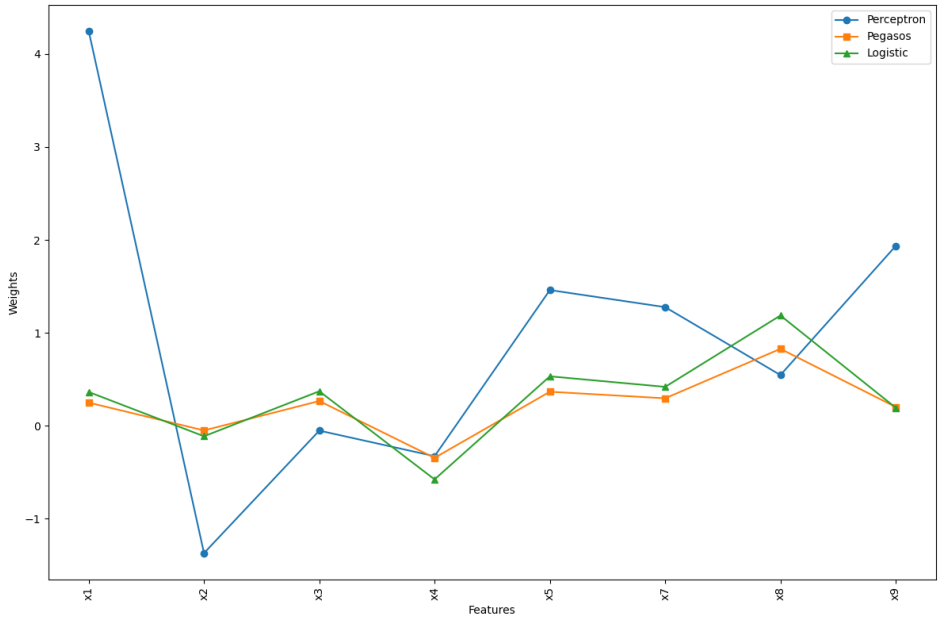

Figure 5: Weights of linear classifiers across features.

4.5.2 Polynomial Models

Table 8 reports the weights learned by the three polynomial models (Perceptron, Pegasos, and Logistic Regression) across all original features, squared terms, and pairwise interactions.

Feature	Perceptron	Pegasos	Logistic
x1	26.870877	0.584646	0.548374
x1^2	-6.247125	-0.129868	-0.137086
x1*x2	1.558961	0.232955	0.233088
x1*x3	2.136795	-0.066951	-0.092111
x1*x4	-20.285954	-0.302808	-0.224582
x1*x5	-2.458918	0.019977	-0.043769
x1*x7	-2.082160	0.068901	0.043295
x1*x8	41.719938	1.096347	0.952769
x1*x9	-3.473415	-0.022300	-0.058342
x2	4.611714	0.210879	0.162051
x2^2	3.730529	0.191110	0.201000
x2*x3	-1.808995	-0.055253	-0.119646
x2*x4	3.279047	0.069802	0.038500
x2*x5	-1.730448	-0.116031	-0.128943
x2*x7	-1.709607	-0.019357	0.044339
x2*x8	5.938727	0.067664	0.100870
x2*x9	126.708075	3.150824	2.990327
x3	10.749086	0.365915	0.406675
x3^2	-5.701623	-0.154956	-0.196409
x3*x4	-0.260327	0.092269	0.102727
x3*x5	2.932123	0.041543	0.027064
x3*x7	-1.272855	0.118999	0.088392
x3*x8	-8.083555	-0.036328	-0.073287
x3*x9	2.026977	0.057859	0.046853
x4	-16.004683	-0.721978	-0.712344
x4^2	-8.775559	-0.215478	-0.263748
x4*x5	-0.585467	0.211947	0.199419
x4*x7	-7.553442	-0.065113	-0.017050
x4*x8	-41.883500	-0.950318	-0.861967
x4*x9	0.016183	0.216226	0.150731
x5	34.365624	0.672377	0.717842
x5^2	-9.784883	-0.124863	-0.150079
x5*x7	1.112306	0.080190	0.070372
x5*x8	11.140805	0.048316	0.069708
x5*x9	-4.482009	-0.027395	0.003001
x7	23.670708	0.584116	0.585323
x7^2	-3.491770	-0.095555	-0.110351
x7*x8	5.611980	0.079674	0.034903
x7*x9	-4.202384	0.141744	0.127292
x8	69.913605	1.996533	1.882523
x8^2	-6.043211	-0.103206	-0.137165
x8*x9	-0.681050	0.133368	0.153082
x9	23.840508	0.713959	0.631568
x9^2	0.922655	0.085000	0.105991

Table 8: Weights of polynomial classifiers.

In the Perceptron, weights become extremely large in magnitude, with several coefficients exceeding 25 or dropping below -40, especially for terms like x_1x_8, x_2x_9, x_4x_8, and x_8. This again reflects the

absence of regularization, which leads to instability in weight estimation.

In contrast, both Pegasos and Logistic assign moderate and comparable weights across original and interaction terms. Notably, the highest weights are assigned to features like x_2x_9, x_8, and x_9, consistent with patterns observed in the linear case, but now extended to combinations that capture non-linear relationships.

Overall, the polynomial models show that:

- **Polynomial Perceptron** amplifies instability already seen in the linear version, with extreme coefficients that reduce interpretability.
- **Polynomial Pegasos and Logistic** maintain controlled weight magnitudes thanks to regularization, while leveraging interaction terms to model complex dependencies.
- Features such as x_2x_9, x_8, and x_1x_8 receive substantial attention, suggesting important nonlinear effects captured by the expansion.

5 Kernel Methods

While polynomial feature expansion allows linear models to capture some non-linear relationships, it still relies on explicitly generating new features. A more powerful and flexible alternative is offered by kernel methods, which enable algorithms to operate in high-dimensional feature spaces without explicitly computing the transformation. This is achieved through the *kernel trick*, which replaces the inner product in the transformed space with a kernel function evaluated directly in the input space.

Formally, given a feature mapping $\phi : \mathbb{R}^d \to \mathcal{H}$, the kernel function is defined as:

$$K(\mathbf{x}, \mathbf{x}') = \langle \phi(\mathbf{x}), \phi(\mathbf{x}') \rangle$$

This allows linear algorithms to learn complex, non-linear decision boundaries by implicitly operating in the space $\mathcal{H}$, without ever computing $\phi(\mathbf{x})$ explicitly.

In this project, kernelized versions of the Perceptron and Pegasos algorithms were implemented using two types of kernels: the polynomial kernel and the Gaussian kernel.

The **polynomial kernel** of degree n is defined as:

$$K_n(\mathbf{x}, \mathbf{x}') = (1 + \mathbf{x}^\top \mathbf{x}')^n$$

This kernel implicitly maps the data into a feature space where all monomials up to degree n are represented, allowing the model to capture complex interactions among features.

The **Gaussian kernel**, also known as the Radial Basis Function (RBF) kernel, is defined as:

$$K_\gamma(\mathbf{x}, \mathbf{x}') = \exp\left(-\frac{\|\mathbf{x} - \mathbf{x}'\|^2}{2\gamma}\right)$$

where γ is a positive parameter that controls the width of the Gaussian. The Gaussian kernel maps the input data into an infinite-dimensional space, making it highly flexible and effective at capturing localized patterns.

These kernel functions were used to extend the Perceptron and Pegasos algorithms, allowing them to capture non-linear relationships in the data. As with previous models, hyperparameter tuning was performed using 5-fold cross-validation to identify the optimal values for kernel parameters and iteration counts.

5.1 Kernelized Perceptron

The Kernelized Perceptron is a non-parametric extension of the classic Perceptron algorithm that leverages the kernel trick to handle non-linear classification tasks. Instead of maintaining a weight vector $\mathbf{w}$, the algorithm builds a set of support vectors corresponding to training examples that were misclassified during training.

At each iteration t, given a new training example $(\mathbf{x}_t, y_t)$, the prediction is computed as:

$$\hat{y}_t = \text{sgn}\left(\sum_{s \in S} y_s K(\mathbf{x}_s, \mathbf{x}_t)\right)$$

where S is the set of indices of all past support vectors, and $K(\cdot, \cdot)$ is the chosen kernel function (Gaussian or polynomial). If the prediction $\hat{y}_t$ is incorrect, the current example is added to the support set S.

This approach allows the Perceptron to learn non-linear classifiers without explicitly transforming the input space, relying instead on kernel evaluations between the current input and previously misclassified examples.

Model performance depends on two key hyperparameters: the number of training epochs and the kernel-specific parameter—σ for the Gaussian kernel and the degree n for the polynomial kernel. In fact, the number of support vectors typically grows with the number of mistakes, increasing the computational cost of both training and prediction. This makes careful tuning of the hyperparameters especially important, as it directly affects not only the model's accuracy but also its efficiency.

5.1.1 Gaussian Kernel

To evaluate the performance of the Kernelized Perceptron with Gaussian kernel, a grid search was conducted over different values of the kernel width parameter σ and the number of training epochs. The results of 5-fold cross-validation are reported in Table 9.

σ	**Epochs**	**CV 0-1 Loss**
0.1	10	0.1873
0.1	20	0.1873
0.1	50	0.1873
0.5	10	0.1429
0.5	20	0.1429
0.5	50	0.1429
1.0	10	0.0624
1.0	20	0.0582
1.0	**50**	**0.0578**

Table 9: Cross-validation results for Kernelized Perceptron with Gaussian kernel.

The best performance was obtained with $\sigma = 1.0$ and 50 training epochs, corresponding to a misclassification rate of 0.0578. Using these parameters, the model converged after 20 epochs.

Final evaluation on the full training and test sets yielded the following results:

- **Training misclassification rate:** 0.0000
- **Test misclassification rate:** 0.0665

The model achieves perfect classification on the training set and very low error on the test set, suggesting that the Gaussian kernel allows the Perceptron to effectively capture complex decision boundaries. The small gap between training and test performance indicates good generalization with no evident overfitting.

5.1.2 Polynomial Kernel

Then the Kernelized Perceptron was evaluated using the polynomial kernel, with cross-validation performed over different values of the degree n and number of training epochs $\in \{10, 20, 50\}$. The results are shown in Table 10.

The best performance was obtained with degree $n = 3$ and 50 training epochs, achieving a cross-validated misclassification rate of 0.0356. This configuration was selected for the final model evaluation.

Compared to the Gaussian kernel, the polynomial kernel also provides excellent performance, especially at degree 3, where it effectively balances complexity and generalization. The cross-validation results show a clear improvement in performance as the degree increases from 2 to 3, with diminishing returns for higher degrees.

Degree	Epochs	CV 0-1 Loss
2	10	0.0628
2	20	0.0601
2	50	0.0632
3	10	0.0564
3	20	0.0444
3	**50**	**0.0356**
4	10	0.0476
4	20	0.0464
4	50	0.0443

Table 10: Cross-validation results for Kernelized Perceptron with polynomial kernel.

The final evaluation on the training and test sets is reported below:

- **Training misclassification rate:** 0.0000
- **Test misclassification rate:** 0.0620

As with the Gaussian kernel, the model achieves perfect classification on the training set and low error on the test set. These results confirm that the polynomial kernel allows the Perceptron to construct highly expressive decision boundaries while maintaining strong generalization. Despite the strong predictive performance, the algorithm did not converge within the maximum number of epochs. This outcome is expected, as the Perceptron algorithm lacks regularization and is not guaranteed to converge when the data is not linearly separable in the kernel-induced feature space. Moreover, the computational cost is significantly higher compared to linear models. Each misclassified example is added as a support vector, and with no upper bound on their number, both training and prediction times increase rapidly. This makes the polynomial kernelized version of the Perceptron more computationally demanding, especially as the degree and number of epochs grow.

5.2 Kernelized Pegasos

The Kernelized Pegasos algorithm extends the Pegasos framework to non-linear classification tasks by incorporating the kernel trick. Instead of learning a weight vector $\mathbf{w}$ explicitly, the model maintains a vector of dual coefficients $\boldsymbol{\alpha}$, which define the decision boundary as a weighted combination of kernel evaluations over the training examples.

At each iteration $t \in \{1, \ldots, T\}$, a training index i_t is sampled uniformly at random. The algorithm updates the dual coefficient vector $\boldsymbol{\alpha}$ according to the following rule:

- For all $j \neq i_t$, set $\alpha_{t+1}[j] = \alpha_t[j]$
- If the margin condition is violated:

$$y_{i_t} \cdot \frac{1}{\lambda t} \sum_j \alpha_t[j] y_j K(\mathbf{x}_{i_t}, \mathbf{x}_j) < 1$$

  then update:

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

- Else, keep $\alpha_{t+1}[i_t] = \alpha_t[i_t]$

After T iterations, the algorithm outputs the final coefficient vector $\boldsymbol{\alpha}_{T+1}$. The prediction function is then given by:

$$f(\mathbf{x}) = \operatorname{sgn}\left(\sum_j \alpha_j y_j K(\mathbf{x}_j, \mathbf{x})\right)$$

This kernelized version enables Pegasos to learn non-linear classifiers by expressing the decision boundary entirely in terms of kernel evaluations, without explicitly computing the high-dimensional feature transformation.

As in the linear case, model performance depends critically on the choice of hyperparameters. For the Gaussian kernel, the key parameters are the regularization strength λ, the kernel width σ, and the number of iterations T. For the polynomial kernel, the parameters to tune are the regularization strength λ, the polynomial degree n, and the number of iterations T. All hyperparameters were selected through 5-fold cross-validation, balancing model complexity, accuracy, and computational cost.

5.2.1 Gaussian Kernel

The Kernelized Pegasos algorithm was evaluated using the Gaussian (RBF) kernel. A comprehensive grid search was conducted over different values of the kernel width $\sigma \in \{0.1, 0.5, 1.0\}$, the regularization parameter $\lambda \in \{0.001, 0.01, 0.1\}$, and the number of iterations $T \in \{500, 1000, 2000\}$. The results of the 5-fold cross-validation are reported in Table **??**.

σ	λ	**Iterations**	**500**	**1000**	**2000**
	0.001		0.2180	0.1925	0.1769
0.1	0.01		0.2130	0.2015	0.1796
	0.1		0.2169	0.1962	0.1837
	0.001		0.1918	0.1646	0.1425
0.5	0.01		0.1938	0.1663	0.1417
	0.1		0.1901	0.1685	0.1417
	0.001		0.1540	0.1170	**0.0995**
1.0	0.01		0.1613	0.1540	0.1525
	0.1		0.1889	0.1693	0.1662

Table 11: Cross-validation results for Kernelized Pegasos with Gaussian kernel.

The best performance was achieved with $\sigma = 1.0$, $\lambda = 0.001$, and $T = 2000$, yielding a cross-validated misclassification rate of 0.0995. Using these hyperparameters, the model was retrained on the full training set and evaluated on the test set. The final results are:

- **Training misclassification rate:** 0.0850
- **Test misclassification rate:** 0.1000

The model achieves low error rates on both training and test sets, indicating that the Gaussian kernel allows the Pegasos algorithm to learn non-linear boundaries while maintaining good generalization.

5.2.2 Polynomial Kernel

The Kernelized Pegasos algorithm was also evaluated using the polynomial kernel. A grid search was performed over the degree $n \in \{2, 3, 4\}$, the regularization parameter $\lambda \in \{0.001, 0.01, 0.1\}$, and the number of iterations $T \in \{500, 1000, 2000\}$. The results of the 5-fold cross-validation are summarized in Table 12.

The best performance was achieved with degree $n = 2$, $\lambda = 0.1$, and $T = 2000$, yielding a cross-validated misclassification rate of 0.1006. The model was retrained on the full training set using these hyperparameters and then evaluated on the test set.

- **Training misclassification rate:** 0.0923
- **Test misclassification rate:** 0.0990

These results indicate that the polynomial kernel enables the Pegasos algorithm to capture non-linear decision boundaries with high accuracy. The relatively low gap between training and test errors also confirms the effectiveness of regularization in preventing overfitting.

Degree	λ	Iterations	500	1000	2000
2	0.001		0.1573	0.1162	0.1073
	0.01		0.1461	0.1211	0.1093
	0.1		0.1336	0.1134	**0.1006**
3	0.001		0.1997	0.1693	0.1486
	0.01		0.1989	0.1576	0.1365
	0.1		0.2016	0.1650	0.1269
4	0.001		0.1923	0.1475	0.1289
	0.01		0.1815	0.1492	0.1199
	0.1		0.1937	0.1498	0.1169

Table 12: Cross-validation results for Kernelized Pegasos with polynomial kernel.

6 Conclusions

The final comparison across all implemented models, based on test accuracy, is summarized in Figure 6.

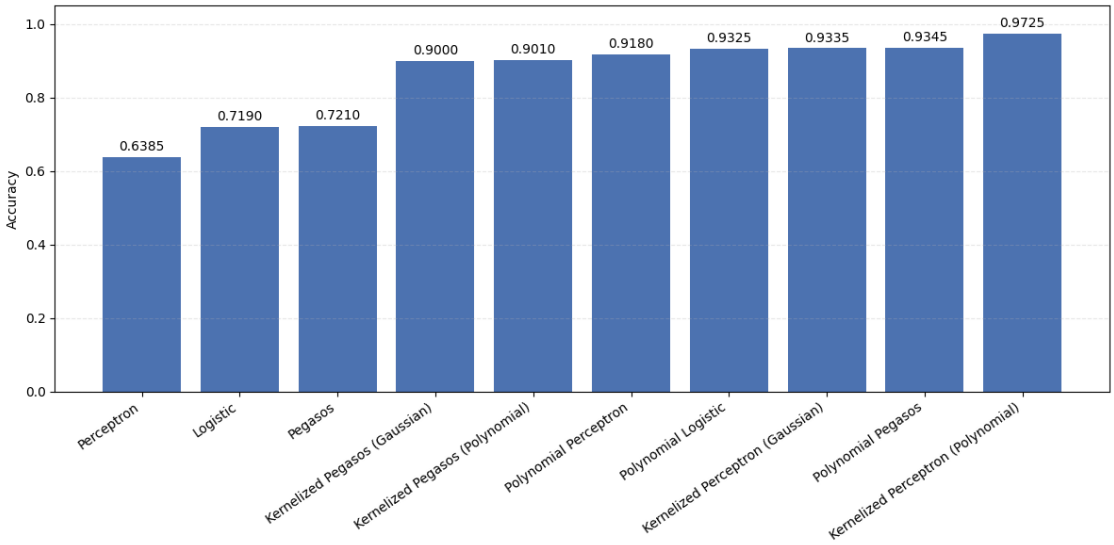

Figure 6: Test accuracy of all models.

The three baseline linear models (Perceptron, Logistic Regression, and Pegasos) yielded the lowest accuracies, with the Perceptron performing worst (Accuracy = 0.6385). Logistic and Pegasos performed similarly (≈ 0.72), with Pegasos slightly ahead, confirming the advantage of regularization in linear settings.

A substantial improvement was observed with the introduction of kernel methods. Both kernelized versions of Pegasos achieved test accuracies around 0.90 (Gaussian: 0.9000; Polynomial: 0.9010), significantly outperforming their linear counterparts. This confirms that the kernel trick, even without explicit feature expansion, enables the model to capture non-linear patterns effectively.

Polynomial feature expansion generally led to strong performance across all models. Polynomial versions of Perceptron, Logistic Regression, and Pegasos achieved accuracies above 0.9180, with Polynomial Logistic reaching 0.9325 and Polynomial Pegasos 0.9345—demonstrating the models' ability to fit complex patterns, especially when regularization is applied.

However, the highest test accuracy overall was obtained by the **Kernelized Perceptron with Polynomial kernel** (**0.9725**), which outperformed both the regularized kernelized models and those based on polynomial feature expansion. This result is particularly notable, as it shows that a non-regularized algorithm, when combined with a sufficiently expressive kernel and properly tuned, can achieve excellent generalization.

In conclusion, the experiments support the following insights:

- **Linear models** are insufficient to capture the complexity of the dataset, as shown by their relatively low accuracy.
- **Kernel methods** provide a powerful way to increase model flexibility and capture non-linear patterns without relying on manual feature engineering, substantially boosting performance.
- **Polynomial feature expansion** yields excellent results, particularly when paired with models that include regularization.
- **Regularization**, as implemented in Pegasos and logistic regression, contributes to stability and robustness, especially in high-capacity models.
- However, in this specific task, the highest performance was achieved by a *non-regularized* model—the Kernelized Perceptron with Polynomial kernel—suggesting that regularization is not always necessary when the model is sufficiently expressive and hyperparameters are carefully selected.

These findings suggest that, when computational resources allows, kernelized or polynomially-expanded classifiers represent strong choices for non-linear classification tasks.