Kernelized Linear Classification

Data Science for Economics - Machine Learning Project

Melissa Rizzi (30546A)
melissa.rizzi@studenti.unimi.it

September 2024

1 Introduction

Classification is a fundamental task that involves predicting the category or label to which a given sample belongs, based on its numerical features. The primary goal of this project is to implement and compare various supervised classification algorithms, with a focus on linear learning techniques and kernel-based methods.

The models to be implemented included:

- Perceptron
- Support Vector Machines (SVMs) using the Pegasos algorithm
- Regularized logistic classification

After the initial implementation of the basic models, performance improvements will be pursued through polynomial feature expansion and the application of kernel methods, including:

- The kernelized Perceptron with the Gaussian and the polynomial kernels
- The kernelized Pegasos with the Gaussian and the polynomial kernels for SVM

Model performance will be evaluated using zero-one loss to determine the algorithm miss-classification rate. Moreover, attention will be given to preventing data leakage between training and test sets, optimizing hyperparameters, and analyzing potential overfitting or underfitting.

2 Data Pre-Processing

Data preprocessing is a critical step in any machine learning project, as the quality of the data directly impacts the performance of the model. In this project, a dedicated class was developed to execute various preprocessing tasks to ensure the dataset's integrity and suitability for model training.

The dataset analyzed contained 10,000 samples and 10 features. To maintain high data quality, an initial examination was performed to identify any missing values, with the aim of removing rows with missing data to prevent potential biases in the model training process. Fortunately, no missing values were detected in the dataset. Secondly, to prevent outliers from negatively affecting performance, the Interquartile Range (IQR) method was applied to identify outliers across the numerical columns. These outliers are illustrated in Figure 1, highlighting the points that fall outside the acceptable range and could potentially skew the analysis. Rows exhibiting outliers in at least two

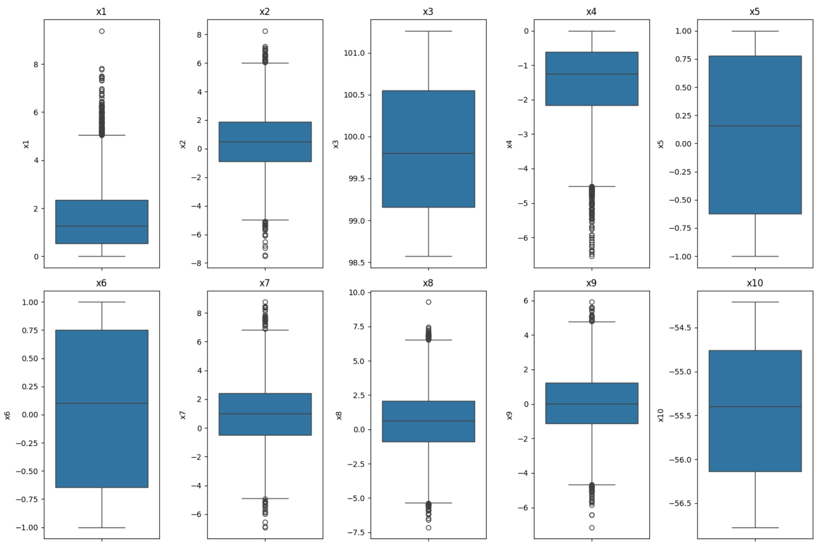

Figure 1: Boxplot

columns were classified as global outliers. A total of 49 such rows were identified and subsequently removed to enhance data quality.

Additionally, correlation analysis was carried out to uncover relationships between features. To visualize the relationships between features, a pairplot was employed, as illustrated in Figure 2.

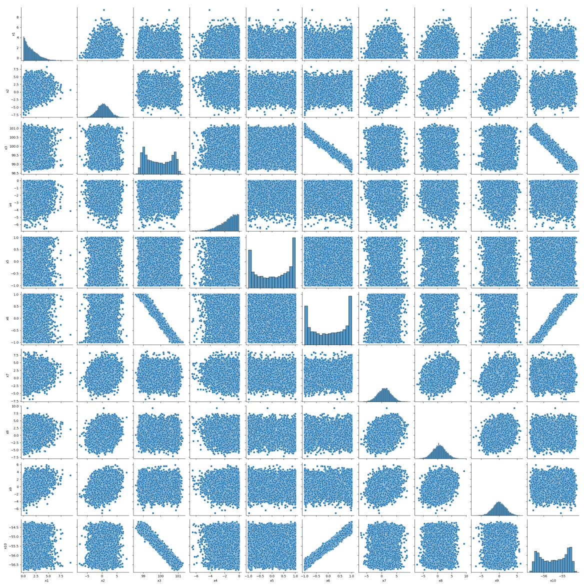

Figure 2: Scatterplot

This visualization provided scatterplots for each pair of features, allowing for an intuitive assessment of their interactions. Then, a correlation matrix was generated and presented in Figure 3. This matrix quantified the strength and direction of the relationships between features.

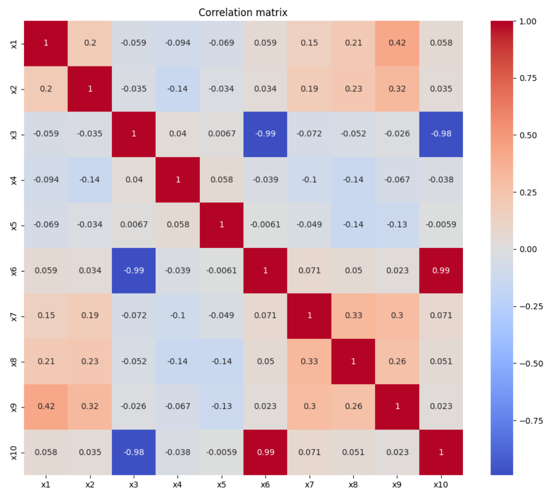

Figure 3: Correlation matrix

Features exhibiting high correlation were identified, and to mitigate the risk of multicollinearity, which can distort model performance, one of each pair of highly correlated features was removed. As shown in Figure 3, three variables demonstrated very high correlations. Consequently, two of these variables (x_6 and x_{10}) were removed from the dataset to address potential multicollinearity issues. After completing these preprocessing steps, the dataset was refined to consist of 9951 samples and 8 numerical variables. This cleaned dataset is now better suited for subsequent analysis and model training.

After refining the dataset, summary statistics were calculated for each of the remaining numerical features. The table 1 below provides key metrics, including the mean, standard deviation, and median for each variable. These statistics offer a deeper understanding of the dataset's central tendencies and variability.

	x_1	x_2	x_3	x_4	x_5	x_7	x_8	x_9
count	9951.000000	9951.000000	9951.000000	9951.000000	9951.000000	9951.000000	9951.000000	9951.000000
mean	1.578511	0.504129	99.848945	-1.492803	0.078464	0.969408	0.622340	0.043674
std	1.302123	2.038578	0.711082	1.117692	0.706878	2.149320	2.192278	1.755784
min	0.002443	-7.524934	98.572455	-6.545655	-1.000000	-6.906971	-7.140755	-7.151890
25%	0.522786	-0.886803	99.159439	-2.168608	-0.622727	-0.501048	-0.884040	-1.130132
50%	1.271157	0.484162	99.802698	-1.258343	0.157288	1.003269	0.620415	0.017506
75%	2.336024	1.889216	100.549728	-0.603027	0.778626	2.425789	2.084785	1.237907
max	9.384223	8.248503	101.260768	-0.000003	1.000000	8.760306	9.287266	5.913217

Table 1: Descriptive statistics

The means and standard deviations reveal the central tendency and spread of each feature. For example, x_3 has a mean close to 100 with a relatively low standard deviation (0.711), indicating that its values are tightly clustered around the mean. Conversely, x_2 has a higher standard deviation (2.0386), suggesting greater variability in its values. By comparing the mean and median, we can infer the asymmetry of the data. Features like x_1 and x_2 have means and medians that are relatively close, indicating that their distributions are fairly symmetric. However, x_4 has a mean of -1.4928 and a median of -1.2583, suggesting a slight asymmetry in its distribution. The distribution of variables can also be visualized graphically along the diagonal in Figure 2.

The variability in some features may affect the performance of machine learning models, so standardization was carried out on the features to ensure they were on the same scale. Finally, to avoid data leakage, the dataset was divided into training, validation, and test sets to evaluate the model's performance at different stages of development. The data was randomly shuffled and split, with 20% reserved for testing and an additional 20% of the training set used for validation.

3 Machine Learning Algorithms

In this chapter, we classify labels based on numerical features by implementing and evaluating key machine learning algorithms from scratch. The effectiveness of predictions is measured with a nonnegative loss function ℓ, which quantifies the discrepancy $\ell(y, \hat{y})$ between the predicted label $\hat{y}$ and the true label y. The simplest loss function, which will be use to evaluate model performance, is the zero-one loss:

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

It measeures the accuracy by counting the number of incorrect predictions. If the model predicts the label correctly, the loss is 0; if it predicts incorrectly, the loss is 1. The average loss over the entire test dataset gives you the percentage of misclassifications.

3.1 Perceptron

The Perceptron algorithm is a simple and foundational method for binary classification, aiming to find a linear decision boundary that separates two classes. It works by processing each training example one at a time and updating its linear classifier when it misclassifies a sample. During training, the algorithm iteratively adjusts the weights w of the linear classifier to reduce classification errors.

The perceptron update w if the predicted label $\hat{y}_t$ does not match the true label y_t. This condition can be expressed as

$$y_t \cdot (w^\top x_t) \leq 0$$

If the condition above is met, the update rule for the weight vector w is:

$$w \leftarrow w + y_t x_t$$

The predicted label $\hat{y}_t$ is computed using the sign function applied to the linear combination of weights and input features:

$$\hat{y}_t = \text{sign}(w^\top x_t)$$

In applying the Perceptron algorithm to a binary classification problem, it is important to recognize that the algorithm does not always converge, particularly when the training set is not linearly separable. This makes it crucial to carefully choose the maximum number of epochs, a hyperparameter that controls how many times the algorithm iterates over the entire training set. The optimal number of epochs is found through cross-validation.

Cross-validation involves dividing the training set into multiple subsets, training the model on a combination of these subsets, and validating it on the remaining subset. This method helps in assessing how well the model generalizes to unseen data and aids in identifying the best hyperparameters. This is typically done by minimizing a risk estimate computed using the training data. If S defines the training set and Θ_0 is a suitably chosen subset of all possible parameters, then the goal is to find $\theta^* \in \Theta_0$ such that:

$$\ell_{DA_{\theta^*}}(S) = \min_{\theta \in \Theta_0} \ell_{DA_\theta}(S)$$

By applying cross-validation, the performance of the Perceptron can be evaluated under different epoch limits and the one that yields the best results can be selected. As can be seen in Table 2 tested epochs were $T = [10, 100, 1000, 10000]$ and it was determined that the optimal performance, measured by the lowest average zero-one loss, is achieved with a maximum of 100 epochs.

Epochs	**Mean Accuracy**
10	0.6323
100	0.6417
1000	0.6281
10000	0.6373

Table 2: Cross Validation Result

It is important to note that increasing the number of epochs beyond this point does not necessarily improve the model's accuracy and may even lead to overfitting, where the model performs well on the training data but poorly on unseen data.

Once the optimal number of epochs is identified, the model is retrained on the full training set, comprising 80% of the dataset. The test set, consisting of the remaining 20%, is then used to evaluate the final model. The test error, calculated using the zero-one loss function, is 0.4319, indicating the percentage of misclassified samples in the test set.

3.2 Support Vector Machines using the Pegasos algorithm

Support Vector Machines (SVMs) are powerful classifiers known for their effectiveness in various classification tasks. Formally, an SVM seeks to identify a hyperplane that maximizes the margin between data points of different classes, thus achieving the best separation between them. This hyperplane is called the "maximally separating hyperplane," and its margin is defined as the distance between the hyperplane and the nearest data points, which are referred to as Support Vectors. The goal of an SVM is to construct this linear classifier by optimizing the margin, which directly contributes to the model's robustness and generalization capability.

To efficiently solve the optimization problem associated with SVMs, several algorithms have been developed, including the Pegasos algorithm. Pegasos, which stands for "Primal Estimated sub-GrAdient SOlver for SVM," leverages a stochastic gradient descent approach. Instead of processing the entire dataset at once, Pegasos uses mini-batches to update the model parameters, which significantly speeds up the optimization process. The optimization problem that Pegasos addresses can be written as:

$$\min_{\mathbf{w}} \frac{\lambda}{2}\|\mathbf{w}\|^2 + \frac{1}{m}\sum_{i=1}^{m} \max(0, 1 - y_i(\mathbf{w} \cdot \mathbf{x}_i))$$

where: λ is the regularization parameter and m is the number of samples in the mini-batch,

The weight update rule can be expressed as:

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

where η_t is the learning rate at iteration t and $\nabla \ell_{Z_t}(\mathbf{w}_t)$ is the gradient of the loss function ℓ with respect to the mini-batch Z_t at iteration t.

In this context, it is essential to carefully select not only the maximum number of epochs but also the values for the regularization parameter λ and the learning rate η, where η is typically a function of iteration t. To identify the optimal hyperparameters, cross-validation was employed, specifically using a 5-fold cross-validation approach. This process ensures that the selected hyperparameters generalize well across different subsets of the data, reducing the likelihood of overfitting. As shown in the following Table 3, the tested values were $T = [10, 100, 1000, 10000]$, $\lambda = [0.01, 0.1, 1]$ and $\eta_t = [eta1 = \frac{1}{0.01 \cdot t}, eta2 = \frac{1}{0.001 \cdot t}]$

Table 3 shows that the best hyperparameters are determined to be $\lambda = 0.01$, $\eta = \frac{1}{0.01 \cdot t}$, and $T = 10000$ with best accuracy 0.7198. These parameters yielded the highest accuracy on the validation sets, indicating that they provide the best balance between model complexity and training performance. When applied to the test set, the model achieved an accuracy of 0.2923.

3.3 Regularized Logistic Classification

The last machine learning technique for linear classification that will be explored is Regularized Logistic Classification. The goal is to predict the probability that an ob-

T	λ	η_t	**Mean Accuracy**
10	0.01	eta1	0.6399
		eta2	0.3217
	0.1	eta1	0.5129
		eta2	0.3812
	1	eta1	0.4466
		eta2	0.5339
100	0.01	eta1	0.6239
		eta2	0.5663
	0.1	eta1	0.6209
		eta2	0.4474
	1	eta1	0.2839
		eta2	0.3931
1000	0.01	eta1	0.7059
		eta2	0.5982
	0.1	eta1	0.6858
		eta2	0.6426
	1	eta1	0.6229
		eta2	0.2548
10000	0.01	eta1	0.7198
		eta2	0.6881
	0.1	eta1	0.7132
		eta2	0.6926
	1	eta1	0.7149
		eta2	0.6619

Table 3: Mean Accuracy of Pegasos for Different Epochs (T), Regularization Parameters (λ), and Learning Rates (η_t).

servation belongs to one of two classes. The model relies on the sigmoid function, defined as:

$$\sigma(z) = \frac{1}{1+e^{-z}}$$

which maps the linear combination of features and weights to a value between 0 and 1, interpreted as a probability. To estimate the weights w, the logistic loss function, which is used instead of hinge loss chosen for SVM algorithm, is minimized. Regularization is added to this loss function to prevent overfitting by penalizing large weights, commonly using L2 regularization, which adds a term $\frac{\lambda}{2}\|\mathbf{w}\|^2$ to the loss, where λ is the regularization parameter.

The logistic regression model was implemented and tuned through cross-validation to identify the optimal parameters. The tested values were $T = [10, 100, 1000, 10000]$, $\lambda = [0.01, 0.1, 1]$ and $\eta_t = [eta1 = \frac{1}{0.01 \cdot t}, eta2 = \frac{1}{0.001 \cdot t}]$ The result of the tuning phase are in Table 4.

After determining the best set of parameters, $\lambda = 0.1$, $\eta = \frac{1}{0.01 \cdot t}$, and $T = 10000$ with

T	λ	η_t	**Mean Accuracy**
10	0.01	eta1	0.6296
		eta2	0.5310
	0.1	eta1	0.4982
		eta2	0.5530
	1	eta1	0.5407
		eta2	0.4462
100	0.01	eta1	0.6471
		eta2	0.5835
	0.1	eta1	0.6398
		eta2	0.5088
	1	eta1	0.5716
		eta2	0.4731
1000	0.01	eta1	0.6997
		eta2	0.6820
	0.1	eta1	0.6893
		eta2	0.6296
	1	eta1	0.6611
		eta2	0.5569
10000	0.01	eta1	0.7201
		eta2	0.6918
	0.1	eta1	0.7205
		eta2	0.6851
	1	eta1	0.7106
		eta2	0.5715

Table 4: Mean Accuracy of Logistic classification for Different Epochs (T), Regularization Parameters (λ), and Learning Rates (η_t).

best accuracy 0.7205, the model was retrained on the full training dataset to ensure the most accurate fit. Subsequently, the model's performance was rigorously evaluated on the test set using the zero-one loss metric, which measures the proportion of incorrect predictions. This evaluation resulted in a misclassification error of 0.2767.

3.4 Polynomial Feature Expansion of Degree 2

To further improve the performance of these models, which include Perceptron, SVM with Pegasos, and Logistic Regression, feature expansion of degree 2 was employed. Feature expansion is a technique that enriches the original feature set by including polynomial combinations of the existing features. Specifically, a degree-2 expansion transforms the input vector X into a new feature set that includes all the original features, their squared terms, and their pairwise products. Since in the considered dataset the original features are $x_1, x_2, \ldots, x_8$, the expanded feature set will include the terms $x_1^2, x_2^2, \ldots, x_8^2$ as well as interaction terms such as $x_1 \cdot x_2, x_1 \cdot x_3, \ldots, x_7 \cdot x_8$

By expanding the feature space in this manner, the models are given the opportunity to learn higher-order interactions and patterns in the data, which can lead to improved predictive performance. In practice, this approach was applied to each model, allowing them to better fit the underlying data distributions and potentially reduce the error rates on unseen test data. Table 5 presents the zero-one loss for various algorithms, both before and after polynomial expansion.

	Basic Algorithm	**Expansion**
Perceptron	0.4319	0.0743
Pegasos	0.2923	0.0758
Logistic	0.2767	0.0784

Table 5: Results for Different Algorithms

As illustrated in Table 5, feature expansion significantly enhances classification performance across all methods by substantially reducing the zero-one loss. This improvement underscores the effectiveness of incorporating additional features to capture more complex patterns in the data. We can affirm that the Perceptron is the worst algorithm before the polynomial expansion, while yielding the highest misclassification rate. Nevertheless, all algorithms yield very similar values after the expansion, with none standing out as the absolute best.

3.5 Linear Weight Analysis

After training the models, it is essential to examine and compare the linear weights associated with both the original numerical features and the transformed features resulting from polynomial expansion. The weights in a linear model, which are shown in the following Table 6 reflect the importance and contribution of each feature to the model's predictions.

	Perceptron	Pegasos	Logistic
Feature			
x_1	-3.681947	0.261799	0.133400
x_2	-1.350421	-0.136179	0.104563
x_3	-3.122238	0.266791	0.078606
x_4	-0.408906	-0.473928	-0.202684
x_5	2.345457	0.257369	0.230063
x_7	2.182495	0.272245	0.169428
x_8	3.790075	0.810062	0.563241
x_9	-0.511020	0.334385	0.137972

Table 6: Features weights w for different algorithms

It can be observed that the weights assigned to each feature vary significantly across the algorithms, highlighting their differing approaches to optimization and decision boundaries. For example, feature x_1 is assigned a significantly negative weight by the Perceptron, while Pegasos and Logistic Regression assign it small positive weights, suggesting a different assessment of its influence. Similarly, feature x_8 consistently receives positive weights, with Logistic Regression assigning the highest value, indicating its significance across all models.

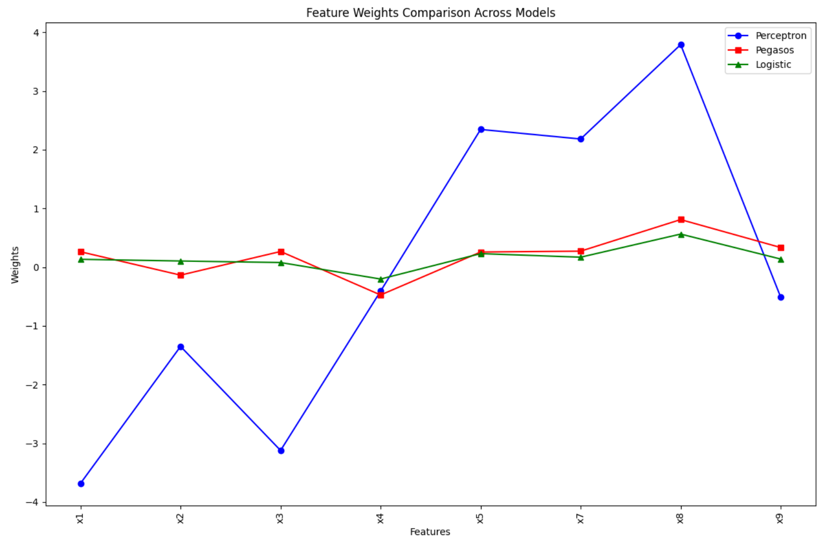

Figure 4: Weights using different algorithm

As highlighted in Figure 4, the Perceptron algorithm shows much greater variance in its weight values compared to Pegasos and Logistic Regression. This suggests that Perceptron may be more sensitive to specific features, resulting in a wider range of weight magnitudes. To facilitate a more accurate comparison between the models, it might be helpful to normalize the weights across algorithms, thereby reducing the impact of this variance and providing a clearer perspective on how each model prioritizes the features.

After understanding the importance and difference between different linear model, it's importance to study differences and confront those with the weights obtained after feature expansion. Analyzing these weights provides insights into how feature expansion has affected the model's performance and the relative significance of each feature. For the original features $x_1, x_2, \ldots, x_8$, the weights indicate their direct influence on the model's predictions. These weights reflect the contribution of each feature in a linear context. In contrast, after applying polynomial expansion of degree 2, new features such as x_i^2 and $x_i \cdot x_j$ are introduced, which capture non-linear relationships and interactions between features.

Variables	Perceptron	wPegasos	Logistic	Variables	Perceptron	Pegasos	Logistic
x_1	21.027814	0.562421	0.661518	$x_1 \cdot x_8$	1.329153	-0.042600	-0.023535
x_2	-1.528449	0.142308	0.139926	$x_2 \cdot x_3$	-7.147318	-0.111004	-0.128826
x_3	14.697149	0.447023	0.309845	$x_2 \cdot x_4$	-3.076577	0.032155	0.144567
x_4	-25.150225	-0.454053	-0.507313	$x_2 \cdot x_5$	5.159226	0.039751	-0.176694
x_5	26.861186	0.517268	0.647046	$x_2 \cdot x_6$	4.864806	-0.032498	0.123896
x_6	20.027495	0.402116	0.567052	$x_2 \cdot x_7$	6.124066	0.045407	0.166856
x_7	73.193950	1.532144	1.586769	$x_2 \cdot x_8$	131.883322	2.787110	2.684383
x_8	40.502914	0.731394	0.578630	$x_3 \cdot x_4$	1.795029	0.009443	-0.019102
x_1^2	-7.957064	-0.043113	-0.157603	$x_3 \cdot x_5$	-3.983100	0.153743	0.015290
x_2^2	2.372926	0.140919	0.074181	$x_3 \cdot x_6$	3.034796	-0.012644	0.018129
x_3^2	-1.632210	-0.152636	-0.126078	$x_3 \cdot x_7$	5.688858	0.007567	-0.003656
x_4^2	-9.052694	-0.133709	-0.105616	$x_3 \cdot x_8$	-5.539861	0.222340	-0.010479
x_5^2	-2.741817	-0.103066	-0.203495	$x_4 \cdot x_5$	6.142729	0.171032	0.130116
x_6^2	-0.688453	0.028934	-0.100139	$x_4 \cdot x_6$	-5.074252	-0.014157	0.010062
x_7^2	-2.682585	0.081556	-0.139350	$x_4 \cdot x_7$	-41.666161	-0.888688	-0.736327
x_8^2	-4.456963	0.013488	0.051721	$x_4 \cdot x_8$	-5.773459	-0.100154	-0.125723
$x_1 \cdot x_2$	5.884838	0.068899	0.159370	$x_5 \cdot x_6$	-0.865006	0.039393	0.105397
$x_1 \cdot x_3$	-1.754474	0.032347	-0.132570	$x_5 \cdot x_7$	4.058984	0.176951	0.154867
$x_1 \cdot x_4$	-19.306484	-0.239698	-0.179774	$x_5 \cdot x_8$	-4.043484	-0.112257	-0.046415
$x_1 \cdot x_5$	-2.563452	-0.066294	-0.037740	$x_6 \cdot x_7$	3.848747	0.019794	0.095182
$x_1 \cdot x_6$	5.203050	0.118026	-0.100197	$x_6 \cdot x_8$	2.072002	-0.007205	0.142795
$x_1 \cdot x_7$	56.513777	0.958018	0.931005	$x_7 \cdot x_8$	-3.430003	0.076713	0.183387

Table 7: Features weights w after polynomial expansion

Notably, the Perceptron algorithm again shows much greater variance in its weight values, particularly for interaction terms like $x_2 \cdot x_8$ and $x_7 \cdot x_8$ reflecting a heightened sensitivity to these combined features, compared to Pegasos and Logistic Classification. The expanded feature set also affects how individual features are weighted. For instance, x_7^2 and x_8^2 are assigned negative weights in Pegasos and Logistic Regression, while Perceptron assigns positive or near-zero weights. Interaction terms like $x_3 \cdot x_5$ and $x_4 \cdot x_5$ also show varied importance across the models, with Perceptron assigning higher weights to these terms compared to Pegasos and Logistic Regression.

These examples illustrate how adding more features through polynomial expansion can significantly impact the weights assigned by different algorithms. Each algorithm's treatment of these expanded features reveals its distinct approach to capturing complex relationships in the data.

4 Kernel Methods

In the preceding chapters, we focused on linear methods for classification, which work well when the data is linearly separable, meaning a linear decision boundary can effectively distinguish between different classes. However, in many practical scenarios, data may not be linearly separable in its original feature space, which can limit the performance of linear models.

To address this limitation, we introduce kernel methods, which provide a powerful way to improve the predictive performance by transforming data into higher-dimensional spaces where linear separation may become feasible. The key idea behind kernel methods is to map the input data into a new feature space where complex patterns and relationships are more easily captured. Kernel functions play a central role in this transformation process. They compute the dot product between the transformed data points in the higher-dimensional space without explicitly performing the transformation. The kernel trick helps us find an efficiently computable kernel function $K : R^d \times R^d \to R$ such that:

$$K(x, x') = \phi(x)^\top \phi(x') \text{ for all } x, x' \in R^d$$

This approach makes it computationally efficient to handle complex datasets and provides the flexibility to model intricate relationships between features. In this chapter, the application of two popular kernels will be explored: the polynomial kernel and the Gaussian kernel.

The polynomial kernel allows capturing polynomial relationships between features. It computes the dot product between data points in a polynomially expanded feature space:

$$K(x_i, x_j) = (x_i \cdot x_j + 1)^d$$

where d is the polynomial degree.

The Gaussian kernel, or Radial Basis Function (RBF) kernel, is used to measure the similarity between data points based on their distance in the input space. It is defined as:

$$K(x_i, x_j) = \exp\left(-\frac{\|x_i - x_j\|^2}{2\sigma}\right)$$

where σ is a parameter that controls the width of the kernel.

4.1 Kernelized perceptron

In the kernelized Perceptron algorithm, given a kernel K, the linear classifier generated can be expressed as:

$$h(x) = \operatorname{sgn}\left(\sum_{s \in S} y_s K(x_s, x)\right)$$

where:

- S is the set of support vectors,
- y_s are their corresponding labels,

- $K(x_s, x)$ denotes the kernel function applied to the support vectors and the input x.

This formulation highlights how the kernel function transforms the input space into a higher-dimensional space, enabling the Perceptron to learn non-linear decision boundaries.

Due to the increased computational complexity associated with kernel methods compared to the linear Perceptron, performing cross-validation to tune hyperparameters is computationally challenging. Instead of using cross-validation, a grid search was conducted on a validation set derived from an initial split of the training set. This approach reduces computational costs by training the kernelized Perceptron only once per parameter combination rather than for each fold in a cross-validation process. Although this method is less precise than cross-validation, it remains a sensible and rigorous approach for hyperparameter tuning, balancing computational efficiency with model evaluation.

First, the Gaussian kernel will be applied trying to find the optimal hyperparameter between $\sigma = [0.01, 0.1, 1]$ and $T = [10, 20]$. Despite this reduction in computational burden, the Perceptron still converged during parameter tuning, though the number of epochs required varied with different parameter settings. The results of this parameter tuning process are summarized in the following Table 8, which provides a comprehensive overview of the performance for each parameter combination evaluated.

Sigma	**Epochs**	**Converged After (Epochs)**	**Accuracy**
0.01	10	2	0.0013
0.01	20	1	0.0013
0.1	10	1	0.8536
0.1	20	1	0.8536
1	10	Did not converge	0.9529
1	20	16	0.9554

Table 8: Results of hyperparameter tuning

Using the best-found parameters ($\sigma = 1$ and $T = 20$) with an accuracy of 0.9554, the Perceptron was retrained on the entire training set. The performance on the test set was evaluated, and the algorithm converged after 14 epochs, achieving a test error of 0.0401. This outcome highlights the effectiveness of the selected hyperparameters in optimizing model performance. It also demonstrates a balance between precision and computational efficiency, as compared to conducting a full cross-validation.

Following this, we also implemented the polynomial kernel with the objective of finding the optimal degree, ranging from 2 to 3, and the ideal number of epochs, between 7 and 15. The number of epochs tested for the polynomial kernel was fewer than those for the Gaussian kernel due to the increased computational complexity associated with this method. The results from the tuning phase are presented in Table 9 below.

Degree	**Epochs**	**Accuracy**
2	7	0.8925
2	15	0.9239
3	7	0.8900
3	15	0.9353

Table 9: Results of hyperparameter tuning

After completing the tuning step, the perceptron will be retrained on the entire training set using the optimal degree of 3 and $T = 15$. In contrast to the Gaussian kernel, the polynomial kernel did not converge within the maximum number of epochs specified, but also in this case there is an improvement compared to the basic linear perceptron algorithm.

4.2 Kernelized Pegasos

Following the discussion on kernel methods and the kernelized Perceptron, this section addresses the adaptation of the Pegasos algorithm to leverage kernel functions. For simplicity, it will be adapted the basic Pegasos algorithm used before without the optional projection step.

For any iteration t, the weight vector w_{t+1} is updated according to:

$$w_{t+1} = \frac{1}{\lambda t} \sum_{i=1}^{t} \mathbf{1}\left[y_i t \langle w_t, \phi(x_i t) \rangle < 1\right] y_i t \phi(x_i t),$$

where $\mathbf{1}$ denotes the indicator function that evaluates to 1 if the condition is satisfied, and 0 otherwise.

For each t, $\alpha_{t+1} \in R^m$ is considered the vector such that $\alpha_{t+1}[j]$ counts how many times example j has been selected so far obtaining a non-zero loss on it, namely,

$$\alpha_{t+1}[j] = |\{t' \leq t : i_{t'} = j \wedge y_j \langle w_{t'}, \phi(x_j) \rangle < 1\}|$$

Instead of explicitly storing w_{t+1}, it is represented using α_{t+1} as follows:

$$w_{t+1} = \frac{1}{\lambda t} \sum_{j=1}^{m} \alpha_{t+1}[j] y_j \phi(x_j)]$$

This representation allows the Pegasos algorithm to efficiently handle kernelized data while avoiding the direct computation of high-dimensional weight vectors. This adaptation underscores the flexibility of the Pegasos algorithm in incorporating kernel methods to address more complex learning tasks.

As with the kernelized perceptron, the Gaussian kernel is evaluated first. Hyperparameter tuning is conducted using a validation set comprising 20% of the training data, with the remaining 80% used for training. The tested hyperparameters include $T = [10, 100, 1000]$, $\sigma = [0.1, 0.5, 1]$ and $\lambda = [0.01, 0.1, 1]$. The results of the tuning phase are shown in Table 10 below.

σ	λ	**T**	**Accuracy**
0.1	0.01	10	0.5590
		100	0.7098
		1000	0.8015
	0.1	10	0.5854
		100	0.7111
		1000	0.8241
	1	10	0.6790
		100	0.7305
		1000	0.8122
0.5	0.01	10	0.6526
		100	0.7569
		1000	0.8116
	0.1	10	0.6074
		100	0.7337
		1000	0.8304
	1	10	0.6778
		100	0.7431
		1000	0.8304
1	0.01	10	0.6294
		100	0.7318
		1000	0.8624
	0.1	10	0.6313
		100	0.7739
		1000	0.8844
	1	10	0.6985
		100	0.7820
		1000	0.8876

Table 10: Accuracy for Different σ, Regularization Parameters (λ), and Epochs (T).

The best-performing hyperparameters were found to be $\lambda = 1$, $\sigma = 1$ and $T = 1000$, resulting in an accuracy of 0.8876. These parameters were then applied to retrain the model using the entire training set, followed by evaluation on the test set, yielding a misclassification rate of 0.1160. This shows an improvement compared to the basic SVM model but it shows a loss higher than the one obtained with the polynomial features expansion, which was 0.0758.

The same approach was applied to the polynomial kernel, using the same values for λ and T. In this case, the additional parameter tested was the polynomial degree, with values of 2 and 3. The results of this tuning phase are provided in the Table 11 below. The final loss on the test set, with the optimal parameters (Degree = 2, $\lambda = 0.01$, and $T = 1000$), is 0.1316. While this result is better than that of the simple SVM model, it is not as effective as the polynomial feature expansion method, which still proves to be superior.

Degree	λ	**T**	**Accuracy**
2	0.01	10	0.5515
		100	0.7977
		1000	0.8926
	0.1	10	0.6369
		100	0.7569
		1000	0.8781
	1	10	0.6055
		100	0.7242
		1000	0.8788
3	0.01	10	0.6156
		100	0.7531
		1000	0.7569
	0.1	10	0.5440
		100	0.7111
		1000	0.8241
	1	10	0.5923
		100	0.5641
		1000	0.8304

Table 11: Accuracy for Different Degrees, Regularization Parameters (λ), and Epochs

5 Conclusion

In this project, various linear and kernelized classification algorithms were implemented and evaluated to classify data based on numerical features. The core models implemented included the Perceptron, Pegasos, and regularized logistic regression, each tested in both its basic linear form and with the addition of polynomial and Gaussian kernels.

The results indicated that while linear models offer a straightforward approach to classification, they are often constrained by the assumption of linear separability in the data. Polynomial feature expansion was applied to address this limitation, enriching the feature space by introducing higher-order interactions. This expansion significantly improved the performance of all models, reducing the zero-one loss and allowing the models to capture more complex patterns within the data.

The kernel methods, particularly the Gaussian and polynomial kernels, further enhanced model performance by mapping the data into higher-dimensional spaces where non-linear relationships could be more effectively modeled. However, the increased computational complexity associated with kernel methods required a balanced approach to hyperparameter tuning. Instead of traditional cross-validation, a more computationally efficient grid search was employed, yielding results that, while less precise, still provided valuable insights into the optimal settings for each model.

In summary, this project highlighted the effectiveness of both polynomial feature expansion and kernel methods in improving classification accuracy.Future work could focus on exploring additional kernel functions, further optimizing hyperparameters, or incorporating advanced feature engineering techniques to push the boundaries of model performance even further.