Kernelized Linear Classification

Giovanni Buscemi

Università degli studi di Milano

1 Introduction

The objective of this project is to implement and evaluate various classification models in a supervised learning context. Each model consists of a **Predictor** $f : X \to Y$ that maps data points into the label space, aiming to minimize a loss function, which is strongly related to the concept of **Empirical Risk Minimization (ERM)**, a fundamental principle in machine learning, which seeks to minimize the empirical risk(i.e., the average loss function calculated on the training data).
The models considered in this project include the **Perceptron**, **SVM** trained with the **Pegasos** algorithm, and **regularized logistic classification**. Additionally, both the Perceptron and SVM have been extended using the **Kernel Trick**, a technique that allows data to be mapped into a nonlinear feature space, enhancing the performance of models in cases where data are not linearly separable. However, the use of kernels introduces a higher computational burden, which has been managed by implementing the kernel matrix computation through vectorized operations optimized with **NumPy**.

Overall, this project explores the effectiveness of linear and nonlinear models by finding best hyperparameter with Grid-Search and K-fold validation, training and evaluating their performance in terms of accuracy, generalization capability, and computational costs. The results show high performance from the Kernel Perceptron, particularly with the polynomial kernel, and a low generalization capability from the Logistic Classifier. The low generalization capability of the Regularized Logistic Classifier may be due to the fact that a probability-based model is not always well-suited to capture complex class relationships, especially in contexts where distances between data points are crucial for classification.

2 Dataset and Preprocessing

The dataset used in this project contains 10,000 samples with 10 numerical features and two balanced classes. Before training the models, a check was performed to verify the presence of missing values. The data was then split into training and test sets using a stratified split to preserve the original class distribution in both sets. This approach reduces the risk of data leakage and allows for an accurate evaluation of the models' generalization capability on unseen data.

After splitting the data, the features were standardized—setting a mean of zero and a unit standard deviation for each feature. This standardization ensures

that all features contribute equally during training, preventing features with larger scales from dominating the learning process.

3 Implemented model

All the models implemented in this project are linear predictors, meaning they compute a decision function as a linear combination of the features. The decision function can be expressed as:

$$h(\mathbf{x}) = \mathbf{w}^\top \mathbf{x} + b$$

where $\mathbf{w}$ is the weight vector, $\mathbf{x}$ is the feature vector, and b is the bias term. The objective of each model is to find the optimal parameters $\mathbf{w}$ and b by minimizing a specific loss function. Despite the linear nature of these models, they differ in the loss functions they minimize and the optimization techniques they employ.

A linear predictor defines a hyperplane in the feature space, given by:

$$\mathbf{w}^\top \mathbf{x} = c$$

where c is a constant. This hyperplane divides the space into two half-spaces:

$$H^+ = \{\mathbf{x} : \mathbf{w}^\top \mathbf{x} > c\} \quad \textit{and} \quad H^- = \{\mathbf{x} : \mathbf{w}^\top \mathbf{x} \leq c\}$$

A linear classifier then assigns a label $+1$ to points in H^+ and a label -1 to points in H^-. The goal is to find the hyperplane that best separates the classes, minimizing classification errors according to the chosen loss function.

Training a linear classifier involves finding the optimal weight vector $\mathbf{w}$ such that the decision boundary defined by the hyperplane accurately separates the classes.

3.1 Perceptron

The Perceptron is one of the simplest and most fundamental algorithms in machine learning for classification tasks. It is a linear model that seeks to find a hyperplane that separates two classes of data. The algorithm iteratively explores the data and updates the weights of the hyperplane based on the classification errors encountered during the learning process. In each iteration, the Perceptron aims to minimize the number of errors by adjusting the hyperplane in the correct direction until an optimal linear separation is found (if the data are linearly separable). The Perceptron is an example of online learning, as the model is updated incrementally as new data points are observed.

Algorithm: Perceptron

Hyperparameter : Epoch, η

Initialize the weight vector $\mathbf{w} = \mathbf{0}$ and bias $b = 0$.

For each epoch:

– For each training example $(\mathbf{x}_i, y_i)$:
  • If $y_i(\mathbf{w}^\top \mathbf{x}_i + b) \leq 0$:
    * Update the weights: $\mathbf{w} \leftarrow \mathbf{w} + \eta(y_i \mathbf{x}_i)$
    * Update the bias: $b \leftarrow b + \eta y_i$

Repeat until convergence or for a fixed number of epochs.

The Perceptron seeks a hyperplane that correctly separates all observations in the dataset. However, if the data are not linearly separable, the algorithm will never converge and will continue updating the weights indefinitely. This limitation restricts the Perceptron to linearly separable problems, but its simplicity and speed make it an important starting point for supervised learning.

3.2 Pegasos SVM

The Pegasos (Primal Estimated sub-GrAdient SOlver for SVM) algorithm is an optimization technique designed to solve the Support Vector Machine (SVM) problem efficiently. The goal of an SVM is to find the hyperplane that best separates data into two classes, maximizing the margin between them. The margin is defined as the distance between the hyperplane and the nearest data points, known as "support vectors." Pegasos is particularly noted for its ability to train SVMs efficiently using stochastic gradient descent (SGD), which processes training examples one at a time or in small batches, and iteratively updates the model's weight vector.

The optimization problem in SVMs is to minimize a specific objective function, which combines the hinge loss with a regularization term. The objective function can be expressed as:

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

where:

– $\mathbf{w}$ is the weight vector.
– λ is the regularization parameter.
– m is the number of training examples.
– y_i is the class label for example i.
– $\mathbf{x}_i$ is the feature vector for example i.

The first term $\lambda \frac{1}{2}\|\mathbf{w}\|^2$ is the regularization term, which helps prevent overfitting by penalizing large weights. The second term represents the hinge loss, which penalizes misclassified examples and those that fall within the margin.

Stochastic Gradient Descent (SGD) is a method where the model's parameters are updated incrementally based on a subset of training examples (a minibatch) rather than the entire dataset. This approach significantly reduces computational costs, making the algorithm scalable to large datasets, so, instead of processing one example at a time, the Pegasos algorithm can be adapted to

use mini-batches of data in each iteration. This variant is known as Mini-Batch Pegasos, and it works as follows:

Algorithm: Mini-Batch Pegasos

1. Initialize the weight vector $\mathbf{w} = \mathbf{0}$.
2. For each iteration $t = 1, 2, \ldots, T$:
   - Randomly select a minibatch $A_t \subseteq \{1, \ldots, m\}$ of size k.
   - Identify the set $A_t^+ = \{i \in A_t : y_i \mathbf{w}^\top \mathbf{x}_i < 1\}$.
   - Update the learning rate: $\eta_t = \frac{1}{\lambda t}$.
   - Update the weight vector:

$$\mathbf{w} \leftarrow (1 - \eta_t \lambda)\mathbf{w} + \frac{\eta_t}{k} \sum_{i \in A_t^+} y_i \mathbf{x}_i$$

   - Optionally, project $\mathbf{w}$ onto the $\frac{1}{\sqrt{\lambda}}$-ball to ensure boundedness.
3. Output the final weight vector $\mathbf{w}_T$.

The choice of the mini-batch size k significantly influences the behavior of the Pegasos algorithm. If $k = 1$, the algorithm operates in a fully stochastic mode, updating the model parameters based on a single training example at a time. Conversely, if $k = m$, where m is the total number of training examples, the algorithm performs a full batch update, using the entire dataset to compute the gradient. The mini-batch approach strikes a balance between these two extremes, providing a trade-off between computational efficiency and convergence stability.

3.3 Regularized Logistic Classification

Regularized Logistic Classification is a linear model widely used for binary classification tasks. It estimates the probability that a given sample belongs to a particular class using the logistic (sigmoid) function :

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

The model seeks to find an optimal weight vector $\mathbf{w}$ that minimizes the classification error while preventing overfitting through the use of regularization term.

Optimization Objective The optimization problem in Regularized Logistic Classification is defined as the minimization of the logistic loss (also known as cross-entropy loss) combined with a regularization term. The objective function can be expressed as:

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

where:

– $\mathbf{w}$ is the weight vector.
– λ is the regularization parameter.
– m is the number of training examples.
– y_i is the true label for the i-th training example.
– $\mathbf{x}_i$ is the feature vector for the i-th training example.

The first term $\lambda \frac{1}{2}\|\mathbf{w}\|^2$ is the regularization term, which penalizes large weights to prevent overfitting. The second term is the logistic loss, which measures the discrepancy between the predicted probability and the actual label.

To efficiently minimize the objective function, Regularized Logistic Classification often uses Stochastic Gradient Descent (SGD), in the same way of the Pegasos SVM Algorithm. Here the pseudocode :

Algorithm: Mini-Batch Regularized Logistic Classification

1. Initialize the weight vector $\mathbf{w} = \mathbf{0}$.
2. For each iteration $t = 1, 2, \ldots, T$:
   – Randomly select a mini-batch $A_t \subseteq \{1, \ldots, m\}$ of size k.
   – Compute the gradient for the mini-batch:

$$\nabla \mathbf{w} = \lambda \mathbf{w} - \frac{1}{k} \sum_{i \in A_t} y_i \mathbf{x}_i \left(1 - \frac{1}{1 + e^{-y_i \mathbf{w}^\top \mathbf{x}_i}}\right)$$

   – Update the learning rate: $\eta_t = \frac{1}{\lambda t}$.
   – Update the weight vector:

$$\mathbf{w} \leftarrow \mathbf{w} - \eta_t \nabla \mathbf{w}$$

3. Output the final weight vector $\mathbf{w}_T$.

Mini-Batch Regularized Logistic Classification is effective in balancing the trade-offs between convergence speed.

3.4 Kernel approach

The kernel approach is a powerful technique that allows linear learning algorithms to learn non-linear decision functions. This is achieved by implicitly mapping the input data into a high-dimensional feature space through the **kernel trick** which allow to operate in higher-dimensional space through the computation of inner products in the high-dimensional feature space without explicitly mapping the data into that space. Formally, given a mapping $\phi : R^n \to R^m$ (where typically $m \gg n$), the kernel function K satisfies:

$$K(\mathbf{x}_i, \mathbf{x}_j) = \phi(\mathbf{x}_i)^\top \phi(\mathbf{x}_j)$$

Here, $\phi(\mathbf{x})$ is the mapping to the high-dimensional space, and $K(\mathbf{x}_i, \mathbf{x}_j)$ is the kernel function that computes the inner product in the feature space without needing to compute $\phi(\mathbf{x})$ explicitly. Kernel function used in this project :

– **Polynomial Kernel**: $K(\mathbf{x}_i, \mathbf{x}_j) = (\mathbf{x}_i^\top \mathbf{x}_j + c)^d$
– **Gaussian (RBF) Kernel**: $K(\mathbf{x}_i, \mathbf{x}_j) = \exp\left(-\frac{\|\mathbf{x}_i - \mathbf{x}_j\|^2}{2\sigma^2}\right)$

When using a kernel function, the learning algorithm is applied in the kernel space, where the decision boundary is linear, although it corresponds to a non-linear boundary in the original feature space. The decision function for a classifier in the kernel space can be written as:

$$f(\mathbf{x}) = sign\left(\sum_{i=1}^{m} \alpha_i y_i K(\mathbf{x}_i, \mathbf{x}) + b\right)$$

where:

– α_i are the dual coefficients corresponding to the support vectors $\mathbf{x}_i$.
– y_i are the class labels.
– $K(\mathbf{x}_i, \mathbf{x})$ is the kernel function evaluating the similarity between $\mathbf{x}_i$ and the test point $\mathbf{x}$.
– b is the bias term.

Kernel Matrix Computation : To efficiently implement kernel-based algorithms, the **kernel matrix** (also known as the Gram matrix) is computed. The kernel matrix $\mathbf{K}$ is an $m \times m$ matrix where each entry is defined as:

$$\mathbf{K}_{ij} = K(\mathbf{x}_i, \mathbf{x}_j)$$

This matrix encapsulates all pairwise similarities between training examples, allowing the algorithm to operate in the kernel space. Using the kernel matrix, the Perceptron and Pegasos SVM algorithms can be adapted to work in the kernel space.

Kernelized Perceptron Algorithm The Perceptron algorithm can be adapted to the kernel space by replacing the inner product with the kernel function. The algorithm is as follows:

Algorithm: Kernelized Perceptron

1. Initialize the coefficient vector $\alpha = \mathbf{0}$
2. For each epoch:
   – For each training example $(\mathbf{x}_i, y_i)$:
     • Compute the decision function: $f(\mathbf{x}_i) = \sum_{j=1}^{m} \alpha_j y_j K(\mathbf{x}_j, \mathbf{x}_i) + b$
     • If $y_i f(\mathbf{x}_i) \leq 0$:
       ∗ Update the coefficients: $\alpha_i \leftarrow \alpha_i + 1$
3. Output the final coefficients α.

Kernelized Pegasos SVM Algorithm Similarly, the Pegasos SVM algorithm can be kernelized by incorporating the kernel function in place of the inner product. The kernelized Pegasos SVM algorithm is as follows:

Algorithm: Kernelized Pegasos SVM

1. Initialize the coefficient vector $\alpha_1 = \mathbf{0}$
2. For each iteration $t = 1, 2, \ldots, T$:
   - Randomly choose $i_t \in \{0, \ldots, |S|\}$.
   - For all $j \neq i_t$, update the coefficients:

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

   - If $y_{i_t} \frac{1}{\lambda t} \sum_j \alpha_t[j] y_j K(x_{i_t}, x_j) < 1$, then:

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

   - Else:

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

3. Output the final coefficients α.

Computational Considerations Computing the kernel matrix can be computationally intensive, especially for large datasets, as it requires $O(m^2)$ operations, where m is the number of training examples. To accelerate these calculations, efficient vectorized operations using `NumPy` are employed. Once the kernel matrix is computed, significantly reduce the overall computational cost.

The kernel approach offers a significant advantage by allowing linear algorithms to learn complex, non-linear decision boundaries. However, it also introduces higher computational costs and memory requirements, particularly when using non-linear kernels like the Gaussian RBF.

4 Methodology and experimental result

For each model, a 5-fold Cross-Validation (K = 5) was employed during the Grid Search to evaluate different hyperparameter configurations. This method divides the dataset into five subsets, using four subsets for training and one for validation, rotating this process across all subsets. The performance of each hyperparameter configuration was averaged over the folds to select the optimal parameters.

During the hyperparameter tuning phase, various values for the regularization parameter λ, learning rate η, and kernel parameters were explored. The models' performance was assessed using standard accuracy, according to 0-1 loss.

4.1 Perceptron Results

For the Perceptron model, the optimal hyperparameters were found to be a learning rate of 0.01 and 50 epochs. Under these conditions, the Perceptron achieved a training accuracy of 0.680 and a test accuracy of 0.682. When applying polynomial feature expansion with a degree of 2, the model showed a significant performance improvement. The training accuracy increased to 0.9375, while the test accuracy reached 0.9345.

4.2 Pegasos SVM Results

For the Pegasos SVM model, the best-performing hyperparameters were found to be a regularization parameter λ of 1, 200 epochs, and a batch size of 32. With these settings, the model achieved a training accuracy of 0.728 and a test accuracy of 0.7285.

Applying polynomial feature expansion of degree 2 resulted in a slight improvement in performance. The training accuracy increased to 0.755, and the test accuracy improved to 0.754.

4.3 Regularized Logistic Classification Results

For the Regularized Logistic Classification model, the best hyperparameters included a regularization parameter λ of 0.01, 50 epochs, and a batch size of 32. The best cross-validated score achieved was 0.571875. The training accuracy at the final epoch was 0.54525, while the test accuracy was 0.5485.

When polynomial feature expansion was applied, the model's performance improved. The training accuracy increased to 0.603875, and the test accuracy improved to 0.607, indicating that polynomial features contributed to better model performance.

4.4 Kernelized Results

For the Kernelized Perceptron with a polynomial kernel, the optimal hyperparameters were 100 epochs, a polynomial degree of 3, and a coef0 value of 1. Under these settings, the model achieved a training accuracy of 0.97 and a test accuracy of 0.96, showcasing the effectiveness of the polynomial kernel in enhancing the model's performance.

For the Gaussian Kernel, the best hyperparameters were found to be 200 epochs and a gamma value of 10. With these settings, the model attained a training accuracy of 0.987 and a test accuracy of 0.9665.

Regarding the Kernelized Pegasos SVM with polynomial kernel, the optimal hyperparameters were a regularization parameter λ of 0.1, 400 epochs, a polynomial kernel with a degree of 2, and a coef0 value of 1.0. This configuration yielded a test accuracy of 0.8275, indicating a significant performance boost with the polynomial kernel.

For the Kernelized Pegasos SVM with Gaussian kernel, the optimal hyperparameters included a regularization parameter λ of 0.001, 400 epochs, and a gamma value of 1. The best cross-validated score was 0.814, and the test accuracy was 0.8045.

4.5 Discussion of Results

Model/Configuration	Training Accuracy	Test Acc
Perceptron	0.680	0.68
Perceptron with Polynomial Feature Expansion (Degree 2)	0.9375	0.934
Pegasos SVM	0.728	0.728
Pegasos SVM with Polynomial Feature Expansion (Degree 2)	0.755	0.75
Regularized Logistic Classification	0.54525	0.548
Regularized Logistic Classification with Polynomial Feature Expansion	0.603875	0.60
Kernelized Perceptron (Polynomial Kernel)	0.97	0.96
Kernelized Perceptron (Gaussian Kernel)	0.987	0.966
Kernelized Pegasos SVM (Polynomial Kernel)	-	0.827
Kernelized Pegasos SVM (Gaussian Kernel)	-	0.804

Table 1. Summary of accuracy results for various models and configurations.

The experimental results demonstrate that the use of the Kernel Trick and polynomial feature expansion significantly improved the models' performance on non-linearly separable datasets. However, these improvements came at the cost of increased computational time. In the Perceptron algorithm, each update require O(d), where d is the number of feature of the data point. In the Kernelized Perceptron, each update require O(N d), where N is the number of feature, because every update the algorithm calculate the distance between the other points. The Grid Search and Cross-Validation process helped in fine-tuning the hyperparameters, ensuring a good balance between accuracy and computational efficiency.

For all models where training accuracy is available, the training accuracy closely aligns with the test accuracy, indicating a lack of overfitting. This suggests that the models generalize well to unseen data.

On the other hand, underfitting is observed in the Regularized Logistic Classification model, which demonstrates poor generalization on the data. In the context of binary classification, an accuracy slightly above 50 percent implies a performance equivalent to random class selection, which is unacceptable. This underperformance indicates that the model fails to capture the underlying patterns in the data, possibly due to an overly simplistic model structure or inadequate feature representation. This lack of sufficient model complexity or feature richness prevents the Regularized Logistic Classification from effectively distinguishing between classes, resulting in its inability to achieve a meaningful predictive performance.