UNIVERSITÀ
DEGLI STUDI
DI MILANO

Kernelized Linear Classification

Ali Tabaraei

Statistical Methods for Machine Learning
Under the supervision of *Prof. Nicolò Cesa-Bianchi*
Department of Computer Science
University of Milan

a.a. 2023-2024

Kernelized Linear Classification

Ali Tabaraei

Abstract

In this project [3], we explored the classification of labels based on numerical features using various machine learning algorithms implemented from scratch, including the Perceptron, Pegasos SVM, and regularized logistic classification. The dataset, comprising 10 numerical features and 10,000 samples, was loaded and processed using Python's Pandas and Numpy libraries. Initial data exploration confirmed the absence of missing values and duplicates, and the distribution of both features and target labels was analyzed to inform preprocessing strategies.

Key preprocessing steps included feature scaling through Z-score normalization and outlier removal using the Interquartile Range (IQR) method. Feature selection was performed based on correlation analysis, leading to the exclusion of highly correlated features to prevent redundancy and improve model performance.

Using K-fold Nested Cross-Validation as our hyper-parameter tuning method, the models were evaluated along the dataset confirming a sound procedure and ensuring an unbiased evaluation of the models. We evaluated the models' performance using accuracy, precision, and recall as metrics, and also analyzed their computational efficiency in terms of runtime. Throughout the analysis, careful attention was paid and overfitting was avoided, while mild underfitting persisted in some baseline models.

The study revealed that baseline models exhibited limited performance on non-linearly separable data due to their linear nature. However, both feature-expanded and kernelied implementations of baseline models resulted in significant performance improvements, with the Polynomial Kernel Perceptron emerging as the top performer.

Keywords: Perceptron, Pegasos for SVM, Feature-Expansion, Kernels

University of Milan
August 2024

Declaration

Signature: Ali Tabaraei
Date: August 2024

Contents

List of Figures

List of Tables

Chapter 1

Data Loading and Exploration

Throughout this first section, our main focus is to:

1. Effectively load and store the data in a `Pandas` DataFrame for subsequent processing
2. Analyze the distribution of the target labels and features to design the preprocessing strategy to be followed accordingly

1.1 Loading the Dataset and Libraries

Throughout the project, `Pandas` and `Numpy` libraries are utilized for data wrangling and numerical functionalities on data, `Matplotlib` and `Seaborn` are jointly used for data visualization, and `tqdm` was employed to display a progress bar during each training process. However, due to rise of performance issues, we disregarded the use of `tqdm` in the project later.

Moreover, we attempted to load the dataset directly from the provided Google Drive link [1] associated with the project. The dataset comprises 10 features, denoted as $x_1, \ldots, x_{10}$, and contains `10,000` samples in total. Using the `info()` function from `Pandas`, it was confirmed that there are no missing values, as all features have a complete set of `10,000` samples.

Upon analyzing the dataset, we observed the following:

- The feature matrix X consists exclusively of numerical values, eliminating the need for any conversion from categorical to numerical data. All features are stored as `float64`, so no data type conversion is required.
- The target labels y are binary, taking values from the set $\{-1, 1\}$. These labels are stored as `int64`, which aligns well with our classification objectives.

1.2 Data Distribution Exploration

Analyzing the distribution of target labels y using the `value_counts()` function on the DataFrame reveals that the dataset is almost perfectly *balanced*, with roughly the same number of positive and negative classes distributed among the samples of the dataset (`5,008` negative and `4,992` positive). Figure 1.1 illustrates this distribution.

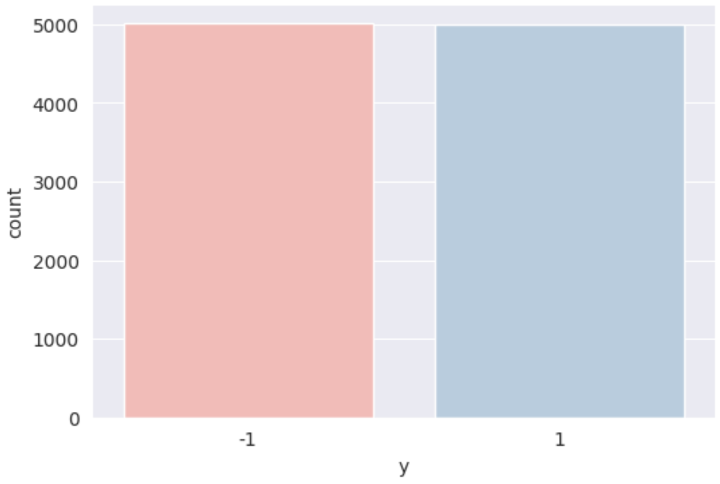


Figure 1.1: Distribution of Target Labels

On the other hand, analyzing the distribution of features $x_i \in \mathcal{X}$ using the `describe()` function, it can be seen that the range of values each feature takes is different than the other. Consequently, we need to apply appropriate scaling methods during preprocessing to ensure that features with larger value ranges do not dominate the influence on our predictions. These different distributions are visualized in Figure 1.2.

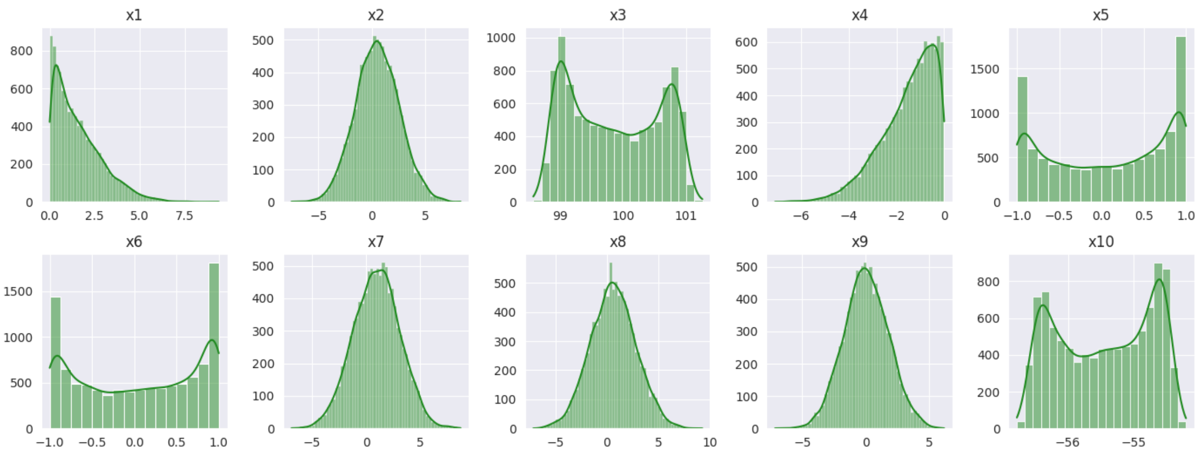


Figure 1.2: Distribution of Features

Additionally, we will assess the potential presence of noise and outliers. While techniques such as the *z-score* are available for outlier detection, we will first examine the boxplot for each feature. Subsequently, we will use the Interquartile Range (IQR) method to establish outlier boundaries and remove them, as it is widely recognized and reliable in the field.

The IQR measures the spread of the central 50% of a dataset by calculating the difference between the third quartile (Q3) and the first quartile (Q1), given by:

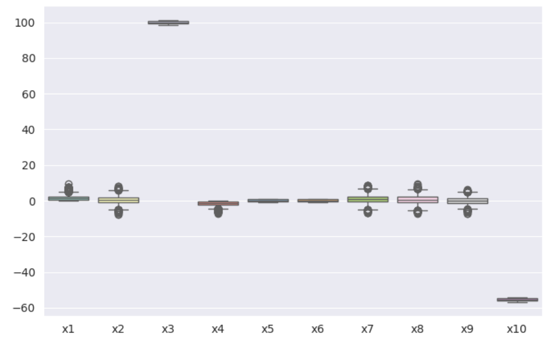
$$\text{IQR} = Q3 - Q1 \tag{1.1}$$

To identify outliers, we compute the lower and upper bounds as follows, where the data points outside these bounds are considered potential outliers:

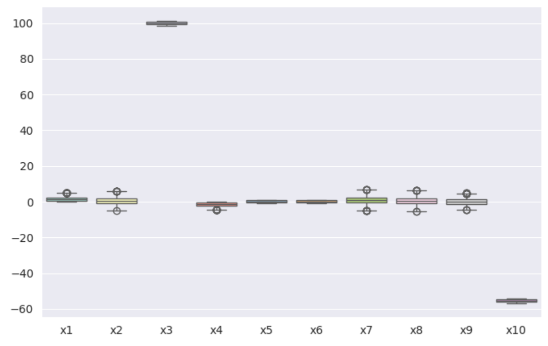
$$\text{Lower Bound} = Q1 - 1.5 \times \text{IQR} \tag{1.2}$$

$$\text{Upper Bound} = Q3 + 1.5 \times \text{IQR} \tag{1.3}$$

The IQR method is effective in identifying the impact of outliers, thereby improving the accuracy of our dataset analysis. As it is illustrated in Figure 1.3, the majority of outliers have been addressed using the IQR technique, allowing us to proceed to the next step. Following outlier removal, the dataset now includes `9,456` samples, indicating that approximately `550` samples were removed.



(a) Before outlier removal



(b) After outlier removal

Figure 1.3: Box-plots of different features and the effect of outlier removal

This method, however, may not eliminate all data points that are distant from the majority of the data. Some points, although not classified as extreme outliers, may still be noticeably separated from the main cluster of data and can appear as mild outliers in the boxplot.

Chapter 2

Data Preprocessing

In this section, our primary goal is to preprocess the data effectively to ensure it is ready for machine learning algorithms and free from errors. By the end of this exploration and analysis, the dataset should be clean and well-prepared for model training. We will address the following key criteria:

1. **Presence of Missing Values:** The dataset is checked for presence of any missing values. Null values can complicate numerical computations and degrade model performance. To address this, we will either impute missing values or remove the affected rows/columns to maintain data integrity.

2. **Check for Duplicates:** Duplicate samples with identical characteristics can skew the results and reduce model performance. We will identify and remove any duplicate entries to ensure each sample in the dataset is unique.

3. **Feature Selection:** Selecting the most informative features is undeniably one of the most important tasks before feeding the data into machine learning models. In this particular dataset, since we do not have prior knowledge about the information provided by features, we will only rely on the identification of correlation among features, to filter our highly correlated features.

4. **Feature Normalization:** In the data exploration step, we discovered that the features in our dataset vary in range, which can negatively impact the performance of certain algorithms such as SVM. Therefore, we need to adopt normalization techniques to prevent side effects, ensuring they contribute equally to the model's learning process.

5. **Partitioning the Dataset:** To prevent data leakage and ensure a fair evaluation of our models, we will split the dataset into training and test sets. This separation ensures that the test set remains unseen during training, providing an unbiased assessment of model performance and ensuring that we follow a sound methodology.

2.1 Presence of Missing Values

As we mentioned earlier in the previous section, there is no Null values present in our dataset, hence no further consideration is required in this regard.

However, in order to confirm this and analyze the existence of Null values, we will count the occurrences of such values using `isnull()` function in each column. It was confirmed that all the columns were free of missing values.

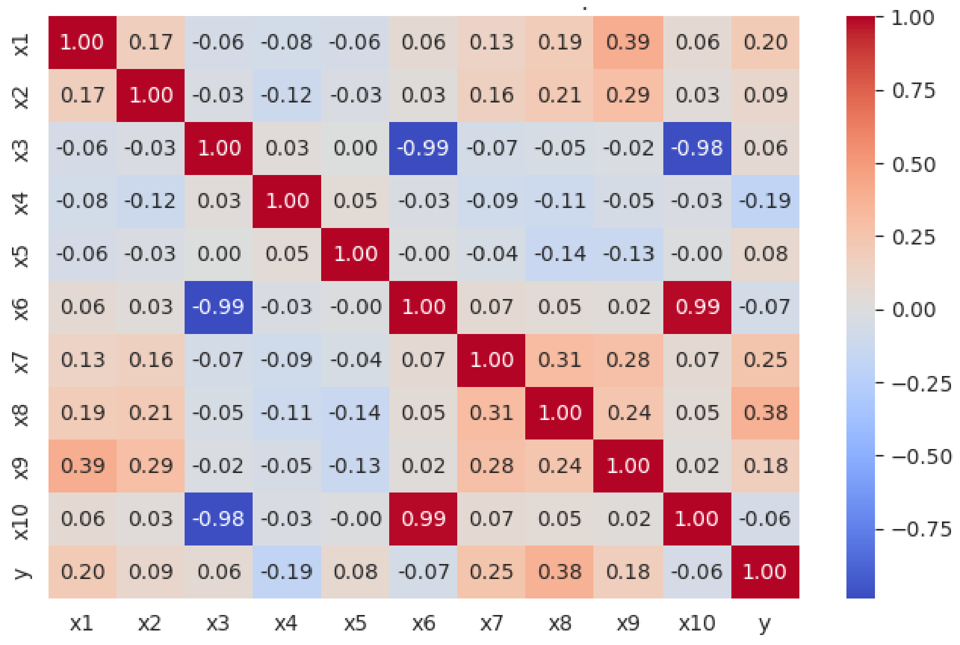
2.2 Check for Duplicates

Using the `duplicated` function provided by Pandas, we can count the duplicated rows. As it turns out, there are no duplicated rows present in our dataset, therefore no actions are expected in this manner.

2.3 Feature Selection

It is evident that many of the machine learning applications rely on the fact the features are *independent* from each other, hence they are not correlated with one another. Highly correlated features can sometimes provide redundant information, where we might consider dimensionality reduction techniques or feature selection methods in such cases.

The identification of the correlation among features is an important task, which can be achieved by using the *correlation matrix* to assess the correlation among different features and identify potentially irrelevant information. A correlation matrix provides the correlation coefficients between pairs of features, which helps to understand the relationships between them. After computing this matrix, we attempt to visualize it using a *heatmap* to analyze the situation appropriately.



	x1	x2	x3	x4	x5	x6	x7	x8	x9	x10	y
x1	1.00	0.17	-0.06	-0.08	-0.06	0.06	0.13	0.19	0.39	0.06	0.20
x2	0.17	1.00	-0.03	-0.12	-0.03	0.03	0.16	0.21	0.29	0.03	0.09
x3	-0.06	-0.03	1.00	0.03	0.00	-0.99	-0.07	-0.05	-0.02	-0.98	0.06
x4	-0.08	-0.12	0.03	1.00	0.05	-0.03	-0.09	-0.11	-0.05	-0.03	-0.19
x5	-0.06	-0.03	0.00	0.05	1.00	-0.00	-0.04	-0.14	-0.13	-0.00	0.08
x6	0.06	0.03	-0.99	-0.03	-0.00	1.00	0.07	0.05	0.02	0.99	-0.07
x7	0.13	0.16	-0.07	-0.09	-0.04	0.07	1.00	0.31	0.28	0.07	0.25
x8	0.19	0.21	-0.05	-0.11	-0.14	0.05	0.31	1.00	0.24	0.05	0.38
x9	0.39	0.29	-0.02	-0.05	-0.13	0.02	0.28	0.24	1.00	0.02	0.18
x10	0.06	0.03	-0.98	-0.03	-0.00	0.99	0.07	0.05	0.02	1.00	-0.06
y	0.20	0.09	0.06	-0.19	0.08	-0.07	0.25	0.38	0.18	-0.06	1.00

Figure 2.1: Correlation Matrix Heatmap

Interpreting the correlation matrix:

- The "red" cells with values close to `1` denote perfect positive correlation
- The "blue" cells with values close to `-1` denote perfect negative correlation
- The "light" cells with values close to `0` denote no correlation among those features

Analyzing the correlations, it can be seen that the features x_6 and x_{10} are positively correlated together (with value `0.99`) while being negatively correlated with x_3 (with values `-0.99` and `-0.98`). Additionally, analyzing their correlation with respect to the label y, it can be seen that all of them similarly hold a value close to `0.06`, providing no extra information compared to the other one for the classification.

After identifying highly correlated features, since we have a few number of features available, we will proceed with manually removing redundant features that are highly correlated. To this regard, we remove the features x_3 and x_{10}, preserving the feature x_6 in our dataset.

2.4 Feature Normalization

The two most common techniques for normalization are:

- **Min-Max Scaling**, which scales the data to a fixed range (usually `0` to `1`).
- **Z-score Normalization**, which standardizes the features to have a mean of `0` and a standard deviation of `1`.

The *Z-score Normalization* technique is more consistent with respect to Gaussian assumptions, and it is more reliable when dealing with algorithms that assume normal distribution centered around zero. Therefore, it naturally fits better our problem, and we will proceed with this technique. It can be computed as:

$$Z = \frac{X - \mu}{\sigma} \tag{2.1}$$

where X represents each feature, and the corresponding mean and standard deviation for each of the features are μ and σ, respectively.

2.5 Partitioning the Dataset

To maintain the integrity of our data and prevent data leakage, one approach is to partition the dataset into training and test sets, such that the model is trained on the training set and its performance is evaluated on the test set. This partitioning can be achieved as follows:

1. **Shuffle the Data:** The dataset is shuffled to randomize the order of the samples, ensuring a more representative distribution and reducing the likelihood of biased results due to any inherent order in the data.

2. **Split the Data:** The shuffled data is partitioned such that the first 80% of the samples are allocated to the training set, while the remaining 20% are reserved for the test set.

3. **Separate Features and Labels:** The target labels are separated from the feature variables, being stored in variables X (features) and y (labels) for both training and test sets.

This procedure ensures that the training and test sets are appropriately isolated, preserving the validity of our model evaluation.

However, since we aim to perform hyper-parameter optimization alongside our model implementation, we will adopt the *K-Fold Nested Cross-Validation* technique instead, which is fully discussed in the next section.

Chapter 3

Hyper-parameter Tuning and Model Evaluation

A common characteristic of many algorithms is that they include a set of hyper-parameters that are not determined by the algorithm itself, but rather chosen by a human. These hyper-parameters define a class of predictors, which can be represented as $\{A_\theta : \theta \in \Theta\}$. In order to tune the hyper-parameters, the *K-fold Cross-Validation* and *K-fold Nested Cross-Validation* techniques are among our choices, which are described in the following.

3.1 K-fold Cross-Validation

For a fixed given hyper-parameter θ, we can use this technique to estimate $\mathbb{E}[\ell_D(A_\theta(S))]$ as follows:

- Let S be the entire dataset of size m, where we partition it into K subsets (folds) $S_1, \ldots, S_K$ of size m/K each. S_i denotes the testing part while $S_{-i} \equiv S \setminus S_i$ denotes the training part for the i-th fold and $i = 1, \ldots, K$.
- In order to estimate the $\mathbb{E}[\ell_D(A_\theta(S))]$, we first run A on each *training part* S_{-i} of the folds $i = 1, \ldots, K$ and obtain the predictors $h_i = A(S_{-i}) \ldots, h_K = A(S_{-K})$. Then, We take an average error on the *testing part* of each fold:

$$\ell_{S_i}(h_i) = \frac{K}{m} \sum_{(\boldsymbol{x},y) \in S_i} \ell(y, h_i(\boldsymbol{x})) \tag{3.1}$$

- Finally, we compute the K-fold CV estimate denoted by $\ell_S^{CV}(A)$ by averaging these errors:

$$\ell_S^{CV}(A) = \frac{1}{K} \sum_{i=1}^{K} \ell_{S_i}(h_i) \tag{3.2}$$

Tuning hyper-parameters on a given training set aims to achieve the smallest risk. In practice, we aim to find $\theta^* \in \Theta$ such that:

$$\ell_D(A_{\theta^*}(S)) = \min_{\theta \in \Theta_0} \ell_D(A_\theta(S)) \tag{3.3}$$

After splitting the training set into S_{train} and S_{dev}, the algorithm is run on S_{train} once for each of the hyper-parameters in Θ_0, and the resulting predictors are tested on the development set S_{dev}. Choosing the hyper-parameters with the smallest error on the validation set, we obtain the final predictor by re-training the learning algorithm on the original training set we had before splitting.

3.2 K-fold Nested Cross-Validation

Tuning hyper-parameters via this technique aims to estimate the performance of A_θ on a typical training set of a given size when θ is tuned on the training set, which is computed by averaging the performance of predictors obtained with potentially different values of their hyper-parameters, according to the following pseudo-code:

Algorithm 1: K-fold Nested Cross-Validation

```
   Input: Dataset $S$
1  Split $S$ into $K$ folds $S_1, \dots, S_K$
2  for $i = 1, \dots, K$ do
3
4
5      Compute $S_{-i} \equiv S \setminus S_i$, the training part of $i$-th fold
6      Run CV on $S_{-i}$ for each $\theta \in \Theta_0$ and find $\theta_i = \underset{\theta \in \Theta_0}{\operatorname{argmin}} \ell^{CV}_{S_{-i}}(A_\theta)$
7      Re-train $A_{\theta_i}$ on the training part $S_{-i}$ to get the predictor $h_i = A_{\theta_i}(S_{-i})$
8      Compute $\varepsilon_i = \ell_{S_i}(h_i)$, the error of the testing part of $i$-th fold
9  end
   Output: $\frac{1}{K} \sum_{i=1}^{K} \varepsilon_i$
```

In the Appendix A.1, the Python implementation of this technique can be found. The `NestedCV` class is designed to take in a `dataset`, `model`, `param_grid`, `outer_K`, and `inner_K` as inputs, where for all experiments in the project we set `outer_K` and `inner_K` to `5` and `4`, respectively. It performs nested cross-validation for the specified model across various hyper-parameter combinations defined in the parameter grid. The class optimizes hyper-parameters using the validation set, trains the model on the training set, and evaluates performance on the test set.

This approach ensures a robust and sound evaluation process, avoiding data leakage. We will utilize this module in the following sections as we train and evaluate different models, reporting the `accuracy`, `precision`, and `recall` measures, which are defined as:

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} \tag{3.4}$$

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{3.5}$$

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{3.6}$$

Chapter 4

Model Implementation

In this section, we will implement the following machine learning algorithms:

1. The Perceptron algorithm
2. Support Vector Machines (SVMs) using the Pegasos algorithm
3. Regularized logistic classification (i.e., the Pegasos objective function with logistic loss instead of hinge loss)

In addition, we will attempt to evaluate these models with proper hyper-parameter optimization, reporting the training and test errors.

4.1 Perceptron

We know that a linear predictor for $\mathcal{X} \in \mathbb{R}^d$ is a function $h : \mathbb{R}^d \to \mathbb{R}$ defined as $h(x) = f(w^T x)$ for some $w \in \mathbb{R}^d$, where $f : \mathbb{R} \to \mathbb{R}$ is often referred to as an *activation function.* In linear classification tasks, we typically use $h(x) = \text{sgn}(w^T x)$, where:

$$\text{sgn}(z) = \begin{cases} 1 & \text{if } z > 0 \\ -1 & \text{otherwise} \end{cases} \tag{4.1}$$

In this case, the *zero-one* loss $\mathbb{I}\{h(x_t) \neq y_t\}$ can be written as $\mathbb{I}\{y_t w^T x_t \leq 0\}$. It should be noted that finding an efficient implementation of Empirical Risk Minimization (ERM) for linear classifiers using zero-one loss is unlikely, and the decision problem of finding h_S is NP-complete, even in the case of binary features.

However, in the linearly separable case, there exists at least a solution which can be found in polynomial time to solve the ERM problem, for which we can use the *Perceptron* algorithm for linear classifiers.

The Perceptron algorithm is a linear binary classifier that iteratively adjusts its weights based on the input features and the misclassified predictions, trying to find a homogeneous separating hyperplane which always terminates on linearly separable cases. It can also be seen as a single unit of an artificial neural network, where in the single layer case it's only capable of learning linearly separable patterns.

The general structure of a single Perceptron is illustrated in Figure 4.1.

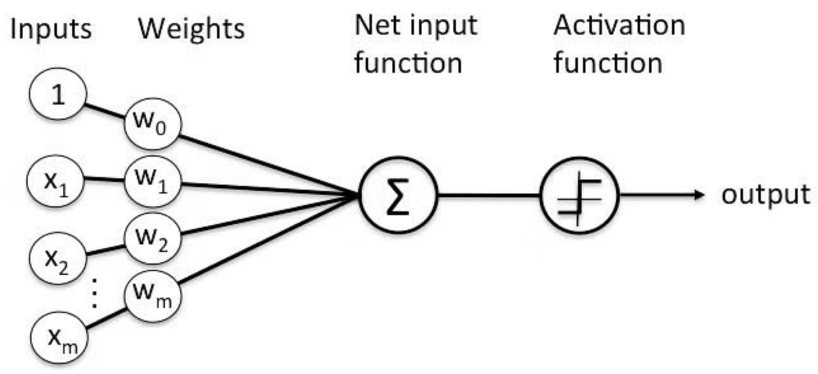


Figure 4.1: Perceptron Structure

We will implement the Perceptron, according to the following pseudo-code:

Algorithm 2: The Perceptron algorithm (for the linearly separable case)

```
   Input: Training set $(x_1, y_1), \dots, (x_m, y_m) \in \mathbb{R}^d \times \{-1, 1\}$
1  Initialize $\boldsymbol{w} = (0, \dots, 0)$
2  while true do
3      for $i = 1, \dots, m$ do  (epoch)
4          if $y_t \boldsymbol{w}^\top \boldsymbol{x}_t \le 0$ then
5              $\boldsymbol{w} \leftarrow \boldsymbol{w} + y_t \boldsymbol{x}_t$ (update)
6      end
7      if no update in last epoch then
8          break
9  end
   Output: $\boldsymbol{w}$
```

Appendix A.2 includes the Python implementation of the Perceptron algorithm. The hyper-parameter `weight_init` was included to test the effect of initializing the weights with zeros as opposed to the random initialization. Additionally, the `bias` term was also included to check it's effectiveness in the training procedure.

Using the `NestedCV` module, we attempted to tune the hyper-parameters, according to the grid provided as `param_grid = {'weight_init': ['zeros', 'random'], 'bias': [True, False], 'max_epochs': [20, 50, 120]}`. According to the best hyper-parameter combination on the validation set, the model was re-trained on the whole training set and evaluated using the test set. Overall, the hyper-parameter optimization took `5min 10s` to be completed.

On average, the training set achieved an accuracy of `67.38%`, precision of `69.74%`, and recall of `59.06%`, while the average test results were accuracy of `67.23%`, precision `69.92%`, and recall `58.95%`. Therefore, the Perceptron model's accuracy on both the training and test set hovers around `65-70%` and it does not show improvement with hyper-parameter tuning after a certain point. This fact suggests that we suffer from a mild underfitting, and the data may not be linearly separable.

Underfitting occurs when the model is too simple to capture the underlying patterns in the data. The Perceptron algorithm is a linear classifier, meaning it can only find a linear decision boundary to separate the classes. If the data is not linearly separable (i.e., the classes cannot be separated by a straight line in the feature space), the Perceptron won't be able to achieve high accuracy and the chosen features are insufficient to draw a clear linear boundary, which leads to underfitting.

In later chapters we will try to examine this hypothesis by both discussing the effect of adding polynomial features to the dataset and applying a kernel trick, checking if the model's performance improves significantly. If it does, then this is a strong indication that the original data was not linearly separable.

4.2 Pegasos for SVM

In general, the SVM optimization problem can be defined as:

- For a given training set $(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_m, y_m) \in \mathbb{R}^d \times \{-1, 1\}$ which is **linearly separable**, SVM outputs the linear classifier corresponding to the unique solution $\boldsymbol{w}^* \in \mathbb{R}^d$ of the *convex optimization problem* with linear constraints defined as:

$$\min_{\boldsymbol{w} \in \mathbb{R}^d} \frac{1}{2}\|\boldsymbol{w}\|^2, \quad \text{such that} \quad y_t \boldsymbol{w}^\top \boldsymbol{x}_t \geq 1 \quad \text{for} \quad t = 1, \ldots, m \tag{4.2}$$

  We can also say $\boldsymbol{w}^*$ geometrically corresponds to the *maximum margin* separating hyperplane, and the maximum margin separator $\boldsymbol{u}^*$ is a solution to:

$$\max_{\boldsymbol{u}:\|\boldsymbol{u}\|=1} \min_t y_t \boldsymbol{u}^\top \boldsymbol{x}_t \tag{4.3}$$

- For a given training set $(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_m, y_m) \in \mathbb{R}^d \times \{-1, 1\}$ which is **not linearly separable**, the constraints can be satisfied up to a certain level called *slack variables* $\xi = (\xi_1, \ldots, \xi_m)$, measuring how much each margin constraint is violated by a potential solution $\boldsymbol{w}$. The average of these violations is then added to the objective function of the SVM, and a *regularization* parameter $\lambda > 0$ is introduced to balance the two terms. Finally, we get:

$$\min_{(\boldsymbol{w},\boldsymbol{\xi}) \in \mathbb{R}^{d+m}} \frac{\lambda}{2}\|\boldsymbol{w}\|^2 + \frac{1}{m}\sum_{t=1}^{m} \xi_t \quad \text{s.t. for } t = 1, \ldots, m \quad \begin{cases} y_t \boldsymbol{w}^\top \boldsymbol{x}_t \geq 1 - \xi_t \\ \xi_t \geq 0 \end{cases} \tag{4.4}$$

  Considering the constraints on slack variables, to minimize each ξ_t we can set:

$$\xi_t = \begin{cases} 1 - y_t \boldsymbol{w}^\top \boldsymbol{x}_t & \text{if } y_t \boldsymbol{w}^\top \boldsymbol{x}_t < 1 \\ 0 & \text{otherwise} \end{cases} = \left[1 - y_t \boldsymbol{w}^\top \boldsymbol{x}_t\right]_+ = \underbrace{h_t(\boldsymbol{w})}_{\text{hinge loss}} \tag{4.5}$$

  So, the SVM problem can be reformulated as:

$$\min_{\boldsymbol{w} \in \mathbb{R}^d} \underbrace{\frac{\lambda}{2}\|\boldsymbol{w}\|^2 + \frac{1}{m}\sum_{t=1}^{m} h_t(\boldsymbol{w})}_{F(\boldsymbol{w})} \tag{4.6}$$

Pegasos is an instance of stochastic gradient descent (SGD) over online gradient descent (OGD) on SVM objective function, which is applied on λ-strongly convex functions to the set of losses $\ell_1, \dots, \ell_m$, where $\ell_t(w) = h_t(w) + \frac{\lambda}{2}\|w\|^2$, run over a sequence of examples randomly drawn from the training set, aiming to minimize F which is described as:

$$F(w) = \frac{1}{m}\sum_{t=1}^{m} \ell_t(w) \tag{4.7}$$

Fixing a realization of random variables $s_1, \dots, s_t$ and $\eta_t = \frac{1}{\lambda t}$, we have:

$$\nabla \ell_{s_t}(w_t) = -y_{s_t} x_{s_t} \mathbb{I}\{h_{s_t}(w_t) > 0\} + \lambda w_t \tag{4.8}$$

We will implement the Pegasos for SVM, according to the following pseudo-code:

```
Algorithm 3: Pegasos for SVM
  Input: Training set $S$, number of rounds $T$, regularization coefficient $\lambda > 0$
1 Initialize $\boldsymbol{w}_1 = (0, \dots, 0)$
2 for $t = 1, \dots, T$ do
3     Draw uniformly at random $Z_t \sim \{1, \dots, m\}$, obtaining $(\boldsymbol{x}_{Z_t}, y_{Z_t})$ from $S$
4     Set $\boldsymbol{w}_{t+1} \leftarrow \boldsymbol{w}_t - \eta_t \nabla \ell_{Z_t}(\boldsymbol{w}_t)$
5 end
  Output: $\bar{\boldsymbol{w}} = \frac{1}{T}\sum_{t=1}^{T} \boldsymbol{w}_t$
```

The implementation for `PegasosSVM` module is provided at the Appendix A.3. This class contains both hinge and logistic loss functions since the majority of the code is shared with the *logistic classification* algorithm. Hence, the hyper-parameter `loss` can be defined as either `"hinge"` or `"logistic"`, `T` denotes the number of rounds, and `lambda_param` controls the regularization coefficient.

According to the `param_grid = {'loss': ['hinge'], 'T': [1000, 2000, 5000], 'lambda_param': [0.001, 0.01, 0.1]}` for the `NestedCV` module, the hyper-parameters were tuned, where for this evaluation we fixed the loss to be explicitly `'hinge'`. The best hyperparameter combination by the majority of folds on the validation set was selected as `{'loss': 'hinge', 'T': 5000, 'lambda_param': 0.1}`, which was further re-trained accordingly on the training set, and evaluated via test set. The hyper-parameter optimization only took `17s` in total.

The training set achieved an average accuracy of `71.63%`, a precision of `70.43%`, and a recall of `71.23%`. In comparison, the test set produced an average accuracy of `71.57%`, a precision of `70.39%`, and a recall of `71.16%`. Both the training and test sets consistently yield results around `70%`. Despite efforts in hyper-parameter optimization, the results cannot be further improved after a certain point, likely due to the model's simplicity which limits its ability to capture complex patterns, similar to the evaluation seen in the Perceptron model.

4.3 Regularized Logistic Classification

The zero-one loss can be shown to be NP-hard, being unlikely to be solved using a polynomial time algorithm. In addition, the loss function is non-smooth and non-convex, where small changes in w can change the loss by a lot, hence we use convex approximations to the zero-one loss. While the SVM is based on hinge loss, the logistic regression model instead uses a loss function called *log-loss*, which has a real gradient (not just a sub-gradient like hinge) and a smooth shape.

In order to implement the regularized logistic classification, we should adopt the Pegasos objective function along with *logistic loss*. Since the structures of the two algorithms are quite similar, it makes sense to modify the existing `PegasosSVM` class from the previous section to accommodate both types of loss functions, including the logistic loss which is defined as:

$$\ell(y, \hat{y}) = \log_2(1 + e^{-y\hat{y}}) \tag{4.9}$$

Given the t-th example of the training set $S = \{(x_1, y_1), \ldots, (x_m, y_m)\}$ in the case of linear models where $\hat{y} = g(\boldsymbol{x}) = \boldsymbol{w}^\top \boldsymbol{x}$ and $w \in \mathbb{R}^d$, the logistic loss for logistic regression can be written as below, where a regularization term is often added to enforce stability and avoid overfitting (Regularized LR):

$$\ell_t(w) = \log_2(1 + e^{-y_t w^\top x_t}) + \frac{\lambda}{2}\|w\|^2 \tag{4.10}$$

Fixing a realization of random variables $s_1, \ldots, s_t$ and $\eta_t = \frac{1}{\lambda t}$, we have:

$$\nabla \ell_{s_t}(w_t) = \frac{-\sigma\left(-y_{s_t} w_t^\top x_{s_t}\right)}{\ln 2} y_{s_t} x_{s_t} + \lambda w_t \tag{4.11}$$

where the function $\sigma : \mathbb{R} \to \mathbb{R}$, called logistic, is defined as:

$$\sigma(z) = \frac{1}{1 + e^{-z}} \in (0, 1) \tag{4.12}$$

Again, the implementation for `PegasosSVM` module containing the logistic loss can be found in Appendix A.3. Given the `param_grid = {'loss': ['logistic'], 'T': [1000, 2000, 5000], 'lambda_param': [0.001, 0.01, 0.1]}` for the `NestedCV` module, the hyper-parameter tuning was performed, where the `'logistic'` loss was explicitly fixed for evaluation. All the folds confirmed that `{'lambda_param': 0.1}` is the best choice for regularization parameter, while the choice of `T` varied by folds on the validation from one another, re-training on the training set and evaluating on the test set accordingly. The whole hyper-parameter optimization process only took around `17s` to complete.

The training set showed an average accuracy of `71.57%`, with a precision of `70.38%` and a recall of `71.09%`. For the test set, the average accuracy was `71.21%`, precision `70.07%`, and recall `70.59%`. The results are again somehow similar to the previous algorithms, stuck around `70%` accuracy, and there seems to be underfitting due to the simplicity of the model.

Chapter 5

Feature Expansion

Linear predictors $h(x) = \text{sgn}(w^T x)$ for $\mathcal{X} \in \mathbb{R}^d$ and $w \in \mathbb{R}^d$ have as many learnable parameters as the number of features. Therefore, the variance (estimation) error is low, while they may potentially suffer from a large bias (approximation) error.

Feature expansion is a well-known technique to overcome the bias issue by introducing new features through nonlinear combinations of the base features. Training a linear predictor upon a feature-expanded training set, we can boost the performance of classifiers to learn new shapes, such as surfaces including ellipses, parabolas, and hyperbolas in the second-degree polynomial feature expansion case. More formally:

$$(x_1, ..., x_d) \in \mathbb{R}^d \xrightarrow{\phi} \phi(x) \in \mathbb{R}^N \quad \text{where} \quad N \gg d \tag{5.1}$$

In general with d features, the feature map $\phi : \mathbb{R}^d \to \mathbb{R}^N$ use as features all monomials of degree up to N, such that:

$$\prod_{s=1}^{k} x_{v_s} \quad \text{for all} \quad v \in \{1, ..., d\}^k \quad \text{and for all} \quad k = 0, 1, ..., n \tag{5.2}$$

In this project, we will use the polynomial feature expansion of degree 2, re-training the models implemented in the previous chapter for such a feature-expanded training set. In the Appendix A.4, the corresponding implementation of `FeatureExpandedNestedCV` is provided, which inherits all the functions and parameters from `NestedCV`, adding the second-degree polynomial expansion on top of them. All the experiments were run using the same `param_grid` and parameters as before, and the results are summarized below.

5.1 Feature-Expanded Perceptron

The entire hyper-parameter optimization process took `3min 45s` to be finished. The training set achieved an average accuracy of `92.69%`, with a precision of `90.52%` and a recall of `95.02%`. The test set had an average accuracy of `92.38%`, with precision at `89.95%` and recall at `95.02%`.

5.2 Feature-Expanded Pegasos for SVM

Hyper-parameter tuning was performed in a `20s` time-frame. The average results on the training set were an accuracy of `91.61%`, precision of `92.50%`, and recall of `89.93%`. The test set yielded an average accuracy of `91.94%`, precision of `92.94%`, and recall of `90.18%`.

5.3 Feature-Expanded Logistic Classification

The hyper-parameter optimization process was completed in `20s`. The training set achieved an average accuracy of `91.01%`, a precision of `90.25%`, and a recall of `91.31%`. The test set results showed an average accuracy of `90.74%`, with a precision of `89.87%` and a recall of `91.13%`.

Chapter 6

Kernel Methods

We know that $N = \Theta(d^n)$ is exponential in the degree n, and computing ϕ becomes practically impossible even for moderately large n. The computational complexity of using feature-expanded training sets for training linear predictors can be handled using *kernels*, which helps us to find an efficiently computable kernel function $K : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ such that we have:

$$K(x, x') = \phi(x)^T \phi(x') \quad \text{for all } x, x' \in \mathbb{R}^d \tag{6.1}$$

The formulas for two of the most commonly used kernels, **polynomial** and **Gaussian** kernels, are provided below respectively:

$$K_n(x, x') = (1 + x^T x')^n \tag{6.2}$$

$$K_\gamma(x, x') = \exp\left(-\frac{\|x - x'\|^2}{2\gamma}\right) \tag{6.3}$$

It should be noted that linear predictors in Reproducing Kernel Hilbert Spaces (RKHS) can suffer from overfitting. For a *polynomial* kernel, increasing the degree n reduces training error since higher-degree curves can better separate training points with complex decision boundaries, leading to overfitting when n is too large. For *Gaussian* kernels, if the γ parameter (corresponding to the width of the Gaussians) is too small with respect to the distance $\|x - x'\|^2$ between training and test points, the resulting predictor converges to 1-NN, leading to a training error equal or close to zero since they are never misclassified. In the following, we will attempt to implement:

1. **Kernelized Perceptron** with *Gaussian* and *polynomial* kernels
2. **Kernelized Pegasos for SVM** with *Gaussian* and *polynomial* kernels

6.1 Kernelized Perceptron

Given a kernel K, the linear classifier generated by the Perceptron can be written as:

$$h_K(x) = \operatorname{sgn}\left(\sum_{s \in S} y_s K(x_s, x)\right) \tag{6.4}$$

We will implement the kernelized Perceptron according to the following pseudo-code:

Algorithm 4: Kernel Perceptron

```
1 Let $S$ be the empty set, i.e. $S \leftarrow \emptyset$
2 for $t = 1, 2, \ldots$ do
3     Get the next example $(\boldsymbol{x}_t, y_t)$
4     Compute $\hat{y}_t = \operatorname{sgn}\left(\sum_{s \in S} y_s K(\boldsymbol{x}_s, \boldsymbol{x}_t)\right)$
5     if $\hat{y}_t \neq y_t$ then
6         $S \leftarrow S \cup \{t\}$
7     end
8 end
```

The implementation of the `KernelPerceptron` module is provided at the Appendix A.5. Using the `kernel` parameter, we can choose either polynomial or Gaussian kernels to be used in the computation, where `degree` is used to define the polynomial degree and `gamma` is utilized as the input for Gaussian kernel. `max_epochs` define the maximum epochs for the learning process, terminating if the convergence was not achieved upon this threshold. Following are the experiments performed on the kernelized Perceptron module.

6.1.1 Polynomial Kernel

Running for the polynomial kernel with `param_grid = {'kernel': ['polynomial'], 'degree': [2, 3]}` for the `NestedCV` module, the hyper-parameter tuning was performed, where the `'polynomial'` kernel was explicitly fixed for evaluation. `degree = 2` was the best hyper-parameter combination selected by the folds on the validation set, which confirms that the second-degree polynomial had the best performance on this dataset. In a further step, using the best hyper-parameters in each fold the training set was re-trained and the model was evaluated on the test set accordingly. The whole hyper-parameter optimization process took `2h 2min 10s` to be completed.

The training set showed an average accuracy of `94.69%`, with a precision of `94.50%` and a recall of `94.55%`. For the test set, the average accuracy was `94.46%`, precision `94.36%`, and recall `94.20%`, which confirms the effectiveness of using polynomial kernel Perceptron along our dataset.

6.1.2 Gaussian Kernel

For the Gaussian kernel, hyper-parameter tuning was conducted using `param_grid = {'kernel': ['gaussian'], 'gamma': [0.01, 0.1, 1]}` within the `NestedCV` module, with the `'gaussian'` kernel explicitly set for evaluation. The folds on the validation set identified `gamma = 1` as the optimal hyper-parameter. The model was re-trained on each fold's training set using the best hyper-parameters and then evaluated on the test set. The entire hyper-parameter optimization process took `9h 16min 31s` to complete.

The training set achieved an average accuracy of `99.03%`, with a precision of `99.08%` and a recall of `98.91%`. On the test set, the average accuracy was `94.10%`, with a precision of `94.07%` and a recall of `93.72%`, validating the success of applying the Gaussian kernel Perceptron to this dataset.

6.2 Kernelized Pegasos for SVM

One of the key advantages of SVMs is that they can be utilized with kernels rather than relying on direct access to the feature vectors x, and be represented as a linear combination of the training instances. To achieve this, instead of considering predictors as linear functions of the training examples, we consider them as linear functions of some implicit mapping $\phi(x)$. The minimization problem then becomes:

$$\min_{\boldsymbol{w}\in\mathbb{R}^d} \underbrace{\frac{\lambda}{2}\|\boldsymbol{w}\|^2 + \frac{1}{m}\sum_{t=1}^{m}\ell(w;(\phi(x_t),y_t))}_{F(\boldsymbol{w})} \tag{6.5}$$

where

$$\ell(w;(\phi(x_t),y_t)) = \max\{0, 1 - y_t\langle w, \phi(x_t)\rangle\} \tag{6.6}$$

and considering that the mapping $\phi(\cdot)$ is implicitly applied through a kernel function $K(\mathbf{x},\mathbf{x}') = \langle\phi(\mathbf{x}),\phi(\mathbf{x}')\rangle$, thereby producing the inner products after the transformation by $\phi(\cdot)$.

We implement the kernelized Pegasos for SVM according to the pseudo-code provided at their official paper [2], which is provided below:

Algorithm 5: Kernelized Pegasos for SVM

```
   Input: Training set $(x_1, y_1), \dots, (x_m, y_m) \in \mathbb{R}^d \times \{-1, 1\}$
1  Let $T$ be number of rounds, $\lambda > 0$ regularization coefficient
2  Initialize $\alpha_1 = (0, \dots, 0)$
3  for $t = 1, 2, \dots, T$ do
4      Draw uniformly at random $Z_t \sim \{1, \dots, m\}$, obtaining $(x_{Z_t}, y_{Z_t})$ from training set
5      for all $j \neq Z_t$ do
6          Set $\alpha_{t+1}[j] = \alpha_t[j]$
7      end
8      if $y_{Z_t}\frac{1}{\lambda t}\sum_j \alpha_t[j] y_j K(x_{Z_t}, x_j) < 1$ then
9          Set $\alpha_{t+1}[Z_t] = \alpha_t[Z_t] + 1$
10     else
11         Set $\alpha_{t+1}[Z_t] = \alpha_t[Z_t]$
12     end
13 end
   Output: $\alpha_{T+1}$
```

The `KernelPegasosSVM` module is implemented and detailed in Appendix A.6. Similar to the module implemented for `KernelPerceptron`, this class also requires `kernel`, `degree`, and `gamma` as parameters. Additionally, we have `lambda_param` as the regularization coefficient, and `T` defining the number of rounds is replaced for the `max_epochs` in the `KernelPerceptron` module. The following presents the experiments conducted using the kernelized Pegasos for SVM.

6.2.1 Polynomial Kernel

Using `NestedCV` module with `param_grid = {'kernel': ['polynomial'], 'degree': [2, 3], 'lambda_param': [0.01, 0.1], 'T': [1000, 2000]}`, the hyper-parameter tuning for the polynomial kernel was carried out, where the `'polynomial'` kernel was fixed for evaluation. Most of the folds on the validation set determined that `{'degree': 2, 'lambda_param': 0.1, 'T': 2000}` was the best-performing hyper-parameter combination, indicating that the second-degree polynomial achieved the highest performance on this dataset. Subsequently, the model was re-trained on the training set of each fold using these optimal hyper-parameters and then evaluated on the test set. The entire hyper-parameter optimization process was completed in `2min 51s`.

The training set produced an average accuracy of `90.89%`, with a precision of `91.03%` and a recall of `90.16%`. For the test set, the average accuracy was `90.32%`, with a precision of `90.22%` and a recall of `89.80%`, confirming the effectiveness of applying the polynomial kernel Pegasos for SVM to this dataset.

6.2.2 Gaussian Kernel

For the Gaussian kernel, hyper-parameter tuning was performed using `param_grid = {'kernel': ['gaussian'], 'gamma': [0.01, 0.1, 1], 'lambda_param': [1e-2, 1e-1], 'T': [1000, 2000]}` within the `NestedCV` module, with the `'gaussian'` kernel fixed for evaluation. The model was then re-trained on the training set of each fold using these best hyper-parameters found and subsequently evaluated on the test set. The entire hyper-parameter optimization process took `7min 24s` to complete.

The training set achieved an average accuracy of `85.91%`, with a precision of `91.11%` and a recall of `78.58%`. On the test set, the average accuracy was `84.87%`, with a precision of `89.84%` and a recall of `77.58%`, validating the success of applying the Gaussian kernel Perceptron to this dataset. The results are not as good as the ones obtained from kernelized Perceptron, yet they are still promising.

Chapter 7

Results

Table 7.1 illustrates and summarizes the performance of all the algorithms investigated in this project. The columns `Accuracy`, `Precision`, and `Recall` demonstrate the corresponding metrics of each algorithm on the test set only. The `Runtime` column, on the other hand, shows the whole amount of time that was taken to perform the hyper-parameter optimization to train and evaluate each of the models.

Algorithm	**Accuracy**	**Precision**	**Recall**	**Runtime**
Perceptron	67.23%	69.92%	58.95%	05:10
Pegasos for SVM	71.57%	70.39%	71.16%	00:17
Logistic Classification	71.21%	70.07%	70.59%	00:17
Feature-Expanded Perceptron	92.38%	89.95%	95.02%	03:45
Feature-Expanded Pegasos for SVM	91.94%	92.94%	90.18%	00:20
Feature-Expanded Logistic Classification	90.74%	89.87%	91.13%	00:20
Polynomial Kernel Perceptron	94.46%	94.36%	94.20%	02:02:10
Gaussian Kernel Perceptron	94.10%	94.07%	93.72%	09:16:31
Polynomial Kernel Pegasos for SVM	90.32%	90.22%	89.80%	02:51
Gaussian Kernel Pegasos for SVM	84.87%	89.84%	77.58%	07:24

Table 7.1: Performance metrics of various algorithms

In order to better comprehend the results reported above, we attempt to visualize these details using line plots, separately for performance measures in Fig 7.1, and runtime measures in Fig 7.2. In the following, we will demonstrate the key points that was observed during the evaluation of results.

Observation I: Overall, the `Polynomial Kernel Perceptron` can be selected as the best model performing on this dataset, surpassing all the other models in every evaluation criterion, while the baseline `Perceptron` achieved the lowest performance on average.

Observation II: Generally, it is evident that the baseline *Perceptron* and *Pegasos for SVM* algorithms (with both hinge and logistic loss) were unable to completely capture the complexity of the models. The reason is that these models are linear classifiers, hence they are not capable of finding a separating hyperplane if the data is not linearly separable. As the results confirm, both the feature-expanded and kernelized versions of the baseline models could outperform them by almost improving `25%` accuracy upon the test set.

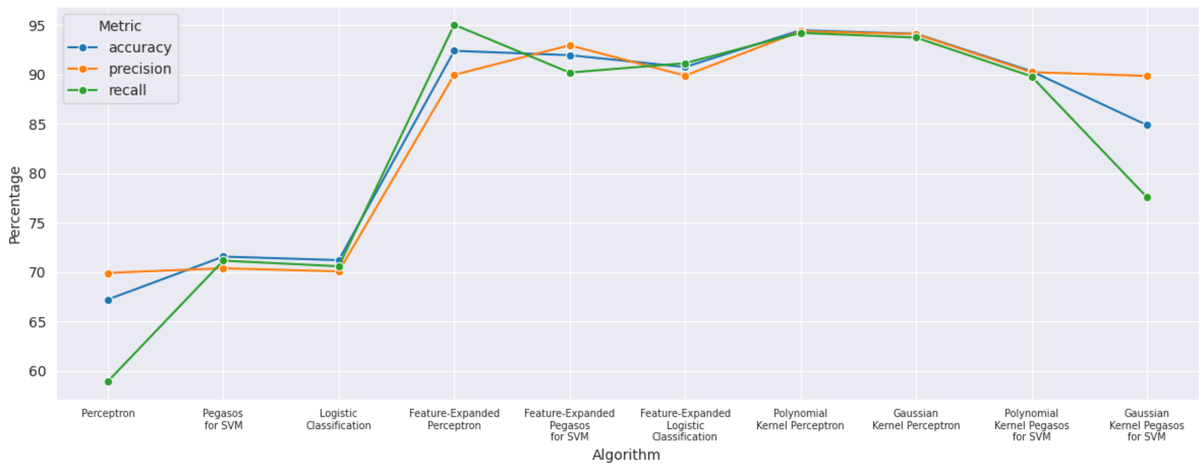


Figure 7.1: Average accuracy, precision, and recall for algorithms on the test set

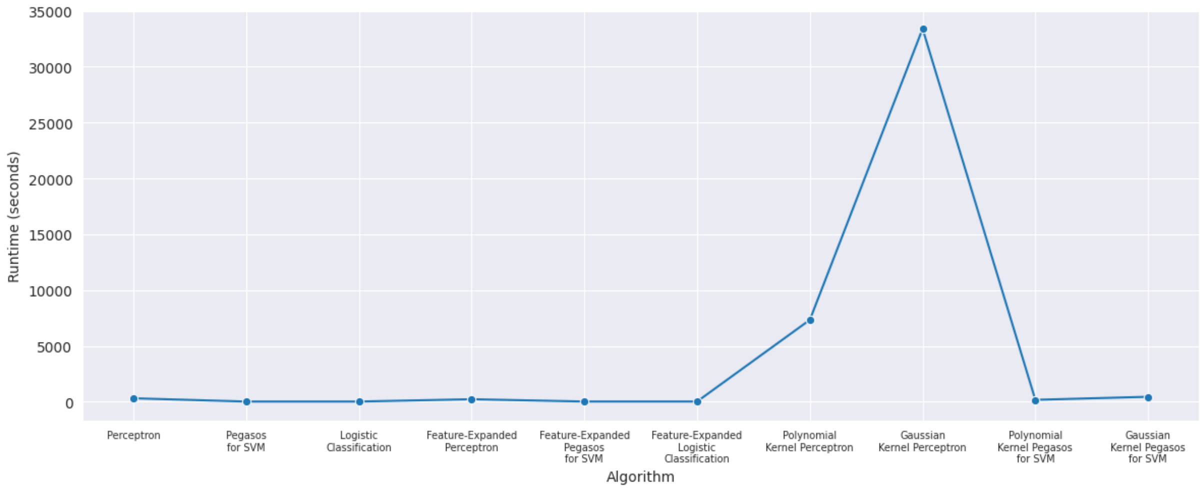


Figure 7.2: Runtime of hyper-parameter optimization on the algorithms

Observation III: The runtime of hyper-parameter optimization for the `Gaussian Kernel Perceptron` was significantly higher than all other methods, taking more than 9 hours to be performed, followed by almost 2 hours dedicated to the tuning of `Polynomial Kernel Perceptron`. The main reason for this low performance is related to the implementation techniques. In this project, the mentality was to implement the original pseudo-code dedicated to each algorithm directly, without modifying their behavior. Taking a look at the implementation in Appendix A.5, in the `_kernelized_predict()` function we iterate over all the misclassified samples that were stored in the set S, performing the calculations among such cases. This implementation is highly inefficient since the multiplications are performed single-handedly one at a time. Instead, we could benefit from a vectorized technique allowed by `NumPy`, and use a vector indicating the presence of each element rather than dealing with the set S. This strategy has been adopted in the `Kernel Pegasos for SVM` which is found in Appendix A.6, showcasing its success in achieving a lower amount of time for training.

Chapter 8

Conclusion

Throughout this project, we demonstrated the implementation and evaluation of various baseline machine learning algorithms—Perceptron, Pegasos SVM, and regularized logistic classification—for label classification based on numerical features. In addition, the feature-expanded and kernelized versions of such algorithms were investigated to analyze their performance. The following conclusions were drawn from the project.

Conclusion I: It was observed that the baseline models could achieve limited performance due to their linear nature, particularly on non-linearly separable data. However, by employing polynomial second-degree feature-expansion and kernel methods, significant improvements and a remarkable increase of up to 25% in accuracy were obtained. Specifically, the Polynomial Kernel Perceptron was identified as the best-performing model, surpassing all others in accuracy, precision, and recall.

Conclusion II: The implementation of the Kernel Perceptron module faced a substantial computational cost, particularly Gaussian Kernel Perceptron requiring over 9 hours for hyper-parameter optimization, and polynomial Kernel Perceptron taking over 2 hours for completion. The analysis further revealed that implementation choices significantly impact runtime performance, as the inefficiency in the original kernel Perceptron implementation was identified and discussed, with recommendations for optimization.

Conclusion III: In most cases, the difference between the training error and test error was negligible. This observation suggests that overfitting was not an issue, as none of the models performed exceptionally well on the training set while failing on the test set. However, mild underfitting was observed in the baseline models. This can be due to the fact that these models are linear classifiers, which are inherently limited in their ability to identify a separating hyperplane in non-linearly separable data.

Bibliography

[1] Nicolò Cesa-Bianchi. *Dataset for Kernelized Linear Classification Project*. 2024. URL: https://drive.google.com/file/d/1HeP6uwyG3oMcJnViG4yCMmzmSeTa1mko.

[2] Shai Shalev-Shwartz, Yoram Singer, and Nathan Srebro. "Pegasos: Primal estimated sub-gradient solver for svm". In: *Proceedings of the 24th international conference on Machine learning*. 2007, pp. 807–814.

[3] Ali Tabaraei. *Kernelized Linear Classification*. https://github.com/tabaraei/Kernelized-Linear-Classification. 2024.

Appendix A

Python Modules

A.1 K-fold Nested Cross-Validation

```python

class NestedCV:
   def __init__(self, dataset, model, param_grid, outer_K=5, inner_K=4):
      self.X = dataset.drop(columns='y').to_numpy()
      self.y = dataset['y'].to_numpy()
      self.n_samples, self.n_features = self.X.shape
      self.model = model
      self.param_grid = [dict(zip(keys, p)) for keys, values in
          [zip(*param_grid.items())] for p in product(*values)]
      self.outer_K = outer_K
      self.inner_K = inner_K

   def _zero_one_loss(self, y_true, y_pred):
      return np.sum(y_true != y_pred)

   def _compute_metrics(self, y_true, y_pred):
      TP = np.sum((y_true == 1) & (y_pred == 1))
      TN = np.sum((y_true == -1) & (y_pred == -1))
      FP = np.sum((y_true == -1) & (y_pred == 1))
      FN = np.sum((y_true == 1) & (y_pred == -1))

      accuracy = (TP + TN) / (TP + TN + FP + FN)
      precision = TP / (TP + FP) if (TP + FP) > 0 else 1.0
      recall = TP / (TP + FN) if (TP + FN) > 0 else 1.0
      return accuracy, precision, recall

   def _KFold(self, K, n_samples, indices=None):
      if indices is None: indices = np.arange(n_samples)
      np.random.shuffle(indices)
      split_points = np.linspace(start=0, stop=n_samples, num=K+1, dtype=int)
```

```
        test_indices = [indices[split_points[i]:split_points[i+1]]
            for i in range(K)]
        train_indices = [np.setdiff1d(indices, test_idx)
            for test_idx in test_indices]
        return zip(train_indices, test_indices)

    def train(self):
        # Outer CV, evaluating the best hyperparameter combination
        self.training_results, self.test_results = list(), list()
        for fold, (outer_train_indices, test_indices) in
            enumerate(self._KFold(K=self.outer_K, n_samples=self.n_samples)):

            # Inner CV, finding the best hyperparameter combination
            print(f'Fold {fold+1}')
            min_loss = np.inf
            best_params = None
            for params in self.param_grid:
                losses = list()
                for train_indices, validation_indices in self._KFold(
                        K=self.inner_K,
                        n_samples=len(outer_train_indices),
                        indices=outer_train_indices
                    ):
                    model = self.model(**params)
                    model.train(self.X[train_indices], self.y[train_indices])
                    y_pred = model.predict(self.X[validation_indices])
                    loss = self._zero_one_loss(self.y[validation_indices], y_pred)
                    losses.append(loss)

                avg_loss = np.mean(losses)
                print(f'Hyperparameter combination: {params}',
                    f'Average Loss: {avg_loss}')
                if avg_loss < min_loss:
                    min_loss = avg_loss
                    best_params = params

            print(f'Retraining the model with the best hyperparameter combination:
                {best_params}')
            model = self.model(**best_params)
            model.train(self.X[outer_train_indices], self.y[outer_train_indices])

            # Training error
            y_pred_train = model.predict(self.X[outer_train_indices])
            accuracy, precision, recall =
                self._compute_metrics(self.y[outer_train_indices], y_pred_train)
            self.training_results.append([accuracy, precision, recall])
            print(f'Training Set Evaluation: Accuracy {accuracy:.2%}',
                f'Precision {precision:.2%}, Recall {recall:.2%}')
```

```

          # Test error
          y_pred_test = model.predict(self.X[test_indices])
          accuracy, precision, recall =
              self._compute_metrics(self.y[test_indices], y_pred_test)
          self.test_results.append([accuracy, precision, recall])
          print(f'Test Set Evaluation: Accuracy {accuracy:.2%},
              f'Precision {precision:.2%}, Recall {recall:.2%}')
          print('-'*80)

      # Average results
      accuracy, precision, recall = np.mean(self.training_results, axis=0)
      print(f'Average Training Results: Accuracy {accuracy:.2%},
          f'Precision {precision:.2%}, Recall {recall:.2%}')
      accuracy, precision, recall = np.mean(self.test_results, axis=0)
      print(f'Average Test Results: Accuracy {accuracy:.2%},
          f'Precision {precision:.2%}, Recall {recall:.2%}')
```

A.2 Perceptron Model Implementation

```

class Perceptron:
   def __init__(self, weight_init='zeros', bias=True, max_epochs=500):
      self.max_epochs = max_epochs
      self.weight_init = weight_init
      self.bias = bias
      self.sgn = lambda z: np.where(z >= 0, 1, -1)

   def train(self, X_train, y_train):
      X, y = X_train.copy(), y_train.copy()
      n_samples, n_features = X.shape
      self.w = np.zeros(n_features) if self.weight_init == 'zeros'
          else np.random.randn(n_features) * 0.01
      self.b = 0
      epoch = 0

      while True:
          epoch += 1
          update = False

          if self.bias:
              for t in range(n_samples):
                  if y[t] * (np.dot(self.w, X[t]) + self.b) <= 0:
                      self.w += y[t] * X[t]
                      self.b += y[t]
                      update = True
```

```python

        else:
            for t in range(n_samples):
                if y[t] * np.dot(self.w, X[t]) <= 0:
                    self.w += y[t] * X[t]
                    update = True

        if (not update) or (epoch >= self.max_epochs):
            break

    def predict(self, X_test):
        return self.sgn(np.dot(X_test, self.w) + (self.b if self.bias else 0))

```

A.3 Pegasos SVM Model Implementation (hinge and logistic)

```python

class PegasosSVM:
    def __init__(self, loss='hinge', T=1000, lambda_param=0.01):
        self.T = T
        self.lambda_param = lambda_param
        self.loss = loss
        self.sgn = lambda z: np.where(z >= 0, 1, -1)

    def _hinge(self, z_t):
        slack = self.y[z_t] * np.dot(self.w, self.X[z_t])
        return max(0, 1 - slack)

    def _logistic(self, z_t):
        z = -self.y[z_t] * np.dot(self.w, self.X[z_t])
        if z >= 0:
            return 1 / (1 + np.exp(-z))
        else:
            exp_z = np.exp(z)
            return exp_z / (1 + exp_z)

    def _gradient_loss(self, z_t):
        regularization_grad = self.lambda_param * self.w

        if self.loss == 'hinge':
            hinge_grad = -self.y[z_t] * self.X[z_t] * (self._hinge(z_t) > 0)
            return hinge_grad + regularization_grad

        elif self.loss == 'logistic':
            logistic_grad = -(self._logistic(z_t) / np.log(2))
```

```python
                            * self.y[z_t] * self.X[z_t]
            return logistic_grad + regularization_grad

    def train(self, X_train, y_train):
        self.X, self.y = X_train.copy(), y_train.copy()
        n_samples, n_features = self.X.shape
        self.w = np.zeros(n_features)
        self.w_sum = np.zeros(n_features)
        if self.T > n_samples: self.T = n_samples

        for t in range(self.T):
            eta_t = 1 / (self.lambda_param * (t+1))
            z_t = np.random.randint(0, n_samples)
            self.w -= eta_t * self._gradient_loss(z_t)
            self.w_sum += self.w

        self.w = self.w_sum / self.T

    def predict(self, X_test):
        return self.sgn(np.dot(X_test, self.w))

```

A.4 Polynomial Second-degree Feature Expanded Nested Cross-Validation

```python

class FeatureExpandedNestedCV(NestedCV):
    def __init__(self, dataset, model, param_grid, outer_K=5, inner_K=4):
        super().__init__(dataset, model, param_grid, outer_K, inner_K)

    def second_degree_feature_expansion(self):
        X_expanded = self.X.copy()

        for i in range(self.n_features):
            for j in range(i, self.n_features):
                new_feature = (X_expanded[:, i] * X_expanded[:, j]).reshape(-1, 1)
                X_expanded = np.hstack((X_expanded, new_feature))

        self.X = X_expanded
        self.n_features = self.X.shape[1]

```

A.5 Kernelized Perceptron

```python

class KernelPerceptron:
   def __init__(self, kernel='polynomial', degree=3, gamma=1, max_epochs=10):
      self.kernel = kernel
      self.degree = degree
      self.gamma = gamma
      self.max_epochs = max_epochs
      self.sgn = lambda z: np.where(z >= 0, 1, -1)

   def _K(self, x1, x2):
      if self.kernel == 'polynomial':
         return (1 + np.dot(x1, x2)) ** self.degree
      elif self.kernel == 'gaussian':
         return np.exp(-(np.linalg.norm(x1 - x2) ** 2) / (2*self.gamma))

   def _kernelized_predict(self, X_t):
      return self.sgn(np.sum([self.y[s] * self._K(self.X[s], X_t)
            for s in self.S]))

   def train(self, X_train, y_train):
      self.X, self.y = X_train.copy(), y_train.copy()
      self.S = set()
      n_samples, n_features = self.X.shape
      epoch = 0

      while True:
         epoch += 1
         update = False

         for t in range(n_samples):
            y_hat = self._kernelized_predict(self.X[t])
            if self.y[t] != y_hat:
               self.S.add(t)
               update = True

         if (not update) or (epoch >= self.max_epochs):
            break

   def predict(self, X_test):
      return np.array([self._kernelized_predict(X) for X in X_test])

```

A.6 Kernelized Pegasos for SVM

```python

class KernelPegasosSVM:
   def __init__(
           self,
           kernel='polynomial',
           degree=3,
           gamma=1,
           lambda_param=0.01,
           T=1000
       ):
       self.kernel = kernel
       self.degree = degree
       self.gamma = gamma
       self.lambda_param = lambda_param
       self.T = T
       self.sgn = lambda z: np.where(z >= 0, 1, -1)

   def _K(self, x1, x2):
           if self.kernel == 'polynomial':
               return (1 + np.dot(x1, x2)) ** self.degree
           elif self.kernel == 'gaussian':
               return np.exp(-(np.linalg.norm(x1-x2, axis=1) ** 2) / (2*self.gamma))

   def _kernelized_predict(self, X_zt):
       return np.sum(self.alpha * self.y * self._K(self.X, X_zt))

   def train(self, X_train, y_train):
       self.X, self.y = X_train.copy(), y_train.copy()
       n_samples, n_features = self.X.shape
       self.alpha = np.zeros(n_samples)
       if self.T > n_samples: self.T = n_samples

       for t in range(self.T):
           eta_t = 1 / (self.lambda_param * (t+1))
           z_t = np.random.randint(0, n_samples)
           if self.y[z_t] * eta_t * self._kernelized_predict(self.X[z_t]) < 1:
               self.alpha[z_t] += 1

   def predict(self, X_test):
       return np.array([self.sgn(self._kernelized_predict(X)) for X in X_test])

```