Contents

1 Introduction

The primary goal of this project is to explore and implement several machine learning algorithms from scratch to classify labels based on numerical features. The performance of these models will be evaluated using a specific loss metric, ensuring a rigorous assessment of their effectiveness. The project will cover the following key objectives:

1. **Implementation of Basic Algorithms:** Develop and test the Perceptron, Support Vector Machines using the Pegasos algorithm, and regularized logistic classification.
2. **Feature Expansion:** Enhance model performance through polynomial feature expansion of degree 2 and compare the results with linear models.
3. **Kernel Methods:** Implement kernelized versions of the Perceptron and Pegasos algorithms using Gaussian and polynomial kernels.
4. **Hyperparameter Tuning:** Perform thorough hyperparameter tuning to optimize model performance.
5. **Evaluation and Analysis:** Evaluate the models' performance, discuss theoretical interpretations, and analyze computational costs, overfitting, and underfitting.

Implementing machine learning algorithms from scratch provides a deep understanding of their inner details, often missed with the widespread use of libraries like `scikit-learn`. Exploring different methods and their enhancements, such as polynomial feature expansion and kernel methods, offers insights into improving model performance. Finally, this kind of in-depth evaluation and probing into such models builds increased awareness of their pratical use and boundaries..

The paper is organized as follows:

- **Dataset Exploration and Preprocessing:** Discusses the dataset used, initial exploration, and preprocessing steps to prepare the data for modeling.
- **Implementation of Machine Learning Algorithms:** Details the theory, implementation, and performance of the Perceptron, SVMs using Pegasos, and regularized logistic classification.
- **Polynomial Feature Expansion:** Explores the impact of polynomial feature expansion on model performance.
- **Kernel Methods:** Describes the implementation and evaluation of kernelized Perceptron and Pegasos algorithms.

- **Hyperparameter Tuning:** Covers the methodology and results of hyperparameter tuning.
- **Conclusion:** Summarizes the findings and suggests future work.

2 Exploratory Data Analysis

In this section, a comprehensive exploratory data analysis is performed on the provided dataset to uncover significant patterns and relationships that can guide the subsequent modeling process.

2.1 Dataset Overview

The dataset comprises 10,000 observations and 10 features. The features are labeled as `x1` through `x10`, and the target variable is labeled as `y`. All features are numerical, and the target variable is binary, taking values of `-1` and `1`.

2.2 Data Cleaning

Before proceeding with the analysis, the dataset was checked for missing values and duplicates. Fortunately, the dataset contains neither missing values nor duplicates, indicating it is clean and ready for analysis.

2.3 Summary Statistics

Table 1 presents the summary statistics for the features in the dataset. Most features exhibit a broad range of values, with some having specific boundaries, such as `x5` and `x6`, which range from -1 to 1.

Table 1: Summary Statistics of the Dataset

Feature	Mean	Std Dev	Min	25th Pctl	50th Pctl	75th Pctl	Max
x1	1.591	1.321	0.002	0.525	1.276	2.352	9.384
x2	0.516	2.054	-7.525	-0.884	0.493	1.902	8.302
x3	99.849	0.711	98.572	99.159	99.803	100.549	101.261
x4	-1.504	1.134	-7.079	-2.180	-1.263	-0.605	-0.000
x5	0.078	0.707	-1.000	-0.625	0.157	0.778	1.000
x6	0.052	0.705	-1.000	-0.644	0.102	0.755	1.000
x7	0.975	2.162	-6.907	-0.501	1.007	2.435	8.760
x8	0.635	2.213	-7.141	-0.880	0.625	2.098	9.287
x9	0.052	1.770	-7.152	-1.130	0.025	1.249	6.211
x10	-55.448	0.710	-56.774	-56.138	-55.397	-54.758	-54.209
y	-0.002	1.000	-1.000	-1.000	-1.000	1.000	1.000

2.4 Correlation Analysis

The analysis of the correlation matrix reveals several notable relationships among the variables:

- Strong negative correlations were observed between x_3 and x_{10} ($r = -0.98$), as well as between x_3 and x_6 ($r = -0.99$). This suggests that x_3 might represent an inverse or opposing factor to both x_6 and x_{10}.
- A strong positive correlation exists between x_6 and x_{10} ($r = 0.99$), indicating that these variables likely measure closely related aspects of the phenomenon under study.
- Moderate positive correlations were found between x_1 and x_9 ($r = 0.43$), x_8 and y ($r = 0.39$), and x_7 and x_8 ($r = 0.33$).
- The dependent variable y shows its strongest positive correlations with x_8 ($r = 0.39$) and x_7 ($r = 0.26$), while exhibiting a notable negative correlation with x_4 ($r = -0.2$).
- Variables x_1 and x_2 display similar correlation patterns with other variables, suggesting a potential underlying relationship between them.
- The variable x_5 demonstrates generally weak correlations with most other variables, implying it may be less relevant or more independent within the dataset.
- Two main groups of correlated variables emerge: (1) x_3 (negatively correlated), x_6, and x_{10}; and (2) x_1, x_2, x_7, x_8, and x_9.

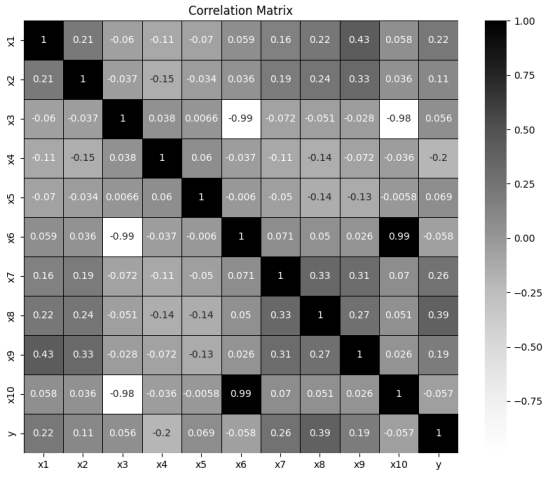

Figure 1: Correlation Matrix of the Variables

2.5 Distribution Analysis

Figure 4 illustrates the distribution of all numerical features.

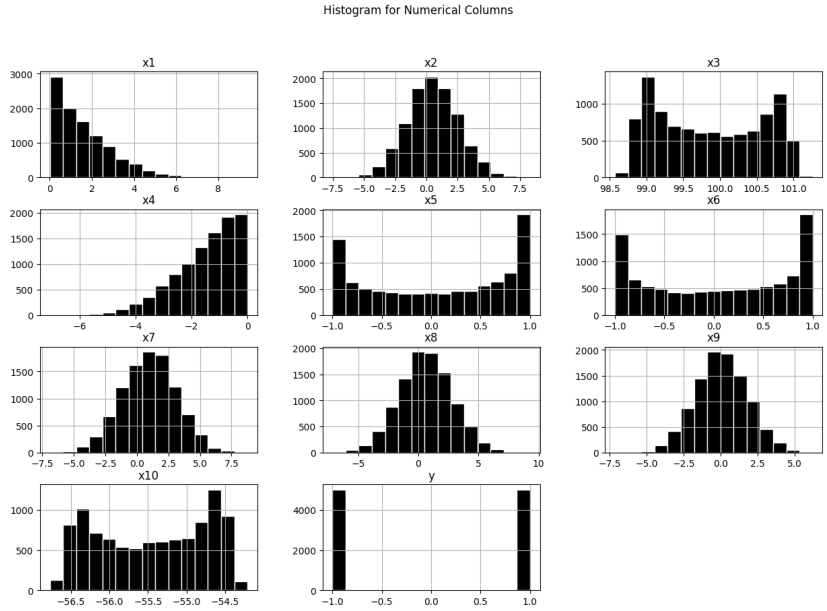

Figure 2: Histograms of Numerical Features

Examination of the histograms for each feature and the target variable reveals diverse distribution patterns:

- x_1: Exhibits a strong right-skewed distribution, with the majority of values concentrated near zero and a long tail extending to the right.
- x_2: Displays an approximately normal distribution, slightly skewed to the left, with most values centered around 0 to 2.5.
- x_3: Shows a bimodal distribution with peaks around 99 and 101, suggesting two distinct groups or states within this feature.
- x_4: Presents a left-skewed distribution, with most values clustered near 0 and a tail extending to negative values.
- x_5 and x_6: Both exhibit similar patterns with a central peak and significant tails on both sides, resembling a mixture of normal and uniform distributions.
- x_7, x_8, and x_9: All display approximately normal distributions centered around 0, with x_7 showing a slightly wider spread.

- x_{10}: Reveals a unique distribution with multiple peaks, suggesting a complex underlying structure or possibly discrete categories within a continuous range.
- y (target variable): Demonstrates a striking bimodal distribution with two distinct peaks at -1 and 1, and virtually no values in between. This suggests a binary classification problem rather than a continuous regression task.

Notable observations:

- The bimodal nature of y indicates that the problem at hand is likely a binary classification task, which may influence the choice of modeling techniques.
- The diverse distribution patterns across features suggest that feature scaling or transformation may be necessary for certain modeling approaches.
- The bimodal distribution of x_3 and the multi-modal nature of x_{10} might indicate underlying categorical variables or complex interactions that warrant further investigation.
- Features x_5 and x_6 show similar distributions, which, combined with their high correlation noted earlier, suggests they might represent related or redundant information.

2.6 Categorical Analysis

The target variable `y` is binary and well-balanced, with nearly equal representation of the classes `-1` and `1`, as depicted in Figure 3. This balance indicates that standard classification techniques can be applied without significant concern for class imbalance.

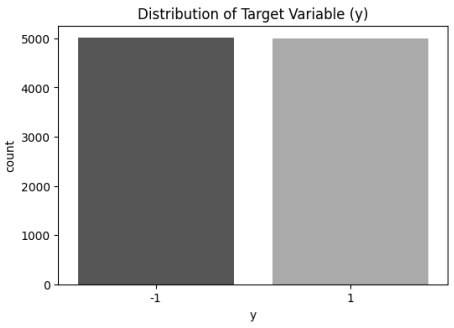

Figure 3: Distribution of Target Variable (`y`)

2.7 Boxplot Analysis and Outliers

The boxplot provides valuable insights into the distribution, central tendency, and presence of outliers for each feature:

- x_1: Shows a compact distribution with a positive skew and several outliers on the upper end.
- x_2: Displays a relatively symmetric distribution with outliers on both ends.
- x_3: Exhibits a narrow distribution centered around 100, with no visible outliers.
- x_4: Demonstrates a slightly negative skew with some outliers on the lower end.
- x_5 and x_6: Both show very narrow distributions, suggesting high consistency in these features.
- x_7, x_8, and x_9: Present similar, symmetric distributions centered around 0, with outliers on both ends.
- x_{10}: Shows a distribution significantly offset from the others, centered around -55.

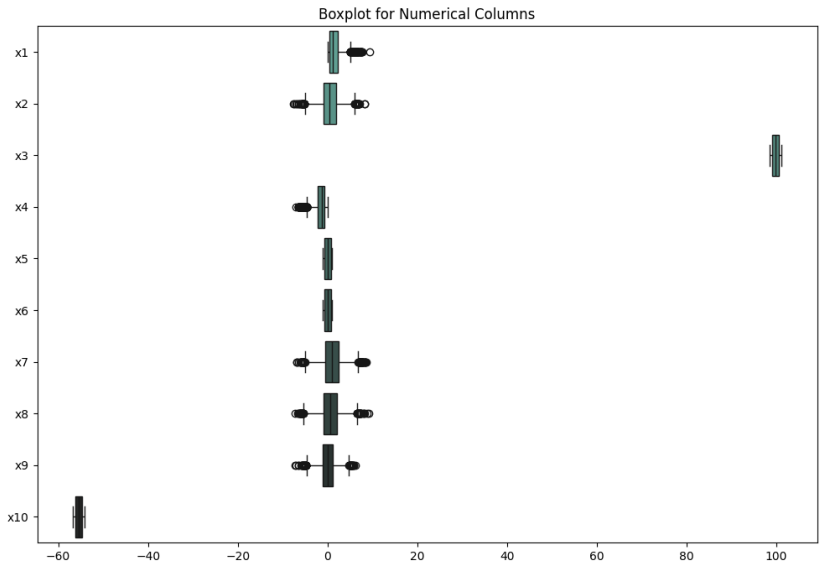

Figure 4: Histograms of Numerical Features

2.8 Data Preprocessing

Before developing the machine learning models, various preprocessing steps were performed to ensure that the dataset was properly ready for the training phase:

Outlier Removal Rows with extreme outliers were removed based on a z-score threshold of 3.3. This strategy allowed for the elimination of the most extreme outliers while retaining the majority of the dataset, thus avoiding a significant loss of information that could degrade model performance. The chosen z-score threshold was determined through optimization of the trade-off between retaining data integrity and the impact of outliers on model accuracy.

Data Standardization The features were standardized using the `StandardScaler()` function from `sklearn`. Standardization was necessary to ensure that all features contributed equally to the model training process. This step helped to normalize the range of the data, centering the mean at 0 and scaling the variance to 1, which is crucial for better convergence and performance of many machine learning algorithms.

Train-Test Split The dataset was split into training and testing sets using an 80/20 ratio. This conventional split ratio was chosen to provide a sufficiently large training set to build accurate models while reserving a meaningful portion of the data for unbiased evaluation. The 80/20 split ensures that the models can be effectively trained on a wide variety of examples while also being rigorously tested on unseen data, thereby allowing for a reliable assessment of their generalization capabilities.

3 Algorithm Implementation

In this section, we provide a comprehensive overview of the implementation of fundamental machine learning algorithms based on what the project description specified. We will focus on each individual algorithm, explaining its usage, functionality, and code implementation. Subsequently, we will evaluate the model's performance using a simple zero-one loss metric. This method will provide a clear and straightforward measure of the accuracy of our models by calculating the proportion of correct predictions made.

3.1 Perceptron

The Perceptron algorithm constitutes a foundational method employed in training linear classifiers that work well on linearly separable datasets. This algorithm aims to find a homogeneous separating hyperplane by iteratively adjusting the current linear classifier based on training examples. It iteratively adjusts the classifier based on misclassified examples, thus ensuring convergence for linearly separable datasets. It provides a simple yet powerful method for training linear

classifiers within a linearly separable context. Nevertheless, its performance can degrade significantly when faced with non-separable or nearly inseparable data, highlighting the need for alternative approaches in such scenarios.

Below is a detailed description of the Perceptron algorithm:

Algorithm 1 The Perceptron Algorithm pseudocode

Input: Training set $(x_1, y_1), \ldots, (x_m, y_m)$
Initialize: $w = (0, \ldots, 0)$
while true **do**
    for t = 1, . . . , m **do** ▷ (epoch)
        if $y_t w^\top x_t \leq 0$ **then**
            $w \leftarrow w + y_t x_t$ ▷ (update)
        end if
    end for
    if no update in the last epoch **then**
        break
    end if
end while
Output: w

3.1.1 Implementation Details

The Perceptron algorithm is implemented as a Python class named `Perceptron`. This class has two primary methods: `fit` and `predict`. The constructor method, `__init__`, initializes the number of iterations (`n_iterations`) the algorithm will run. The `weights` attribute, which represents the model's parameters, is initialized to `None` and is later set during the fitting process.

```
class Perceptron:
    def __init__(self, n_iterations=1000):
        self.n_iterations = n_iterations
        self.weights = None
```

The `fit` method is responsible for training the model. It initializes the `weights` vector to zeros, corresponding to the number of features in the dataset. The labels `y` are then transformed into a format where all non-positive values are replaced with `-1` to align with the Perceptron's binary classification requirement.

The algorithm iterates through the dataset for a specified number of iterations (`n_iterations`). During each iteration, it checks if the current weights misclassify any of the training examples. If a misclassification is found (i.e., the sign of the dot product between the feature vector and the weights is incorrect), the weights are updated by adding the product of the feature vector and its corresponding label. The algorithm tracks whether any updates are made during an iteration to determine if it has converged. If no updates occur, indicating that the model has correctly classified all training examples, the algorithm stops early.

```
def fit(self, X, y):
    n_samples, n_features = X.shape
    self.weights = np.zeros(n_features)

    y_ = np.where(y <= 0, -1, 1)

    for iteration in range(self.n_iterations):
        no_updates = True
        for idx in range(n_samples):
            if y_[idx] * np.dot(X[idx], self.weights) <= 0:
                self.weights += y_[idx] * X[idx]
                no_updates = False

        if no_updates:
            print(f"Converged after {iteration + 1} iterations")
            break
```

Once the Perceptron model has been trained, predictions can be made using the `predict` method. This method computes the dot product between the input features and the learned weights. The sign of this product determines the predicted class label for each example in the dataset.

```
def predict(self, X):
    linear_output = np.dot(X, self.weights)
    y_predicted = np.sign(linear_output)
    return y_predicted
```

3.1.2 Model Performance and Accuracy

After implementing the Perceptron algorithm, the model was trained on the given dataset using the training set. The following code was used to train the model and make predictions on both the training and test sets:

```
perceptron = Perceptron()
perceptron.fit(X_train, y_train)
y_train_pred = perceptron.predict(X_train)
y_test_pred = perceptron.predict(X_test)
```

To assess the performance of the model, a function was created that utilizes a zero-one loss, calculating the accuracy as the ratio of correctly predicted labels to the total number of predictions:

```
def calculate_accuracy(y_true, y_pred):
    correct_predictions = np.sum(y_true == y_pred)
    total_predictions = len(y_true)
    accuracy = correct_predictions / total_predictions
    return accuracy
```

This accuracy function was applied to both the training and test sets to evaluate the model's performance and detect any signs of overfitting or underfitting. The results are as follows:

- **Perceptron train accuracy:** 0.663
- **Perceptron test accuracy:** 0.651

The Perceptron model achieved a training accuracy of 66.26% and a test accuracy of 65.11%. The similarity between these values suggests that the model does not suffer from overfitting. Nevertheless, the somewhat low accuracies suggest the possibility of underfitting, suggesting that the model might be too basic to grasp the fundamental patterns in the data. This is likely due to the Perceptron's limitation as a linear classifier, which struggles with non-linear relationships.

In following sections of this paper, hyperparameter tuning and potentially more advanced methods, such as feature expansion and kernel methods, will be explored to improve the model's performance. These adjustments are anticipated to provide a clearer picture of the model's strenghts and limitations.

3.2 Support Vector Machines: Pegasos Algorithm

Pegasos is a stochastic gradient descent based algorithm specifically designed for efficiently solving the optimization problem associated with Support Vector Machines. In contrast to traditional batch gradient descent approaches that compute the gradient with the complete training set, Pegasos updates the model parameters by iteratively using only one example (or a few examples) randomly selected from the training set in each iteration.

This stochastic method is especially beneficial for handling large datasets, as it greatly decreases the computational complexity and memory needs of the training procedure. Pegasos is essentially an Online Gradient Descent algorithm, updating the weight vector incrementally with the gradient of the loss function computed from randomly chosen training instances.

he process works through a series of rounds T, during which a training example is randomly chosen each time to update the weight vector based on the sub-gradient of the hinge loss function for that example. Additionally, Pegasos incorporates a regularization term controlled by a regularization coefficient $\lambda > 0$, which helps prevent overfitting by penalizing large weight values.

The output of the algorithm is the average of the weight vectors obtained across all iterations, which provides a more stable and generalized model compared to using only the final weight vector.

3.2.1 Implementation Details

The Pegasos algorithm is implemented as a Python function named `pegasos_svm`. It begins with the initialization of two weight vectors: `w` and `w_avg`. The vector

`w` represents the current weight vector, while `w_avg` will store the average of all weight vectors across iterations. The parameters `T` (the total number of iterations) and `lam` (the regularization coefficient) are passed to the function along with the training data.

```
def pegasos_svm(X, y, T, lam):
    m, d = X.shape
    w = np.zeros(d)  # Initialize weights
    w_avg = np.zeros(d)  # Initialize average weights
```

The algorithm proceeds by iterating over the specified number of iterations (`T`). During each iteration, a random training example (x_t, y_t) is selected. The gradient of the hinge loss is computed based on whether the selected example is correctly classified or not. If the example is misclassified (i.e., $y_t \cdot (w \cdot x_t) < 1$), the gradient is calculated as $-y_t \cdot x_t$. Otherwise, the gradient is set to zero, indicating no adjustment is needed for that example.

```
for t in range(1, T + 1):
    # Step 1: Sample a random training example
    index = np.random.randint(m)
    x_t = X[index]
    y_t = y[index]

    # Step 2: Compute gradient of hinge loss
    if y_t * np.dot(x_t, w) < 1:
        grad = -y_t * x_t
    else:
        grad = 0
```

The learning rate (η) for each iteration is inversely proportional to the product of the regularization parameter (λ) and the iteration number (t). The weight vector `w` is updated by scaling it down (a regularization step) and then adding the scaled gradient. The average weight vector `w_avg` is updated by accumulating the current weight vector `w`.

```
    # Calculate learning rate (eta)
    eta = 1.0 / (lam * t)

    # Update weights
    w = (1 - eta * lam) * w + eta * grad

    # Accumulate average weights
    w_avg += w
```

Finally, after all iterations are complete, the average weight vector `w_avg` is computed by dividing the accumulated weights by the total number of iterations (`T`). This averaged weight vector is returned as the final model.

```
    # Compute average weights
    w_avg /= T

    return w_avg
```

To make predictions with the trained Pegasos model, the `predict` function is used. This function takes the input data and the learned weight vector, computes the dot product between each input vector and the weight vector, and returns the sign of the result. This sign determines the predicted class label.

```
# Prediction function
def predict(X, w):
    return np.sign(np.dot(X, w))
```

3.2.2 Model Performance and Accuracy

After implementing the Pegasos algorithm, the model was trained on the provided dataset using the training set. The following code was used to train the model and make predictions on both the training and test sets:

```
# Parameters
T = 1000000  # Number of iterations
lam = 0.01  # Regularization parameter

avg_weights = pegasos_svm(X_train, y_train, T, lam)

# Make predictions
y_pred_train = predict(X_train, avg_weights)
y_pred_test = predict(X_test, avg_weights)

# Evaluate the model
train_accuracy = calculate_accuracy(y_train, y_pred_train)
test_accuracy = calculate_accuracy(y_test, y_pred_test)

print(f"Train Accuracy: {train_accuracy * 100:.2f}%")
print(f"Test Accuracy: {test_accuracy * 100:.2f}%")
```

To assess the performance of the Pegasos SVM model, the accuracy was calculated using a zero-one loss approach, which measures the proportion of correctly classified examples. The accuracy function calculates the ratio of correctly predicted labels to the total number of predictions.

The results for the Pegasos model are as follows:

- **Pegasos train accuracy:** 42.62%
- **Pegasos test accuracy:** 42.20%

These results suggest that the model's performance is relatively weak. It is important to note that default hyperparameters were used, which are not optimized for this specific model. Further hyperparameter tuning will be necessary to identify the optimal values that could improve the model's performance. This process will be explored in subsequent sections to enhance the accuracy and overall effectiveness of the Pegasos algorithm.

3.3 Regularized Logistic Classification

Logistic classification models the probability that a binary response belongs to a particular class using the logistic function. To avoid overfitting, especially when dealing with high-dimensional data, a regularization term is added to the loss function. This term penalizes large coefficients, promoting simpler models.

The regularized logistic classification model is formulated as:

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

where λ is the regularization coefficient, x_t is the feature vector, y_t is the label, and w is the weight vector. The regularization term $\frac{\lambda}{2}\|w\|^2$ helps to prevent overfitting by penalizing large weights.

3.3.1 Implementation Details

The regularized logistic classification algorithm is implemented as a Python function named `regularized_logistic`. It begins with the initialization of two weight vectors: `w` and `w_avg`. The vector `w` represents the current weight vector, while `w_avg` will store the average of all weight vectors across iterations. The parameters `T` (the total number of iterations) and `lam` (the regularization coefficient) are passed to the function along with the training data.

```
def regularized_logistic(X, y, T, lam):
    m, d = X.shape
    w = np.zeros(d)  # Initialize weights
    w_avg = np.zeros(d)  # Initialize average weights
```

The algorithm proceeds by iterating over the specified number of iterations (`T`). During each iteration, a random training example (x_t, y_t) is selected. The gradient of the logistic loss is computed with a regularization term. The gradient is calculated using the sigmoid function, which transforms the linear prediction into a probability.

```
for t in range(1, T + 1):
    # Step 1: Sample a random training example
    index = np.random.randint(m)
    x_t = X[index]
    y_t = y.iloc[index]
```

```
    # Step 2: Compute gradient of logistic loss with regularization
    pred = np.dot(x_t, w)
    grad = -y_t * x_t * sigmoid(-y_t * pred) + lam * w
```

The learning rate (η) for each iteration is inversely proportional to the product of the regularization parameter (λ) and the iteration number (t). The weight vector `w` is updated by subtracting the scaled gradient. The average weight vector `w_avg` is updated by accumulating the current weight vector `w`.

```
    # Calculate learning rate (eta)
    eta = 1.0 / (lam * t)

    # Update weights
    w = w - eta * grad

    # Accumulate average weights
    w_avg += w
```

Finally, after all iterations are complete, the average weight vector `w_avg` is computed by dividing the accumulated weights by the total number of iterations (`T`). This averaged weight vector is returned as the final model.

```
    # Compute average weights
    w_avg /= T

    return w_avg
```

To make predictions with the trained logistic classification model, the `predict` function is used. This function takes the input data and the learned weight vector, computes the dot product between each input vector and the weight vector, applies the sigmoid function to obtain probabilities, and returns binary class predictions based on a specified threshold.

```
# Prediction function
def predict(X, w, threshold=0.5):
    probas = predict_proba(X, w)
    return np.where(probas >= threshold, 1, -1)
```

3.3.2 Model Performance and Accuracy

After implementing the regularized logistic classification algorithm, the model was trained on the provided dataset using the training set. The following code was used to train the model and make predictions on both the training and test sets:

```
# Step 1: Train the model on the training data
T = 10000  # Number of iterations
```

```
lam = 0.01  # Regularization parameter

# Train the model using the training data
w = regularized_logistic(X_train, y_train, T, lam)

# Step 2: Make predictions on both training and test data
train_predictions = predict(X_train, w)
test_predictions = predict(X_test, w)

# Step 3: Calculate accuracy on both training and test data
train_accuracy = calculate_accuracy(y_train, train_predictions)
test_accuracy = calculate_accuracy(y_test, test_predictions)

# Output the accuracies
print(f"Training Accuracy: {train_accuracy * 100:.2f}%")
print(f"Test Accuracy: {test_accuracy * 100:.2f}%")
```

The results for the regularized logistic classification model are as follows:

- **Training accuracy:** 72.42%
- **Test accuracy:** 71.40%

The regularized logistic classification model achieved an accuracy of 72.42% on the training set and 71.40% on the test set. These results suggest that the model performs reasonably well, indicating a good balance between fitting the training data and generalizing to unseen data. However, further hyperparameter tuning, such as adjusting the regularization parameter λ, could potentially improve the model's performance. This process will be explored in subsequent sections to enhance the accuracy and overall effectiveness of the logistic classification algorithm.

4 Polynomial Feature Expansion

Polynomial feature expansion is a technique used to increase the capacity of linear models by adding polynomial terms of the input features. This method allows the model to capture nonlinear relationships between features, potentially improving performance on tasks where such relationships are significant.

For instance, given an input vector $\mathbf{x} = (x_1, x_2, \ldots, x_n)$, a polynomial feature expansion of degree 2 would create new features such as $x_1^2, x_1x_2, \ldots, x_n^2$, effectively mapping the input vector into a higher-dimensional space. This transformation allows a linear model to learn more complex patterns in the data.

In this paper, we will implement a polynomial feature expansion of degree 2 and evaluate whether it improves the performance of our models on the classification task.

4.0.1 Implementation Details

The following code snippet implements a polynomial feature expansion of degree 2 from scratch:

```
# Polynomial Expansion
def polynomial_features(X, degree=2):
    n_samples, n_features = X.shape
    poly_features = [np.ones(n_samples)]  # Include bias term

    # Add original features
    for i in range(n_features):
        poly_features.append(X[:, i])

    # Add polynomial features of degree 2
    for i in range(n_features):
        for j in range(i, n_features):
            poly_features.append(X[:, i] * X[:, j])

    return np.vstack(poly_features).T
```

This function, `polynomial_features`, generates a polynomial feature expansion for a given dataset X. The function works as follows:

- **Initialization:** The function begins by determining the number of samples (n_{samples}) and the number of features (n_{features}) in the input dataset X. It initializes a list `poly_features` with a bias term, which is an array of ones corresponding to each sample in the dataset.
- **Original Features:** The function then iterates over each feature in X and appends it to the `poly_features` list. This step ensures that the original features are included in the expanded feature set.
- **Polynomial Features of Degree 2:** The function proceeds to generate polynomial features of degree 2. It does this by iterating over pairs of features, including interactions between each pair. For each pair of features (x_i, x_j), the function computes the product $x_i \times x_j$ and appends it to the `poly_features` list. This process results in new features that capture the interactions between different features.
- **Return the Expanded Feature Set:** Finally, the function converts the list of features into a NumPy array using `np.vstack(poly_features).T` and returns the expanded feature set. The result is a matrix where each row corresponds to a sample, and each column represents either an original feature, a bias term, or a polynomial feature of degree 2.

The polynomial feature expansion implemented in this code allows the model to learn more complex, nonlinear relationships by mapping the original features into a higher-dimensional space. This transformation can potentially improve model performance on tasks where such interactions are important.

4.0.2 Performance Comparison

After developing the feature expansion function, it was utilized to retrain the models previously discussed in earlier sections. The corresponding code is available within the files in the GitHub repository. Below, the differences in performance for each of the three models are presented:

Perceptron Model

Model	Train Accuracy	Test Accuracy
Perceptron	66.26%	65.11%
Perceptron with Polynomial Features	93.20%	93.00%

Table 2: Accuracy of Perceptron Model with and without Polynomial Features

Pegasos Model

Model	Train Accuracy	Test Accuracy
Pegasos	31.52%	33.52%
Pegasos with Polynomial Features	50.03%	50.30%

Table 3: Accuracy of Pegasos Model with and without Polynomial Features

Regularized Logistic Classification Model

Model	Train Accuracy	Test Accuracy
Regularized Logistic Classification	72.56%	71.40%
Regularized Logistic Classification with Polynomial Features	86.28%	86.45%

Table 4: Accuracy of Regularized Logistic Classification Model with and without Polynomial Features

The results presented in the tables above illustrate the substantial impact that polynomial feature expansion can have on model performance. For the Perceptron model, the training and test accuracies increased dramatically from approximately 65% to over 93%, demonstrating a significant enhancement in the model's capability to capture complex patterns in the data.

In the case of the Pegasos model, while the initial accuracies were relatively low at around 33%, the introduction of polynomial features improved the performance to approximately 50%. Although this is a modest improvement, it indicates that feature expansion can provide benefits even for models that initially perform poorly.

The Regularized Logistic Classification model also showed notable improvement, with training and test accuracies rising from approximately 72% to over 86%. This suggests that the model benefits from the additional complexity introduced by polynomial features, allowing it to better generalize to unseen data.

Overall, these results underscore the importance of feature engineering, particularly polynomial feature expansion, in enhancing the predictive power of machine learning models. These improvements highlight the potential for further exploration and tuning to optimize model performance.

5 Kernelized Models

In machine learning, a *kernel* is a function that computes a dot product in some (often high-dimensional) feature space without explicitly transforming the data into that space. This concept allows algorithms to operate in high-dimensional spaces efficiently, enabling them to learn more complex decision boundaries that are nonlinear in the original input space.

5.1 Kernelized Perceptron

The Kernelized Perceptron algorithm uses the *kernel trick*, which replaces the inner product between data points in the original space with a kernel function that computes the inner product in the feature-expanded space. This approach allows the Perceptron to learn nonlinear decision boundaries without explicitly performing the transformation to the high-dimensional feature space.

5.1.1 Algorithm

The Kernelized Perceptron updates its decision boundary using a kernel function $K(\mathbf{x}, \mathbf{x}')$ that implicitly maps the data into a higher-dimensional space. The decision function for the Kernelized Perceptron is given by:

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

where S is the set of indices of training examples on which the Perceptron made an update. The algorithm for the Kernelized Perceptron is as follows:

The use of the kernel function $K(\mathbf{x}, \mathbf{x}') = \phi(\mathbf{x}) \cdot \phi(\mathbf{x}')$, where ϕ maps the input into the high-dimensional feature space, allows the Perceptron to learn a more complex decision boundary that is nonlinear in the original input space. Popular kernel functions include the polynomial kernel and the Gaussian kernel, which can effectively handle a wide range of data distributions.

Algorithm 2 Kernelized Perceptron Algorithm

Let $S \leftarrow \emptyset$
for each training example $(\mathbf{x}_t, y_t)$ **do**
    Compute $\hat{y}_t \leftarrow \text{sgn}\left(\sum_{s \in S} y_s K(\mathbf{x}_s, \mathbf{x}_t)\right)$
    if $\hat{y}_t \neq y_t$ **then**
        Add t to S
    end if
end for

5.1.2 Implementation Details

The Kernelized Perceptron implementation in Python follows the algorithm described in the introduction. This implementation allows the use of different kernel functions, such as the Gaussian and polynomial kernels. Below is the Python code for the Kernelized Perceptron:

```
class KernelPerceptron:
    def __init__(self, kernel, max_iter=100):
        self.kernel = kernel
        self.max_iter = max_iter
        self.support_vectors = []
        self.support_labels = []

    def fit(self, X, y):
        S = set()
        for _ in range(self.max_iter):
            for t in range(len(X)):
                prediction = 0
                for idx, s in enumerate(S):
                    prediction += self.support_labels[idx] * self.kernel(self.support_vecto
                if (y.iloc[t] * prediction) <= 0:
                    S.add(len(self.support_vectors))
                    self.support_vectors.append(X[t])
                    self.support_labels.append(y.iloc[t])

    def predict(self, X):
        predictions = []
        for x in X:
            prediction = 0
            for s in range(len(self.support_vectors)):
                prediction += self.support_labels[s] * self.kernel(self.support_vectors[s],
            predictions.append(1 if prediction > 0 else -1)
        return predictions

def gaussian_kernel(x, y, gamma=1.0):
    squared_distance = sum((i - j) ** 2 for i, j in zip(x, y))
```

```
    return math.exp(-squared_distance / (2 * gamma))

def polynomial_kernel(x, y, degree=2):
    dot_product = sum(i * j for i, j in zip(x, y))
    return (1 + dot_product) ** degree
```

The `KernelPerceptron` class initializes with a specified kernel function and the maximum number of iterations (`max_iter`). During the fitting process, the algorithm iteratively updates its set of support vectors and labels whenever it misclassifies a training example. The decision function is computed using the provided kernel, which maps the input vectors into a higher-dimensional space.

The `predict` method computes predictions by summing the contributions of the support vectors, weighted by their labels and kernel evaluations. Two example kernel functions are implemented: the `gaussian_kernel` and the `polynomial_kernel`. The Gaussian kernel computes a similarity measure based on the squared distance between vectors, while the polynomial kernel computes a similarity measure based on the dot product of vectors, raised to a specified degree.

5.1.3 Model Performance

The Kernelized Perceptron was tested with both Gaussian and Polynomial kernels:

Gaussian Kernel

- **Training Accuracy:** 0.9977
- **Testing Accuracy:** 0.9332

Using the Gaussian kernel, the Kernelized Perceptron almost perfectly classifies the training set, indicating a very high capacity for fitting complex patterns in the data. However, the slight drop in testing accuracy compared to training accuracy suggests some degree of overfitting. The model performs exceptionally well but is potentially too tailored to the training data.

Polynomial Kernel

- **Training Accuracy:** 0.9366
- **Testing Accuracy:** 0.9291

With the Polynomial kernel, the Kernelized Perceptron shows performance comparable to the polynomial feature-expanded Perceptron, maintaining high accuracy on both training and testing sets. This balance indicates that the Polynomial kernel effectively captures the complexity of the data without significant overfitting.

Overall, the Kernelized Perceptron with Gaussian and Polynomial kernels demonstrates superior performance over the linear Perceptron. The Gaussian

kernel achieves the highest training accuracy, reflecting its capacity to model highly complex decision boundaries. However, the near-perfect training accuracy coupled with a lower test accuracy suggests mild overfitting. In contrast, the Polynomial kernel maintains high accuracy across both sets, indicating a well-balanced model that generalizes effectively.

These results highlight the strength of kernel methods in enhancing the flexibility of linear classifiers, allowing them to handle more complex datasets by implicitly mapping them into higher-dimensional feature spaces. Choosing the right kernel and regularization strategies is crucial to avoid overfitting and ensure robust model performance.

5.2 Kernelized Pegasos

Similar to the Kernelized Perceptron, the Kernelized Pegasos leverages the *kernel trick* to handle non-linearly separable data by implicitly mapping it into a higher-dimensional feature space.

Kernelized Pegasos is particularly well-suited for large-scale datasets due to its ability to perform efficient updates using only a randomly selected subset of training examples at each iteration. This stochastic approach reduces the computational burden compared to standard optimization techniques that require processing the entire dataset. By employing regularization, the algorithm strikes a balance between minimizing classification errors and controlling model complexity, thereby preventing overfitting through penalization of excessively large weights.

5.2.1 Algorithm

The Kernelized Pegasos algorithm iteratively updates the weights by selecting a random example from the dataset in each step and performing a stochastic gradient update. The detailed steps of the algorithm are as follows:

Algorithm 3 Kernelized Pegasos Algorithm

Input: Dataset S, regularization parameter λ, number of iterations T
Initialize: Set $\mathbf{w}_1 \leftarrow \mathbf{0}$
for $t = 1, 2, \ldots, T$ **do**
    Select $i_t \in \{1, \ldots, |S|\}$ uniformly at random.
    Set learning rate $\eta_t \leftarrow \frac{1}{\lambda t}$
    if $y_{i_t} \langle \mathbf{w}_t, \mathbf{x}_{i_t} \rangle < 1$ **then**
        Update $\mathbf{w}_{t+1} \leftarrow (1 - \eta_t \lambda)\mathbf{w}_t + \eta_t y_{i_t} \mathbf{x}_{i_t}$
    else
        Update $\mathbf{w}_{t+1} \leftarrow (1 - \eta_t \lambda)\mathbf{w}_t$
    end if
    [Optional: $\mathbf{w}_{t+1} \leftarrow \min\left\{1, \frac{1}{\sqrt{\lambda}\|\mathbf{w}_{t+1}\|}\right\} \mathbf{w}_{t+1}$]
end for
Output: $\mathbf{w}_{T+1}$

In this algorithm, the weights are updated based on the randomly chosen example and its dot product with the current weight vector, adjusted by the regularization term. If the dot product does not satisfy the margin condition (i.e., if $y_{i_t}\langle \mathbf{w}_t, \mathbf{x}_{i_t} \rangle < 1$), the algorithm adds a penalty term to improve the model's accuracy on the current example. Conversely, if the margin condition is satisfied, the weights are simply scaled by the regularization factor, thereby reducing model complexity.

5.2.2 Implementation Details

The implementation of the Kernelized Pegasos algorithm in Python involves two main variants: one using the Gaussian kernel and another using the Polynomial kernel. Below is the code for both variants along with a prediction function.

```
def kernelized_pegasos_gaussian(X_train, y_train, lambda_param=0.1, T=1000, gamma=1.0):
    n_samples = len(X_train)
    alphas = np.zeros(n_samples)

    for t in range(1, T + 1):
        i_t = np.random.randint(n_samples)
        x_it, y_it = X_train[i_t], y_train.iloc[i_t]
        eta_t = 1 / (lambda_param * t)

        kernel_sum = sum(alphas[j] * y_train.iloc[j] * gaussian_kernel(X_train[j], x_it, gam
                         for j in range(n_samples))

        if y_it * kernel_sum < 1:
            alphas[i_t] += eta_t

    return alphas

def kernelized_pegasos_polynomial(X_train, y_train, lambda_param=0.1, T=1000, degree=2):
    n_samples = len(X_train)
    alphas = np.zeros(n_samples)

    for t in range(1, T + 1):
        i_t = np.random.randint(n_samples)
        x_it, y_it = X_train[i_t], y_train.iloc[i_t]
        eta_t = 1 / (lambda_param * t)

        kernel_sum = sum(alphas[j] * y_train.iloc[j] * polynomial_kernel(X_train[j], x_it, d

        if y_it * kernel_sum < 1:
            alphas[i_t] += eta_t

    return alphas
```

The `kernelized_pegasos_gaussian` function implements the Kernelized Pegasos algorithm using a Gaussian kernel, while the `kernelized_pegasos_polynomial` function uses a Polynomial kernel. Both functions initialize an array of alphas, which serve as the dual coefficients for the support vectors, and iteratively update these coefficients based on a randomly selected training sample at each iteration.

The update rule adjusts the coefficients `alphas` if the margin condition ($y_{i_t} \times \text{kernel\_sum} < 1$) is not met, incrementing the corresponding alpha by the learning rate η_t. This approach helps the algorithm focus on examples that are either misclassified or within the margin, thus effectively refining the decision boundary.

The `predict` function utilizes the learned alphas and the specified kernel function to predict labels for a test set. It computes the weighted sum of kernel evaluations between each test point and the training data, weighted by the alphas and the corresponding training labels.

5.2.3 Model Performance

Kernelized Pegasos with Gaussian Kernel

- **Training Accuracy:** 76.25%
- **Testing Accuracy:** 74.68%

The Kernelized Pegasos with a Gaussian kernel shows a substantial improvement, achieving a training accuracy of 76.25% and a testing accuracy of 74.68%. This significant leap indicates that the Gaussian kernel effectively captures the complex, non-linear relationships within the data. However, the slight drop in testing accuracy compared to training accuracy suggests mild overfitting, where the model might be too closely fitted to the training data, although it still generalizes reasonably well.

Kernelized Pegasos with Polynomial Kernel

- **Training Accuracy:** 64.64%
- **Testing Accuracy:** 64.81%

Using a Polynomial kernel, the Kernelized Pegasos achieves a training accuracy of 64.64% and a testing accuracy of 64.81%. While this approach does not perform as well as the Gaussian kernel, it still significantly outperforms both the linear Pegasos and the Pegasos with polynomial feature expansion. The close values of training and testing accuracy suggest a well-balanced model that neither overfits nor underfits severely.

Overall, the Kernelized Pegasos demonstrates superior performance compared to the linear Pegasos, both with and without polynomial feature expansion. The Gaussian kernel, in particular, provides the best results, suggesting

its strength in capturing intricate data patterns. However, the mild discrepancy between training and testing accuracies points towards a slight tendency for overfitting, a common issue when using powerful kernels without adequate regularization.

On the other hand, the Polynomial kernel maintains a closer balance between training and testing performance, indicating effective generalization with reduced risk of overfitting. This balance could make the Polynomial kernel a more robust choice in scenarios where overfitting is a concern, despite not achieving the highest absolute accuracy.

6 Hyperparameter Tuning

The final step of this project involves hyperparameter tuning. Up to this point, models have been evaluated using only standard parameter settings. However, to optimize model performance, it is essential to determine the most suitable hyperparameter values that can maximize model effectiveness. The focus will be on models that utilize polynomial feature expansion, which have demonstrated both superior performance and reduced computational cost (compared to kernelized models, which require significantly more computational resources and are therefore more challenging to tune).

The subsequent sections will outline the code used to implement hyperparameter tuning from scratch for each model, evaluate performance differences, and assess how hyperparameter optimization can enhance the overall results of the study.

6.1 Perceptron

The hyperparameter tuning for the Perceptron model was conducted to determine the optimal number of iterations. The initial range for the number of iterations considered was {10, 50, 100, 500, 1000, 2000}. The following code was used for this purpose:

```
# Hyperparameter Tuning: Perceptron

n_iterations_list = [10, 50, 100, 500, 1000, 2000]

best_score = 0
best_n_iterations = 0

for n_iterations in n_iterations_list:
    perceptron = PerceptronPoly(n_iterations=n_iterations)
    perceptron.fit(X_train, y_train)
    y_pred = perceptron.predict(X_test)
    score = accuracy_score(y_test, y_pred)
    print(f"Accuracy for n_iterations={n_iterations}: {score}")
```

```
    if score > best_score:
        best_score = score
        best_n_iterations = n_iterations

print(f"Best n_iterations: {best_n_iterations} with accuracy: {best_score}")
```

After evaluating the model with the initial set of iterations, the search was further refined to explore a narrower range. The subsequent range considered was {1000, 1100, 1200, 1300, 1400, 1500} to determine if a more precise optimization could be achieved. The refined tuning code used was:

```
# Refined Hyperparameter Tuning: Perceptron

n_iterations_list = list(range(1000,1501, 100))

best_score = 0
best_n_iterations = 0

for n_iterations in n_iterations_list:
    perceptron = PerceptronPoly(n_iterations=n_iterations)
    perceptron.fit(X_train, y_train)
    y_pred = perceptron.predict(X_test)
    score = accuracy_score(y_test, y_pred)
    print(f"Accuracy for n_iterations={n_iterations}: {score}")

    if score > best_score:
        best_score = score
        best_n_iterations = n_iterations

print(f"Best n_iterations: {best_n_iterations} with accuracy: {best_score}")
```

The final result indicated that the optimal number of iterations was 1000, with an accuracy of 93.37%. This accuracy remained consistent with the previous results, suggesting that the default number of iterations used initially was already well-optimized for this model. Thus, further fine-tuning within the narrower range did not yield a better result.

6.2 Pegasos

In the Pegasos Support Vector Machine model, two key hyperparameters need to be optimized: the number of iterations T and the regularization parameter λ. The initial values for these hyperparameters were set to $T = 1000$ and $\lambda = 0.01$, which resulted in an accuracy of 33.32%. However, a grid search was performed to find more optimal values for these parameters.

The following code was used to explore different combinations of T and λ:

```
# Hyperparameter Tuning: PEGASOS

T_list = list(range(1, 501, 1))
lam_list = [0.001, 0.01, 0.1, 1.0]

best_score = 0
best_params = {}

for T in T_list:
    for lam in lam_list:
        w_avg = pegasos_svm_poly(X_train, y_train, T, lam)
        y_pred = predict(X_test, w_avg)
        score = calculate_accuracy(y_test, y_pred)
        print(f"Accuracy for T={T}, lam={lam}: {score}")

        if score > best_score:
            best_score = score
            best_params = {'T': T, 'lam': lam}

print(f"Best parameters: {best_params} with accuracy: {best_score}")
```

The search was conducted across T values from 1 to 500, specifically {1, 2, 3, ..., 500}, and λ values of {0.001, 0.01, 0.1, 1.0}. This extensive search led to the identification of the optimal hyperparameters: $T = 1$ and $\lambda = 0.01$, which produced an improved accuracy of 50.73%, hence substantially enhancing the model's performance.

6.3 Regularized Logistic Classification

The Regularized Logistic Classification model requires the tuning of two key hyperparameters: the number of iterations T and the regularization parameter λ. Initially, the model without hyperparameter tuning achieved a training accuracy of 72.42% and a test accuracy of 71.40% when using polynomial features. To improve these results, a wide range of values for both T and λ was explored. The ranges considered were $T = \{100, 500, 1000, 5000, 10000\}$ and $\lambda = \{0.001, 0.01, 0.1, 1.0\}$.

The following code was used to perform this initial search:

```
# Hyperparameter Tuning: Regularized Logistic Classification

T_list = [100, 500, 1000, 5000, 10000]
lambda_list = [0.001, 0.01, 0.1, 1.0]

best_score = 0
best_params = {}
```

```
for T in T_list:
    for lambda_ in lambda_list:
        w = regularized_logistic_classification_poly(X_train, y_train, T, lambda_)
        y_pred = predict(X_test, w)
        score = calculate_accuracy(y_test, y_pred)
        print(f"Accuracy for T={T}, lambda_={lambda_}: {score}")

        if score > best_score:
            best_score = score
            best_params = {'T': T, 'lambda_': lambda_}

print(f"Best parameters: {best_params} with accuracy: {best_score}")
```

From this initial exploration, the optimal parameters were found to be $T = 5000$ and $\lambda = 0.01$, yielding an accuracy of 92%. To refine these results, a narrower range of T values was tested, specifically $T = \{5000, 6000, 7000, 8000, 9000, 10000\}$, while maintaining the same range for λ.

The refined tuning process was implemented as follows:

```
# Hyperparameter Tuning: Regularized Logistic Classification

T_list = list(range(5000,10001,1000))
lambda_list = [0.001, 0.01, 0.1, 1.0]

best_score = 0
best_params = {}

for T in T_list:
    for lambda_ in lambda_list:
        w = regularized_logistic_classification_poly(X_train, y_train, T, lambda_)
        y_pred = predict(X_test, w)
        score = calculate_accuracy(y_test, y_pred)
        print(f"Accuracy for T={T}, lambda_={lambda_}: {score}")

        if score > best_score:
            best_score = score
            best_params = {'T': T, 'lambda_': lambda_}

print(f"Best parameters: {best_params} with accuracy: {best_score}")
```

This second round of tuning resulted in a slight adjustment, with the best parameters being $T = 7000$ and $\lambda = 0.01$, achieving an accuracy of approximately 91.14%. To further hone the optimization, the search was limited to T values in the range $\{7000, 7100, \ldots, 8000\}$, as shown in the following code:

```
# Hyperparameter Tuning: Regularized Logistic Classification
```

```
T_list = list(range(7000, 8001, 100))
lambda_list = [0.001, 0.01, 0.1, 1.0]

best_score = 0
best_params = {}

for T in T_list:
    for lambda_ in lambda_list:
        w = regularized_logistic_classification_poly(X_train, y_train, T, lambda_)
        y_pred = predict(X_test, w)
        score = calculate_accuracy(y_test, y_pred)
        print(f"Accuracy for T={T}, lambda_={lambda_}: {score}")

        if score > best_score:
            best_score = score
            best_params = {'T': T, 'lambda_': lambda_}

print(f"Best parameters: {best_params} with accuracy: {best_score}")
```

After this final round of tuning, the optimal parameters were confirmed as $T = 7800$ and $\lambda = 0.01$, with an accuracy of 92%. Compared to the initial accuracy of 71.40% without hyperparameter tuning, the tuning process significantly improved the model's performance. The additional fine-tuning in the narrower range confirmed that these parameters were indeed near-optimal.

7 Conclusion

Among the models evaluated, the Kernelized Perceptron with Gaussian Kernel demonstrated the best performance for the classification task, achieving an accuracy of approximately 99% on the training set and about 93% on the test set. This result suggests that the model effectively captured complex nonlinear relationships in the data, significantly outperforming the basic linear models. However, the slight drop in accuracy between the training and test sets indicates potential overfitting, suggesting that the model is highly tailored to the training data.

Other models, such as the Perceptron with Polynomial Feature Expansion and the Regularized Logistic Classification with Polynomial Feature Expansion, also showed improved performance due to feature expansion, with accuracies reaching around 85% to 93%. These results demonstrate that adding complexity through polynomial features can significantly enhance predictive capabilities compared to simple linear versions.

Potential improvements to the models could include:

- **Cross-Validation:** Utilizing cross-validation to evaluate model performance more robustly. This approach would help mitigate the risk of over-

fitting and provide a more accurate estimate of model performance on unseen data.

- **Further Hyperparameter Tuning:** Conducting a more extensive hyperparameter search, exploring a wider range of values and potential combinations, could further optimize performance.
- **Advanced Feature Engineering:** Introducing more advanced feature engineering techniques, such as automated feature selection or dimensionality reduction methods, could improve the model's ability to generalize to new data.

These improvements offer opportunities to further optimize the models, addressing current limitations and enhancing their predictive capabilities for future classification applications.

GitHub Repository: https://github.com/AntonioWolf01/machine-learning-dse

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