NEURAL NETWORKS - 2nd PROJECT REPORT

Victoria Galanopoulou 10630

November 2024

1 Introduction

In this project, I implemented a Support Vector Machine (SVM) for binary classification using the CIFAR-10 dataset. The primary objective was to analyze the performance of the SVM model under various configurations and compare it with other classification algorithms.

This report is structured as follows:

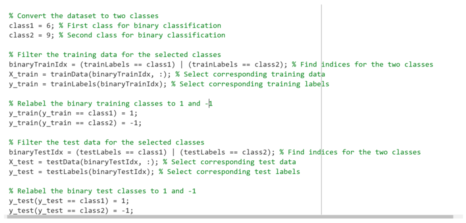
- I will describe the steps taken during training and present the code implementation.
- I will analyze the impact of key model parameters on accuracy and discuss the observed results.
- Finally, I will compare the performance of the SVM with a Multi-Layer Perceptron (MLP) neural network with one hidden layer using hinge loss, as well as the Nearest Neighbor(k-NN) and Nearest Centroid algorithms.

2 Data Handling for Binary Classification

In this project, we aim to perform binary classification using a subset of the CIFAR-10 dataset. Since CIFAR-10 is a multi-class dataset with ten classes, we filter the data to retain only two classes for simplicity and to align with the binary classification objective.

The data preprocessing involves extracting samples belonging to the two selected classes from both the training and test datasets, by keeping only the data whose label belongs to the chosen ones. The labels are then relabeled to match the binary classification framework: one class is assigned a label of **1** and the other a label of **-1**. This ensures the model focuses solely on distinguishing between these two classes.

By reducing the dataset to a binary task, we simplify the problem and make it easier to evaluate the performance of methods such as SVM and MLP. This also allows us to analyze the behavior of the models when dealing with pairs of classes that may vary in difficulty, such as visually similar classes (*e.g., cats and dogs*) or distinct classes (*e.g., trucks and frogs*).



```
% Convert the dataset to two classes
class1 = 6; % First class for binary classification
class2 = 9; % Second class for binary classification

% Filter the training data for the selected classes
binaryTrainIdx = (trainLabels == class1) | (trainLabels == class2); % Find indices for the two classes
X_train = trainData(binaryTrainIdx, :); % Select corresponding training data
y_train = trainLabels(binaryTrainIdx); % Select corresponding training labels

% Relabel the binary training classes to 1 and -1
y_train(y_train == class1) = 1;
y_train(y_train == class2) = -1;

% Filter the test data for the selected classes
binaryTestIdx = (testLabels == class1) | (testLabels == class2); % Find indices for the two classes
X_test = testData(binaryTestIdx, :); % Select corresponding test data
y_test = testLabels(binaryTestIdx); % Select corresponding test labels

% Relabel the binary test classes to 1 and -1
y_test(y_test == class1) = 1;
y_test(y_test == class2) = -1;
```

Figure 1: Processing datasets code

3 Mathematical Background of Support Vector Machines (SVMs)

SVMs are supervised learning algorithms used for classification and regression tasks. The primary goal of an SVM is to find the optimal hyperplane that separates data points of different classes with the largest margin. For a binary classification problem, given a dataset $\{(\mathbf{x}_i, y_i)\}_{i=1}^n$, where $\mathbf{x}_i \in R^d$ are input features and $y_i \in \{-1, 1\}$ are labels, the decision function is defined as:

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

where $\mathbf{w}$ is the weight vector and b is the bias term. The SVM optimizes the following objective:

$$\min_{\mathbf{w},b,\xi} \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^n \xi_i,$$

subject to:

$$y_i(\mathbf{w}^\top \mathbf{x}_i + b) \geq 1 - \xi_i, \quad \xi_i \geq 0, \quad \forall i.$$

Here, ξ_i are slack variables that allow for soft margin classification, and C is the regularization parameter that balances the trade-off between maximizing the margin and minimizing classification errors.

If we were to optimize the problem in terms of α (the Lagrange multipliers), instead of directly using ξ (slack variables), the optimization formulation for the dual problem would be expressed as:

$$\max_{\alpha} \sum_{i=1}^n \alpha_i - \frac{1}{2}\sum_{i=1}^n \sum_{j=1}^n \alpha_i \alpha_j y_i y_j \mathbf{x}_i^\top \mathbf{x}_j$$

Subject to:

$$\sum_{i=1}^n \alpha_i y_i = 0, \quad 0 \leq \alpha_i \leq C, \quad \forall i.$$

1. The goal is to maximize the dual objective function, which incorporates the pairwise dot products of the feature vectors ($\mathbf{x}_i^\top \mathbf{x}_j$) weighted by the Lagrange multipliers α_i and α_j, and the class labels y_i and y_j.

2. $\sum_{i=1}^n \alpha_i y_i = 0$: This ensures that the weighted sum of the Lagrange multipliers aligns with the margin constraints. - $0 \leq \alpha_i \leq C$: The multipliers are bounded by the regularization parameter C, controlling the influence of each data point.

The weights $\mathbf{w}$ in the SVM are computed as a weighted sum of the support vectors, where the Lagrange multipliers α_i and the corresponding class labels y_i scale the input data $\mathbf{x}_i$. Mathematically, the weights are given by:

$$\mathbf{w} = \sum_{i \in S} \alpha_i y_i \mathbf{x}_i,$$

where S is the set of support vectors (those with $\alpha_i > 0$).

The bias b is calculated by averaging the difference between the true labels y_i of the support vectors and the decision function $\mathbf{w}^\top \mathbf{x}_i$. This is expressed as:

$$b = \frac{1}{|S|} \sum_{i \in S} \left(y_i - \mathbf{w}^\top \mathbf{x}_i \right),$$

where $|S|$ is the number of support vectors.

4 SVM from scratch code

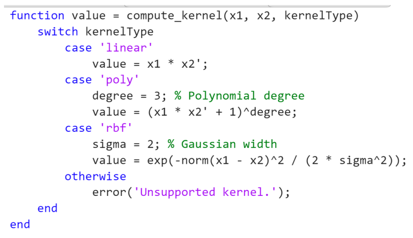
To undesrtand better the SVM theory, I implemented an SVM from scratch, focusing on binary classification and using as an example the provided code that we had. The main components include data preprocessing, kernel computation, training, and evaluation.

Data Preprocessing: The code begins by normalizing the CIFAR-10 dataset to ensure pixel values lie in the range $[0, 1]$. It selects two specific classes for binary classification, which are relabeled as $+1$ and -1. This step is handled by the `preprocess_labels` function, which assigns $+1$ to the target class and -1 to the other class. The processed data is then divided into training and test sets for model evaluation.

Kernel Computation: The kernel function, implemented in `compute_kernel`, maps data into a higher-dimensional space to handle non-linear separability. Three kernel types are supported:

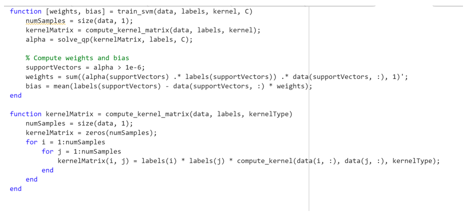
- **Linear:** $K(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{x}_i^\top \mathbf{x}_j$.
- **Polynomial:** $K(\mathbf{x}_i, \mathbf{x}_j) = (\mathbf{x}_i^\top \mathbf{x}_j + 1)^d$, where d is the polynomial degree.
- **RBF:** $K(\mathbf{x}_i, \mathbf{x}_j) = \exp\left(-\frac{\|\mathbf{x}_i - \mathbf{x}_j\|^2}{2\sigma^2}\right)$, where σ controls the width of the Gaussian.

Training the SVM: The training process is performed by solving a quadratic programming (QP) problem, where the dual objective of the SVM is optimized. The function `train_svm` computes the kernel matrix, solves the QP problem to obtain the Lagrange multipliers (α), and calculates the weights ($\mathbf{w}$) and bias (b) using the support vectors.



```
function value = compute_kernel(x1, x2, kernelType)
    switch kernelType
        case 'linear'
            value = x1 * x2';
        case 'poly'
            degree = 3; % Polynomial degree
            value = (x1 * x2' + 1)^degree;
        case 'rbf'
            sigma = 2; % Gaussian width
            value = exp(-norm(x1 - x2)^2 / (2 * sigma^2));
        otherwise
            error('Unsupported kernel.');
    end
end
```

Figure 2: Different Kernels



```
function [weights, bias] = train_svm(data, labels, kernel, C)
    numSamples = size(data, 1);
    kernelMatrix = compute_kernel_matrix(data, labels, kernel);
    alpha = solve_qp(kernelMatrix, labels, C);

    % Compute weights and bias
    supportVectors = alpha > 1e-6;
    weights = sum((alpha(supportVectors) .* labels(supportVectors)) .* data(supportVectors, :), 1)';
    bias = mean(labels(supportVectors) - data(supportVectors, :) * weights);
end

function kernelMatrix = compute_kernel_matrix(data, labels, kernelType)
    numSamples = size(data, 1);
    kernelMatrix = zeros(numSamples);
    for i = 1:numSamples
        for j = 1:numSamples
            kernelMatrix(i, j) = labels(i) * labels(j) * compute_kernel(data(i, :), data(j, :), kernelType);
        end
    end
end
```

Figure 3: SVM basic structure

Prediction and Evaluation: The function `predict_labels` predicts the class labels for test data by evaluating the decision function $f(\mathbf{x}) = \mathbf{w}^\top \mathbf{x} + b$. The accuracy is calculated as the proportion of correctly predicted labels. The overall error rate is computed using the `compute_error` function, which compares the predicted labels with the real ones.

5 SVM Implementation Using MATLAB Functions

To simplify the experiments, I utilized MATLAB's built-in functions, such as `fitcsvm`, to compare the performance of my custom implementation against the results obtained from these ready-made functions.

The `fitcsvm` function in MATLAB trains a Support Vector Machine (SVM) classifier by solving a quadratic optimization problem to find the optimal hyperplane separating the classes. Key parameters include `KernelFunction`, which specifies the kernel type (e.g., linear, polynomial, RBF), and `BoxConstraint`, which balances the trade-off between maximizing the margin and minimizing misclassification errors.

Below, I present the results obtained using a single batch from the CIFAR-10 training dataset, which contains 10,000 samples. For the experiments, a linear kernel was used with regularization parameter $C = 0.01$. The results focus on binary classification for two class pairs: *cat–dog (3–5)* and *frog–truck (6–9)*.

The table highlights the train and test accuracies for both implementations.

The results show that both implementations produce identical accuracies, which validates the correctness of my custom implementation. However, it is worth mentioning that my implementation is significantly slower compared to MATLAB's optimized functions. This difference in execution time underscores the efficiency of MATLAB's built-in libraries, which are optimized for speed and scalability. So, for this reason I used the built-in functions for the rest of the analysis.

Table 1: Compparison of the two codes

Pair	**Train Accuracy my code (%)**	**Test Accuracy my code (%)**	**Train Accuracy (%)**	**Test Accuracy (%)**
cat – dog (3 – 5	69.40	59.95	62.57	59.00
frog – truck (6 – 9)	91.60	86.60	89.91	87.20

6 PCA

Principal Component Analysis (PCA) is a dimensionality reduction technique that transforms high-dimensional data into a lower-dimensional space while preserving as much variance as possible. PCA works by computing the eigenvectors and eigenvalues of the data's covariance matrix, identifying the directions (principal components) that capture the most significant variance. The data is then projected onto these principal components, reducing redundancy and simplifying the input features. This reduction helps in improving computational efficiency and mitigating the risk of overfitting in machine learning models, especially when dealing with high-dimensional datasets. PCA is particularly useful as a preprocessing step for algorithms like SVM, where it helps retain meaningful information while discarding noise and irrelevant features.

The use of PCA can often improve test accuracy, especially when dealing with high-dimensional data. By reducing the number of features while retaining the most significant variance, PCA helps remove redundant or noisy features that may lead to overfitting. For SVMs, which can be sensitive to the curse of dimensionality, PCA simplifies the input space, allowing the model to generalize better to unseen data. However, the actual improvement in test accuracy depends on the dataset and the level of dimensionality reduction. In some cases, excessive reduction of dimensions may lead to a loss of critical information, resulting in a decrease in accuracy. Therefore, selecting the optimal number of principal components is crucial to balancing information reception and model generalization.

Table 2: SVM Binary Classification Results with a Linear Kernel

Pair	**Train Accuracy PCA (%)**	**Test Accuracy PCA (%)**	**Train Accuracy (%)**	**Test Accuracy (%)**
airplane – bird (0 – 2)	80.71	78.90	89.41	65.25
cat – dog (3 – 5)	62.57	59.00	81.87	56.85
frog – truck (6 – 9)	89.91	87.20	97.27	81.90
automobile – truck (1 – 9)	75.29	69.40	89.41	65.25

It has to be mentioned that those results came from testing only 10.000

samples, beacause using the whole dataset without PCA takes excessively long time.

The results in the table demonstrate the effect of PCA on training and test accuracies for binary classification using a linear SVM kernel. Without PCA, the model achieves higher training accuracies (e.g., 97.27% for *frog – truck* and 89.41% for *automobile – truck*), but at the cost of overfitting, as evidenced by the relatively lower test accuracies (81.90% and 65.25%, respectively). This indicates that the SVM struggles to generalize well when the feature space remains high-dimensional.

In contrast, when PCA is applied to retain 90% of the variance, the model shows more balanced results with significantly reduced overfitting. For example, the training accuracy for *frog – truck* drops to 89.91%, but the test accuracy improves to 87.20%, demonstrating better generalization. Similar trends can be observed for the other class pairs, where the test accuracies with PCA are consistently closer to the training accuracies.

Additionally, PCA reduces the computational complexity by lowering the number of features, leading to faster training and testing times. Testing with the whole sample leads to a really slow training, even wit linear kernel, that is the most simple one. When I tested with the whole training sample, 50000 pictures, with linear kernel for cat-dog pair , I had same accuracies, but the execution time was 3 times bigger than the SVM model with PCA. All in all, PCA is a really useful tecnique for such big and complex data sets, like CIFAR-10.

7 Binary Classification - Comparison of different pairs

For binary classification, I randomly selected two classes from the CIFAR-10 dataset and evaluated the SVM's performance. As expected, the accuracy was lower when the two classes shared similar visual characteristics, such as *dog* and *cat*, highlighting the challenge of distinguishing between closely related classes. On the other hand, when the selected classes were visually distinct, such as *truck* and *bird*, the accuracy approached near-perfect levels, demonstrating the SVM's ability to effectively separate well-separated classes. Using a linear kernel and $c = 0.01$, for one batch of the training and test sets, I obtained the following results:

Observations:

- **Similar Classes:** The pairs *cat – dog* $(3 - 5)$ achieved lower accuracy, with test accuracy of 59.00% , respectively. This can be attributed to their similar visual features, such as body structure and texture, which makes it harder for the SVM to draw a clear decision boundary.

- **Distinct Classes:** In contrast, visually distinct pairs such as *frog – truck* $(6 - 9)$ and *airplane – bird* $(0 - 2)$ achieved much higher accuracies, with

Table 3: SVM Binary Classification Results with a Linear Kernel

Pair	**Training Accuracy (%)**	**Test Accuracy (%)**
airplane – bird (0 – 2)	80.71	78.90
cat – bird (3 – 2)	75.59	69.70
cat – dog (3 – 5)	62.57	59.00
frog – truck (6 – 9)	89.91	87.20
automobile – truck (1 – 9)	75.29	69.40
deer – dog (4 – 5)	74.69	69.70
dog – frog (5 – 6)	77.48	74.05

test accuracies of 87.20% and 78.90%, respectively. These results confirm the SVM's capability to effectively separate classes with minimal feature overlap.

- **Edge Cases:** While *automobile – truck* (1 – 9) and *deer – dog* (4 – 5) achieved moderate results (69.40%) and (69.70%) , this performance can be explained by their shared structural features (e.g., wheels, rectangular shape), which may cause misclassifications.

- **Training vs. Test Accuracy:** Overall, the training accuracies are consistently higher than the test accuracies, indicating good generalization with minimal overfitting. For example, in the *frog – truck* pair, the test accuracy (87.20%) remains close to the training accuracy (89.91%).

These observations underline the influence of class similarity on the SVM's performance. While the SVM excels in separating distinct classes, its performance is limited when distinguishing between visually similar classes, as expected for a linear kernel.

8 Analysis of Results with Different Dataset Sizes

The results in the table below illustrate the influence of dataset size on the performance of the SVM with a linear kernel for binary classification tasks involving *cat–dog (3–5)* and *frog–truck (6–9)*. For the *cat–dog* pair, as the dataset size increases from *8,000–2,000* to *50,000–10,000* samples, the test accuracy improves significantly, reaching 61.40% with the full dataset. This highlights the advantage of larger and more diverse datasets in improving generalization for challenging pairs of visually similar classes. Similarly, for the *frog–truck* pair, the test accuracy remains consistently high, ranging between 86.40% and 88.15%, reflecting the ease of separating distinct classes even with smaller datasets.

A noticeable trend is the reduction in the difference between training and test accuracy as the dataset size increases. For example, in the *cat–dog* pair, the gap between training and test accuracy is larger with smaller datasets (approximately 7% for *8,000–2,000*), but this gap reduces significantly to around **2%** with *50,000–10,000* samples. This indicates that larger datasets help reduce overfitting, allowing the model to perform more consistently on unseen data. For the *frog–truck* pair, the training and test accuracies are already closely aligned, suggesting that distinct classes are less prone to overfitting, even with smaller training datasets.

The execution time is augmented when the dataset - number of samples is increased, as expected. The faster execution time observed for the pair 3–5 is likely due to the smaller number of samples in the selected data for this pair compared to the pair 6–9.

Table 4: SVM Binary Classification Results with a Linear Kernel

Pair	**Training Accuracy (%)**	**Test Accuracy (%)**	**Exec. Time**
8000 - 2000 samples (3 – 5)	62.45	55.21	2.91 sec
10.000 – 10.0000 samples (3 – 5)	62.57	59.00	5.19 sec
50.000 – 10.000 samples (3 – 5)	61.57	61.40	27.92 sec
8000 – 2000 samples (6 – 9)	90.58	86.40	3.41 sec
10.000 – 10.0000 samples (6 – 9)	89.91	87.20	5.33 sec
50.000 - 10.0000 samples (6 – 9)	90.24	88.15	27.62 sec

9 Comparison of different Hyperparameters

9.1 Linear Kernel - Different c

The linear kernel assumes that the data is linearly separable (or approximately so), making it a suitable choice for datasets with limited non-linear complexity. The parameter C controls the trade-off between achieving a wider margin and minimizing the classification error. Smaller values of C encourage larger margins by tolerating some misclassified points, which reduces overfitting and improves generalization. Conversely, larger values of C focus on minimizing training error, potentially leading to overfitting.

The results in Table 5 demonstrate how varying C affects the training and test accuracies.

Table 5: Training and Test Accuracies for Different C Values Using a Linear Kernel

C value	**Training Accuracy (%)**	**Test Accuracy (%)**
0.001	89.11	86.50
0.01	89.91	87.20
0.1	90.65	86.65
1	90.85	86.75
10	90.70	86.65
100	88.07	84.30

Analysis of Results:

- **Performance at $C = 0.001$:** With a very small C, the test accuracy drops to 86.50%, despite a relatively high training accuracy of 89.11%. This occurs because an excessively small C prioritizes a very wide margin, which may overly tolerate misclassified points and underfit the data. As a result, the model fails to capture the decision boundaries effectively, leading to a slight decrease in accuracy.
- **Best Performance at $C = 0.01$:** The highest test accuracy (87.20%) is achieved at $C = 0.01$. This value provides an optimal balance between generalization and model fit, resulting in a wide margin that minimizes overfitting without excessively misclassifying points in the training data.
- **Effect of Increasing C**: As C increases from 0.01 to 1, the training accuracy improves slightly (from 89.91% to 90.85%), but the test accuracy remains relatively unchanged. This suggests that larger C values focus on fitting the training data more tightly, which offers little improvement in generalization.
- **Overfitting at $C = 100$:** When $C = 100$, the training accuracy decreases to 88.07%, and the test accuracy drops to 84.30%. This behavior suggests that the model is overfitting the noise in the training set, resulting in poor generalization to unseen data.
- **General Trend**: Smaller C values (e.g., 0.01) promote wider margins and better generalization, while very small C values (e.g., 0.001) lead to underfitting. On the other hand, larger C values (e.g., 10 and 100) result in overfitting, as evidenced by the reduced test accuracy.

These results highlight the importance of carefully selecting C to balance the trade-off between generalization and training accuracy. For this dataset, $C = 0.01$ provides the most consistent and robust performance.

9.2 Polynomial Kernel

The polynomial kernel introduces non-linearity into the SVM by mapping the data into a higher-dimensional space. It is defined as:

$$K(x_i, x_j) = (\gamma \cdot x_i^T x_j + r)^d,$$

where d is the degree of the polynomial, γ is the kernel scale (implicitly set in MATLAB), and C is the regularization parameter. A higher degree (d) allows for more complex decision boundaries, but it also increases the risk of overfitting.

The results for two different polynomial degrees ($n = 3$ and $n = 2$) are shown in Tables 6 and 10.

9.2.1 Performance with Degree 3 ($n = 3$)

Observations for Degree 3:

Table 6: Training and Test Accuracies for Different C Values with Polynomial Degree 3

C **value**	**Training Accuracy (%)**	**Test Accuracy (%)**
0.001	77.57	76.40
0.01	85.27	81.25
0.1	92.69	**82.55**
1	98.67	81.90
10	99.70	80.15

- **Best Performance at $C = 0.1$:** The highest test accuracy (82.55%) occurs at $C = 0.1$. At this value, the model achieves a good balance between training and test performance, indicating effective generalization.
- **Effect of Small C (0.001 and 0.01):** At very small C values, the training and test accuracies are lower, indicating underfitting. The model emphasizes a wider margin at the expense of classification accuracy.
- **Overfitting for Large C:** As C increases (e.g., $C = 1$ and $C = 10$), the training accuracy approaches 100%, but the test accuracy declines. This suggests that the model overfits the training data and struggles to generalize to unseen samples.

9.2.2 Performance with Degree 2 ($n = 2$)

Table 7: Training and Test Accuracies for Different C Values with Polynomial Degree 2

C **value**	**Training Accuracy (%)**	**Test Accuracy (%)**
0.01	80.61	79.25
0.1	86.30	82.75
1	92.94	**83.05**
10	98.58	81.05

Observations for Degree 2:

- **Best Performance at $C = 1$:** The highest test accuracy (83.05%) occurs at $C = 1$, showing that a moderate regularization strength works best for degree 2.
- **Better Generalization at Degree 2:** Compared to degree 3, the test accuracy remains slightly higher for degree 2 at its optimal C. This suggests that a simpler polynomial (lower degree) avoids overfitting and generalizes better to the test set.

- **Overfitting at Large C:** As seen for $C = 10$, the training accuracy increases to 98.58%, but the test accuracy drops to 81.05%. This again indicates overfitting.

9.2.3 Comparison Between Degrees

Comparing the results for $n = 2$ and $n = 3$:

- **Degree 2 Generalizes Better:** The test accuracy for degree 2 (83.05%) is slightly higher than the best test accuracy for degree 3 (82.55%). This highlights that a lower-degree polynomial creates smoother decision boundaries, reducing the risk of overfitting.
- **Degree 3 Overfits More Easily:** With degree 3, the training accuracy increases rapidly for larger C, but the test accuracy starts to decline, indicating overfitting.
- **Optimal C Values Differ:** For degree 3, the best performance is achieved at $C = 0.1$, while for degree 2, it occurs at $C = 1$. This shows that higher-degree polynomials require stronger regularization (smaller C) to avoid overfitting.

Overall, the results demonstrate that the degree of the polynomial kernel significantly impacts the model's ability to generalize. While degree 3 achieves good performance with careful tuning of C, degree 2 provides slightly better generalization and stability.

9.2.4 Performance with Higher Polynomial Degrees

To further analyze the impact of increasing the polynomial degree, Table 8 presents the training and test accuracies for polynomial degrees $n = 4$ and $n = 5$.

Table 8: Training and Test Accuracies for Different Polynomial Degrees

n **value**	**Training Accuracy (%)**	**Test Accuracy (%)**
4	49.34	49.15
5	51.35	50.75

Observations:

- **Low Accuracy for Higher Degrees:** Both the training and test accuracies remain very low for polynomial degrees $n = 4$ and $n = 5$, with test accuracies of 49.15% and 50.75%, respectively. This significant drop indicates that higher-degree polynomials overfit the training data poorly and fail to generalize.

- **Overfitting and Poor Generalization:** Increasing the polynomial degree introduces much more complexity to the decision boundaries. However, in this case, the higher complexity does not benefit the model, as the data likely does not contain sufficient non-linear relationships to justify these degrees. Instead, the model overfits noise in the training data, leading to low test accuracy.
- **Minimal Improvement Between Degrees 4 and 5:** Although the training accuracy improves slightly from 49.34% to 51.35%, the corresponding test accuracy shows only a marginal increase (49.15% to 50.75%). This highlights that increasing the degree further does not provide significant benefits and instead adds unnecessary complexity.
- **Comparison to Lower Degrees:** Compared to the results for degrees $n = 2$ and $n = 3$, where test accuracies reached 83.05% and 82.55%, respectively, higher degrees perform much worse. This confirms that lower-degree polynomials are better suited for this dataset, as they balance model complexity and generalization more effectively.

Overall, the results demonstrate that increasing the polynomial degree beyond $n = 3$ leads to diminishing returns and significant overfitting. Higher-degree polynomials are more prone to capturing noise rather than meaningful patterns in the data, resulting in poor generalization.

9.3 Rbf Kernel

Table 9: Training and Test Accuracies for Different C Values using RBF , classes 6 and 9

C **value**	**Training Accuracy (%)**	**Test Accuracy (%)**
0.01	82.20	79.35
0.1	89.51	86.25
1	96.32	90.15
10	100	90.20

The table illustrates the performance of the RBF kernel for binary classification between classes 6 (frog) and 9 (truck) under different values of the regularization parameter C. As the C value increases, the training accuracy consistently improves, reaching 100% at $C = 10$. This indicates that the model becomes increasingly focused on minimizing training error as C increases, which is expected since larger C values penalize misclassifications more heavily.

For test accuracy, the trend is slightly different. While it starts at a lower value of 79.35% for $C = 0.01$, it peaks at 90.20% for $C = 10$, showing a significant improvement as the model generalizes better. However, the gap between training and test accuracies at higher C values suggests potential overfitting, as the model perfectly memorizes the training data but struggles slightly on unseen data.

C controls how strictly the SVM minimizes misclassification. For small C (e.g., 0.01) the model focuses on a large margin and may underfit because it ignores the fine-grained structure in the data. For larger C (e.g., 1) the model places more emphasis on minimizing the training error, allowing it to capture non-linear patterns effectively. However, if C is too high (e.g., C = 100), the model might overfit. Overall, the RBF kernel achieves a balance of high training and test accuracy, with $C = 1$ providing the best generalization performance at 90.15% test accuracy and 96.32% training accuracy.

9.3.1 Kernel Scale

The KernelScale - γ parameter in the RBF kernel refers to the scaling factor of the Gaussian kernel, which determines the "width" of the kernel function. Indirectly KernelScale represents by specifying the standard deviation (or width) of the Gaussian kernel. It controls the influence of individual training examples, with smaller values of KernelScale (larger γ) leading to tighter kernels that focus on individual data points, and larger values of KernelScale (smaller γ) creating broader kernels that take into account a wider range of data.

Table 10: Training and Test Accuracies for Different Kernel Scale Values using RBF , classes 6 and 9, c=1

C value	**Training Accuracy (%)**	**Test Accuracy (%)**
1	100	50
10	98.11	90.10
100	85.23	82.35
auto	96.32	90.15

The table illustrates the impact of varying `KernelScale` on the training and test accuracies for binary classification between classes 6 (frog) and 9 (truck) with a fixed $C = 1$. A value of `KernelScale` set to 1 achieves perfect training accuracy but results in very poor generalization, as reflected in the low test accuracy of 50%. This indicates severe overfitting since the kernel is overly specific to the training data.

When `KernelScale` is increased to 10, the test accuracy improves significantly to 90.10%, suggesting better generalization as the kernel becomes broader and considers a wider range of data points. For `KernelScale` = 100, the test accuracy drops to 82.35%, indicating underfitting as the kernel becomes too broad and fails to capture perfectly the data.

The `KernelScale` set to "auto" dynamically computes the kernel scale based on the median distance between training points. This provides a balance between overfitting and underfitting, achieving a test accuracy of 90.15% with a training accuracy of 96.32%. This result demonstrates that the "auto" setting can adaptively select a kernel width that balances complexity and generalization.

The choice of `KernelScale` is crucial for balancing training and test performance. A poorly chosen kernel scale can lead to either overfitting or underfitting, whereas the ”auto” setting often provides a robust and adaptive solution.

Something that it has to be noted is that for all the cases, the code runs really fast.

10 Multi-Class Classification

If we wanted to try multi-class classification tasks, we could extend binary classifiers like SVMs to handle multiple classes effectively. We could handle the dataset with the One-vs-All (OvA) approach, the One-vs-One (OvO) approach, and leveraging built-in functions like MATLAB’s `fitcecoc`, which inherently supports multi-class problems.

The OvA approach involves training one SVM for each class, where the goal of each classifier is to distinguish one class from all other classes. For CIFAR-10, this would require training 10 separate SVMs. During prediction, the decision scores from all classifiers are computed, and the class with the highest score is selected. While OvA is conceptually simple and computationally efficient, it can suffer from challenges like imbalanced class decisions, especially when some classes are harder to distinguish.

The OvO approach, on the other hand, trains one SVM for every possible pair of classes. For CIFAR-10, this would require $\frac{n(n-1)}{2} = 45$ SVMs. During prediction, a voting scheme is employed, where each SVM casts a vote for one of the two classes it was trained on, and the class with the most votes is assigned. OvO is particularly advantageous when class overlaps are prominent, as each model focuses on distinguishing two specific classes. However, the computational and memory requirements can become significant as the number of classes grows.

MATLAB provides a built-in solution for multi-class classification through the `fitcecoc` function, which implements the Error-Correcting Output Codes (ECOC) framework. This method generalizes OvA and OvO by using coding matrices to define relationships between classes and binary classifiers.

We would expect the OvO approach to perform slightly better than OvA for complex datasets like CIFAR-10, as it allows each SVM to specialize in distinguishing two specific classes. However, the computational cost of training and maintaining 45 SVMs for CIFAR-10 can be prohibitive.

11 MLP with hinge loss

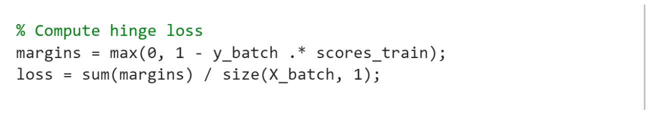
11.1 Hinge Loss and Its Role in Binary Classification

The **hinge loss** is a loss function commonly used for binary classification tasks. It measures the margin between the predicted scores and the true class labels.

The hinge loss is defined as:

$$\mathcal{L}_{\text{hinge}} = \frac{1}{N}\sum_{i=1}^{N} \max(0, 1 - y_i \cdot f(x_i)), \tag{1}$$

where $y_i \in \{-1, 1\}$ represents the true class label, $f(x_i)$ is the predicted score (logit) for sample x_i, and N is the number of samples. The loss is minimized when the prediction $f(x_i)$ is correctly classified and far enough (greater than 1 margin) from the decision boundary. Incorrect classifications or those close to the margin incur a positive loss. Hinge loss emphasizes maximizing the margin of separation between classes, encouraging robust generalization.



```
% Compute hinge loss
margins = max(0, 1 - y_batch .* scores_train);
loss = sum(margins) / size(X_batch, 1);
```

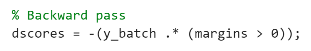
Figure 4: Hinge loss

11.2 Backpropagation with Hinge Loss and Binary Labels

When using hinge loss, the backpropagation process differs slightly from that used with losses like cross-entropy, which require one-hot encoded labels. Instead of one-hot vectors, binary labels $y_i \in \{-1, 1\}$ are directly used. The derivative of the hinge loss with respect to the predicted scores $f(x_i)$ is given by:

$$\frac{\partial \mathcal{L}_{\text{hinge}}}{\partial f(x_i)} = \begin{cases} -y_i, & \text{if } 1 - y_i \cdot f(x_i) > 0, \\ 0, & \text{otherwise.} \end{cases} \tag{2}$$

This gradient indicates that only samples violating the margin ($1 - y_i \cdot f(x_i) > 0$) contribute to the gradient, while correctly classified samples with sufficient margin do not. During backpropagation, the gradient flows back through the network layers, updating the weights based on these ”violating” samples. Additionally, the labels y_i are directly multiplied with the gradient at the final output layer, simplifying the calculation as it does not require converting labels to one-hot form.



```
% Backward pass
dscores = -(y_batch .* (margins > 0));
```

12 Comparison of all the methods

We evaluated the performance of all the aforementioned methods in this project using the entire dataset, focusing on a visually distinct class pair (*frog* vs. *truck*) and a more challenging, visually similar class pair (*cat* vs. *dog*).

The results indicate the following:

Table 11: Comparison of all the methods frog-truck

Method	**Training Accuracy (%)**	**Test Accuracy (%)**
1 Neighbour	-	78.50
3 - Neighbors	-	82.60
Nearest Centroid	-	76.25
MLP sigmoid	88.55	88.20
MLP relu	99.72	93.70
SVM linear	90.24	88.85
SVM polynomial n=2	100	91.85
SVM rbf c=1	96.78	93.25

- **Nearest Neighbor Methods**: The *1-Nearest Neighbor* and *3-Nearest Neighbors* algorithms achieve test accuracies of 78.50% and 82.60%, respectively. These methods are computationally simple but perform suboptimally, as they fail to generalize well for more complex data.
- **Nearest Centroid**: The *Nearest Centroid* method achieves the lowest test accuracy of 76.25%, highlighting its limited ability to distinguish complex classes due to its simple decision boundaries.
- **Multi-Layer Perceptron (MLP)**:
  - The *MLP with sigmoid activation* achieves a balanced performance with a training accuracy of 88.55% and a test accuracy of 88.20%, demonstrating good generalization.
  - The *MLP with ReLU activation* performs the best among all methods, achieving a training accuracy of 97.72% and a test accuracy of 93.70%. This indicates its ability to learn complex decision boundaries at the expense of overfitting.
- **Support Vector Machines (SVM)**:
  - The *SVM with a linear kernel* achieves a training accuracy of 90.24% and a test accuracy of 88.85%, demonstrating solid generalization.
  - The *SVM with a polynomial kernel (degree 2)* achieves perfect training accuracy (100%) but slightly lower test accuracy (91.85%), indicating potential overfitting.
  - The *SVM with an RBF kernel* and $c = 0.01$ performs exceptionally well, achieving a training accuracy of 96.78% and a test accuracy of 93.25%. This demonstrates the RBF kernel's capability to model complex relationships in the data effectively, without any significant overfitting.

Conclusion: The results show that the *MLP with ReLU activation* -although it overfits- and the *SVM with RBF kernel* provide the best performance, achieving test accuracies of 93.70% and 93.25%, respectively. The polynomial SVM achieves high training accuracy but shows slight overfitting. Simpler methods like Nearest Neighbor and Nearest Centroid exhibit significantly lower test accuracy, highlighting their limitations in handling the complexity of the CIFAR-10 dataset.

Table 12: Comparison of all the methods cat-dog

Method	**Training Accuracy (%)**	**Test Accuracy (%)**
1 Neighbour	-	57.50
3 - Neighbors	-	60.60
Nearest Centroid	-	57.25
MLP sigmoid	67.10	64.00
MLP relu	87.18	64.10
SVM linear 0.01	80.24	61.50
SVM polynomial n=3 c=0.1	99.84	63.65
SVM rbf c=1	85.71	67.30

The table compares the performance of various methods for binary classification of CIFAR-10 classes, specifically the challenging *cat* vs. *dog* pair. In contrast to earlier results for the *truck* vs. *frog* pair, where models achieved moderately higher accuracies, the current results highlight a significant drop in performance. For example, the 1-Nearest Neighbor achieves only 57.50% test accuracy, while the 3-Nearest Neighbors improves slightly to 60.60%, demonstrating the difficulty of distinguishing between visually similar classes.

The MLP methods show an improvement over the nearest neighbor approaches, with the sigmoid-activated MLP achieving 64.00% and the ReLU-activated MLP reaching 64.10% test accuracy. However, overfitting is evident in these methods, as seen in the training accuracies (67.10% and 87.18%, respectively). Among the SVM models, the linear SVM achieves a test accuracy of 61.50%, while the polynomial SVM with $n = 3$ and $C = 0.1$ achieves a slightly better test accuracy of 63.65% but suffers from severe overfitting with a training accuracy of 99.84%. The RBF SVM ($C = 1$) outperforms the other methods in this comparison, achieving a balanced training accuracy of 85.71% and a test accuracy of 67.30%.

Continuing the comparison with previous results, we observe that the performance of the methods depends heavily on the chosen class pairs. For visually distinct classes like *frog* and *truck*, the models achieved notably higher accuracies, as seen previously where the MLP and SVM (RBF) methods surpassed 87%. On the other hand, for the more challenging and visually similar pair, *cat* vs. *dog*, the performance of all models drops significantly.

The SVM with RBF kernel remains the most consistent performer across both class pairs, showcasing the ability to generalize better, even for visually complex categories. In contrast, the polynomial SVM tends to overfit the training data, with very high training accuracy but moderate test accuracy. The MLP with $ReLU$ activation shows promise, achieving competitive test accuracy despite overfitting slightly.

Overall, the results indicate that methods like the RBF SVM and MLP with ReLU activation are better suited for datasets with higher intra-class similarity, while simpler methods like linear SVM or nearest neighbors struggle to capture such subtle differences. For distinct class pairs, polynomial SVM and linear models perform adequately, but their tendency to overfit becomes more apparent when dealing with complex scenarios.

Last but not least, it has to be mentioned that the SVM, despite of the kernel function, is much more faster than the other methods, except Nearest Centroid. For example, for an SVM half minute is needed, while for an MLP at least 2-3 min for the appropriate number of epochs. Although nearest centroid method is the fastest, as its duration is less than a second, it shows the worst performance in the accuracies.

12.1 Conclusion

From this comparison, it is clear that while the MLP and SVM methods outperform simpler approaches such as nearest neighbors and nearest centroid, the overall test accuracies remain modest. The results highlight the challenge of distinguishing between visually similar classes (e.g., *cat* vs. *dog*), where even advanced methods like SVMs with polynomial and RBF kernels struggle to generalize. Among the tested methods, the **RBF SVM** provides the best balance between training and test performance, demonstrating its ability to capture non-linear patterns in the data. However, further optimization, regularization, or more advanced techniques would be required to achieve better generalization for such complex class pairs.