CSE574 Introduction to Machine Learning

Programming Assignment 4

Classification and Regression

Prathibha Vuyyala - 50559983, Tharun Teja Mogili - 50559877,
Arun Sandilya Nanyam-50541117

Overview

The objective of this project is to implement classification models using Logistic Regression (LR) and Support Vector Machines (SVMs) for the MNIST handwritten digit dataset. We developed a one-vs-all logistic regression classifier, applied gradient descent for parameter optimization, and extended the model to multi-class classification using the softmax function. Additionally, we used SVMs with different kernels and hyperparameter settings to improve prediction accuracy. The performance of these models was evaluated on training, validation, and testing sets based on classification accuracy and error rates. Experimental results and comparisons between the models are discussed in detail.

Problem 1: Implementation of Logistic Regression

we implemented a one-vs-all logistic regression model to classify handwritten digit images from the MNIST dataset into 10 distinct classes. Since logistic regression is inherently a binary classifier, we trained 10 separate binary classifiers, each distinguishing one digit from the others. The preprocessed dataset was split into training, validation, and testing sets, with features normalized between 0 and 1.

Implementation Details:

1. **Objective Function (blrObjFunction):**

   We implemented the cross-entropy loss function to compute the error between predicted and actual labels. We also computed the gradient of the loss function using the formula:

$$E(\mathbf{w}) = -\frac{1}{N}\ln p(\mathbf{y}|\mathbf{w}) = -\frac{1}{N}\sum_{n=1}^{N}\{y_n \ln \theta_n + (1-y_n)\ln(1-\theta_n)\}$$

   where θn is the predicted probability using the sigmoid function.

2. **Prediction Function (blrPredict):**

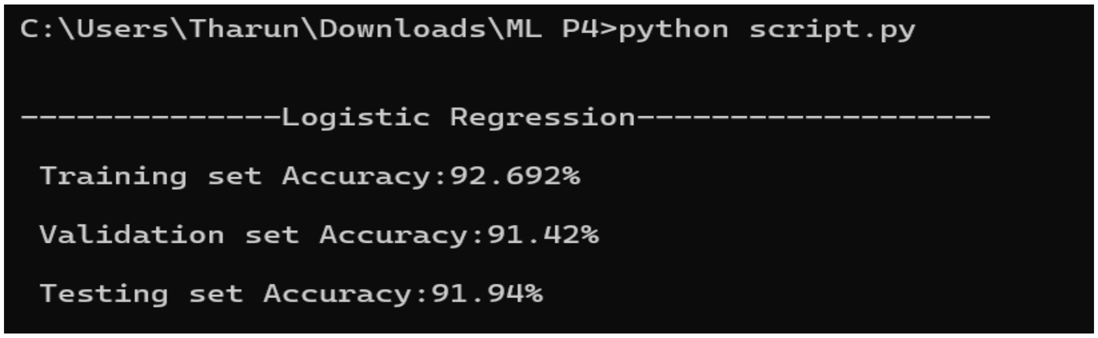
   Using the learned weight matrix W, we computed the posterior probability for each class using the sigmoid function and assigned each feature vector to the class with the maximum posterior probability.

3. **Training Process:**

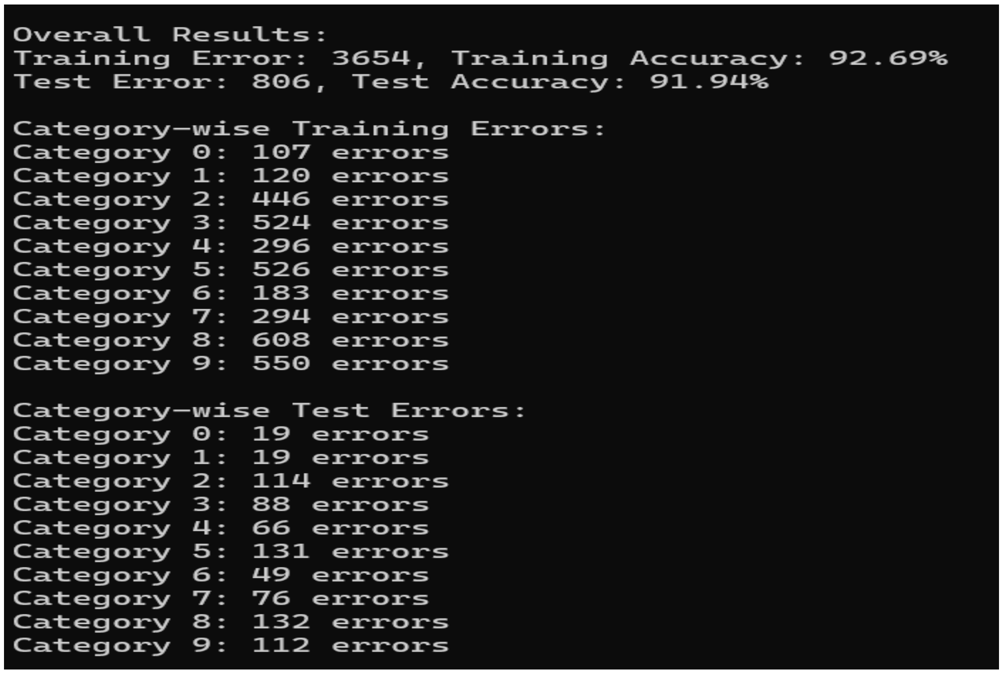
   - We used the scipy.optimize.minimize function to minimize the cross-entropy error for each classifier using gradient descent.
   - The optimization process was repeated for all 10 classifiers, storing their weight vectors in matrix W.

Results and Accuracy:

We evaluated the model's performance on training, validation, and testing sets. The achieved accuracies were:



```
C:\Users\Tharun\Downloads\ML P4>python script.py


--------------Logistic Regression-------------------

 Training set Accuracy:92.692%

 Validation set Accuracy:91.42%

 Testing set Accuracy:91.94%
```



```
Overall Results:
Training Error: 3654, Training Accuracy: 92.69%
Test Error: 806, Test Accuracy: 91.94%

Category-wise Training Errors:
Category 0: 107 errors
Category 1: 120 errors
Category 2: 446 errors
Category 3: 524 errors
Category 4: 296 errors
Category 5: 526 errors
Category 6: 183 errors
Category 7: 294 errors
Category 8: 608 errors
Category 9: 550 errors

Category-wise Test Errors:
Category 0: 19 errors
Category 1: 19 errors
Category 2: 114 errors
Category 3: 88 errors
Category 4: 66 errors
Category 5: 131 errors
Category 6: 49 errors
Category 7: 76 errors
Category 8: 132 errors
Category 9: 112 errors
```

Analysis of Results:

The logistic regression model showed strong performance, with accuracies of 92.69% on the training set, 91.42% on the validation set, and 91.94% on the testing set. The total training error was 3654, while the testing error was 806, indicating a slight gap between the training and testing results. Category-wise analysis revealed that digits like 0 and 1 had the lowest errors, with 107 and 120 errors in training and 19 errors each in testing. In contrast, digits like 8 and 9 showed significantly higher errors, with 608 and 550 errors in training and 132 and 112 errors in testing, highlighting the model's challenges in handling visually complex and overlapping digit patterns.

The difference between training and testing errors can be attributed to several factors. Logistic regression, as a linear classifier, assumes linear decision boundaries, which may not effectively separate complex and overlapping classes like 8 and 9. The training set accuracy is slightly higher since the model directly optimized for it, while the unseen test data naturally introduces variability due to differences in digit styles and patterns. Additionally, digits with higher visual complexity or overlap (e.g., 3, 5, 8, and 9) tend to have higher error rates, reflecting the model's limitations in handling subtle differences in such categories.

Category-specific challenges also contribute to the gap in performance. Digits like 8, which have structural similarities to other digits such as 3 and 5, are more prone to misclassification. Similarly, 9 and 7 may overlap visually, leading to errors. The model performed consistently across datasets, but its linear nature and reliance on pixel intensity as features limit its ability to fully capture the nuances of handwritten digit variations, particularly for more complex categories. These findings highlight areas where the model performs well and where improvements could be targeted.

Problem 2: Implementation of Multi-class Logistic Regression

In this section, we implemented a multi-class logistic regression model using the softmax function to classify handwritten digit images into 10 classes. Unlike the one-vs-all approach, which requires building 10 separate classifiers, the multi-class approach uses a single classifier to predict all classes simultaneously. This method computes the probability distribution over all classes using the softmax function and assigns each input to the class with the highest probability.

Implementation Details:

1. **Objective Function (mlrObjFunction):**
   The softmax function computes the posterior probabilities for all classes. For a given input vector x, the probability of class k is calculated as:

$$P(y = C_k|\mathbf{x}) = \frac{\exp(\mathbf{w}_k^T\mathbf{x})}{\sum_j \exp(\mathbf{w}_j^T\mathbf{x})}$$

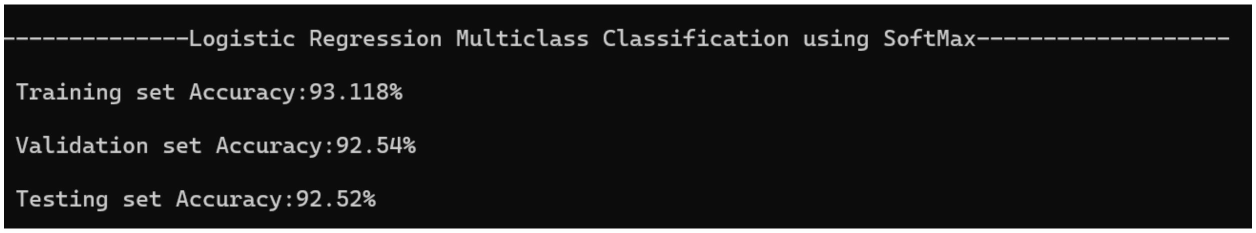
The cross-entropy error function was minimized using gradient descent to optimize the weight matrix W. The gradient of the error function was computed and used to update the weights.

2. **Prediction Function (mlrPredict):**
   The model predicts the class of each input by computing posterior probabilities for all classes and selecting the class with the maximum probability. Bias terms were added to the input features to maintain consistency.

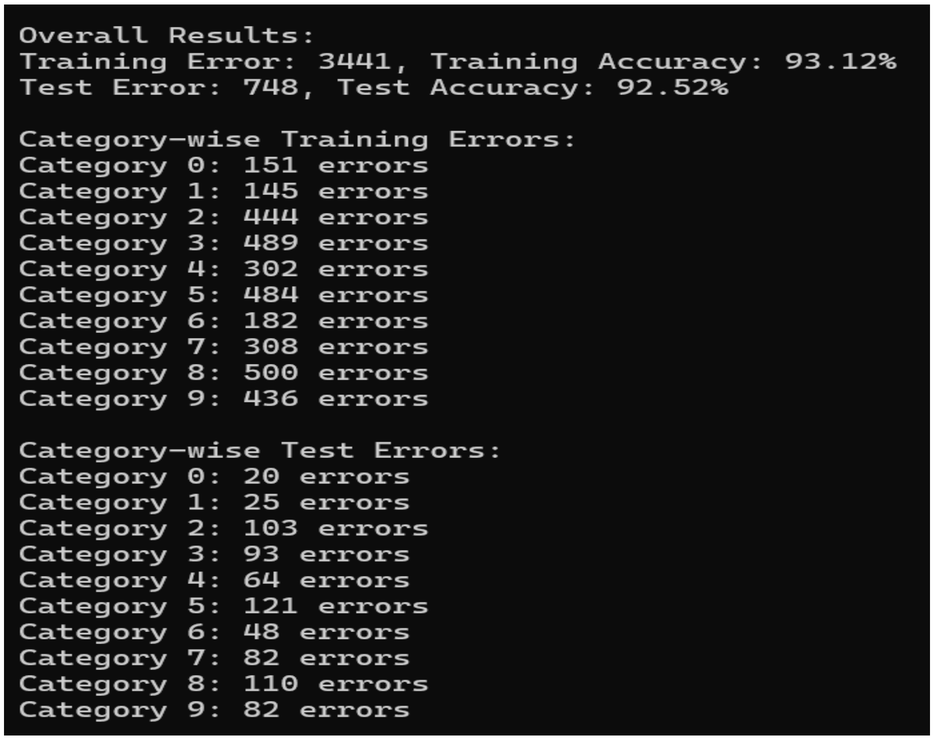
3. Training Process:
   Using the scipy.optimize.minimize function, we optimized the multi-class logistic regression model with a maximum of 100 iterations. The final weight matrix Wb was used for classification across all datasets.

Results and Accuracy:

The performance of the multi-class logistic regression model was evaluated on the training, validation, and testing datasets. The results are summarized as follow



```
-------------Logistic Regression Multiclass Classification using SoftMax-------------------

 Training set Accuracy:93.118%

 Validation set Accuracy:92.54%

 Testing set Accuracy:92.52%
```



```
Overall Results:
Training Error: 3441, Training Accuracy: 93.12%
Test Error: 748, Test Accuracy: 92.52%

Category-wise Training Errors:
Category 0: 151 errors
Category 1: 145 errors
Category 2: 444 errors
Category 3: 489 errors
Category 4: 302 errors
Category 5: 484 errors
Category 6: 182 errors
Category 7: 308 errors
Category 8: 500 errors
Category 9: 436 errors

Category-wise Test Errors:
Category 0: 20 errors
Category 1: 25 errors
Category 2: 103 errors
Category 3: 93 errors
Category 4: 64 errors
Category 5: 121 errors
Category 6: 48 errors
Category 7: 82 errors
Category 8: 110 errors
Category 9: 82 errors
```

Analysis of Results:

The multi-class logistic regression model demonstrated strong performance, with accuracies of 93.12%, 92.54%, and 92.52% on the training, validation, and testing datasets, respectively. The slight drop in accuracy from training to testing is expected, as the model generalizes from seen data to unseen data. Category-wise errors reveal that digits like 0 and 1, which are visually distinct, had fewer errors (e.g., 151 and 145 training errors; 20 and 25 testing errors), while more complex digits like 8 and 9 had higher errors (e.g., 500 and 436 training errors; 110 and 82 testing errors). These findings are consistent with challenges posed by overlapping patterns in handwritten digits.

The difference between training and testing errors can be attributed to the natural variability in digit styles and patterns within the MNIST dataset. While the softmax-based multi-class approach effectively models relationships across all classes, its linear nature limits its ability to handle intricate visual complexities, leading to higher misclassification rates for digits like 8 and 9. The model's reliance on pixel intensity features further highlights the need for more sophisticated methods to capture complex patterns.

Compared to the one-vs-all strategy, the multi-class model achieved slightly higher accuracy and better computational efficiency. The single classifier approach leverages shared information across classes, reducing redundancy and improving generalization. However, both approaches share similar limitations when distinguishing visually similar or complex digits. Overall, the multi-class logistic regression model showcased robust performance while highlighting areas for potential improvement, such as incorporating non-linear decision boundaries or advanced feature engineering.

Problem 3: Implementation of Support Vector Machines

Support Vector Machines (SVMs) were used to classify the MNIST handwritten digit dataset using both linear and radial basis function (RBF) kernels. Various hyperparameters, including the regularization parameter C and gamma, were explored to evaluate their impact on classification accuracy. The experiments aimed to identify the optimal configuration for balancing training, validation, and testing performance.

Implementation Details:

Linear Kernel:

- A linear kernel was used to create a simple decision boundary. All other hyperparameters were kept at their default values.

RBF Kernel with Gamma = 1:

- The RBF kernel, which introduces non-linear decision boundaries, was evaluated with gamma fixed at 1. This configuration focuses on the influence of each data point in the decision boundary.

RBF Kernel with Default Gamma:

- The default "scale" gamma value was used to adjust the kernel width dynamically based on the input data.

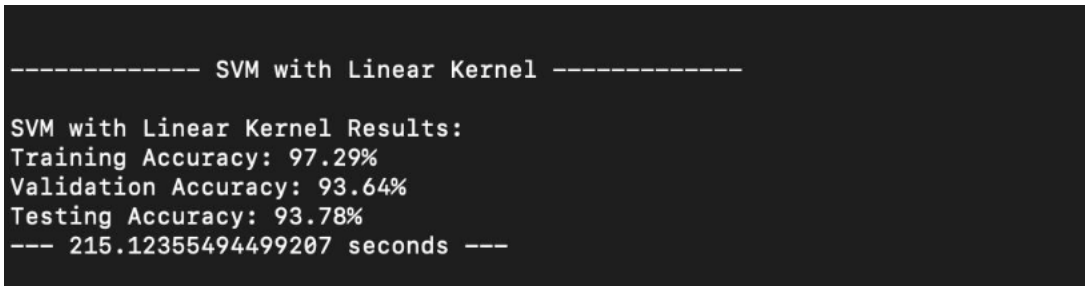
RBF Kernel with Varying C:

- The regularization parameter C was varied from 10 to 100 in increments of 10 to observe its effect on model performance. Higher C values reduce misclassification on training data but may lead to overfitting.

For all configurations, the SVC module from scikit-learn was used, and accuracies were recorded for training, validation, and testing datasets.

Results and Accuracy:

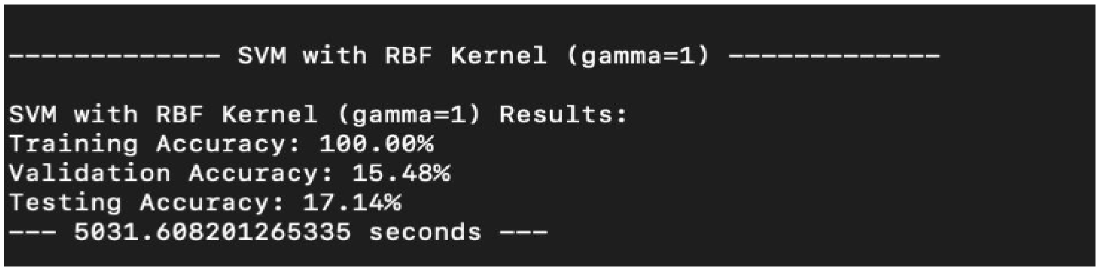
1. **Linear Kernel:**



```
------------- SVM with Linear Kernel -------------

SVM with Linear Kernel Results:
Training Accuracy: 97.29%
Validation Accuracy: 93.64%
Testing Accuracy: 93.78%
--- 215.12355494499207 seconds ---
```

The linear kernel achieved reasonable accuracy but was limited by its inability to capture non-linear relationships in the data.

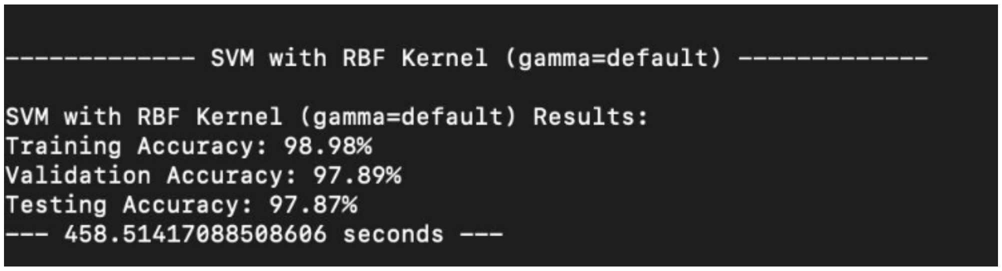
2. **RBF Kernel (Gamma = 1):**



```
------------- SVM with RBF Kernel (gamma=1) -------------

SVM with RBF Kernel (gamma=1) Results:
Training Accuracy: 100.00%
Validation Accuracy: 15.48%
Testing Accuracy: 17.14%
--- 5031.608201265335 seconds ---
```

This configuration overfitted the training data, resulting in poor performance on validation and testing datasets due to an excessively small kernel width.

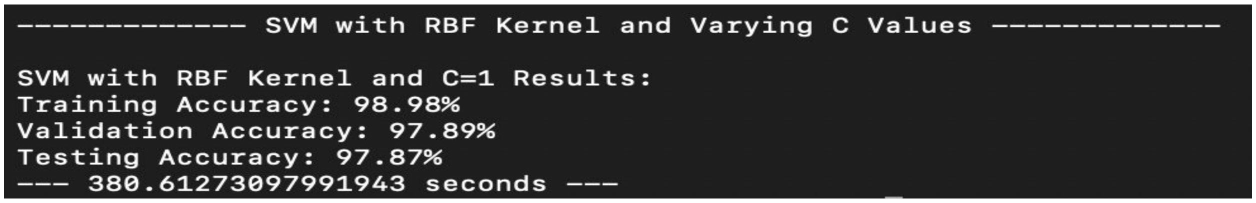
3. **RBF Kernel (Default Gamma):**



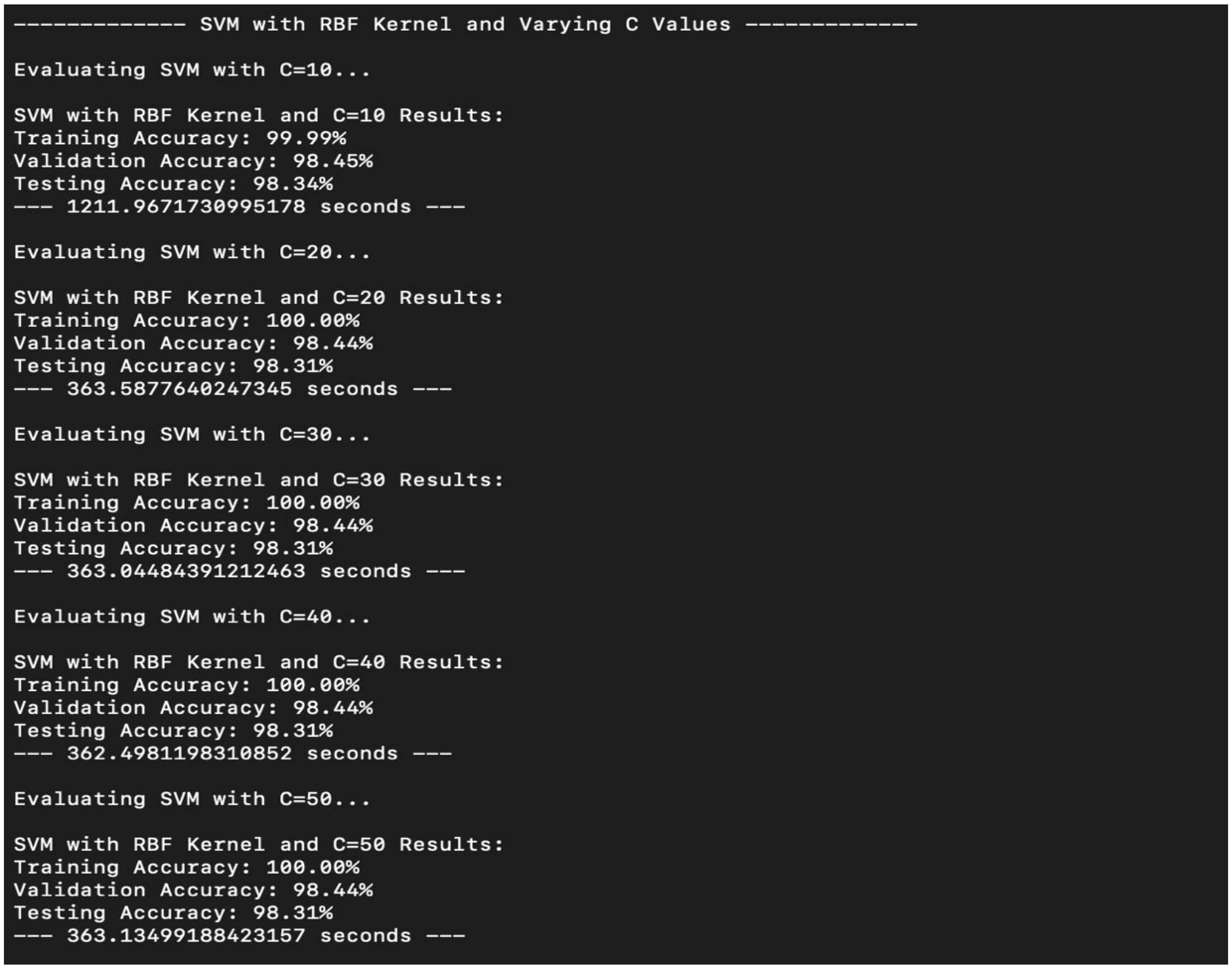
```
------------- SVM with RBF Kernel (gamma=default) -------------

SVM with RBF Kernel (gamma=default) Results:
Training Accuracy: 98.98%
Validation Accuracy: 97.89%
Testing Accuracy: 97.87%
--- 458.51417088508606 seconds ---
```

Using the default gamma value achieved the best balance between training and testing accuracies, indicating effective generalization.

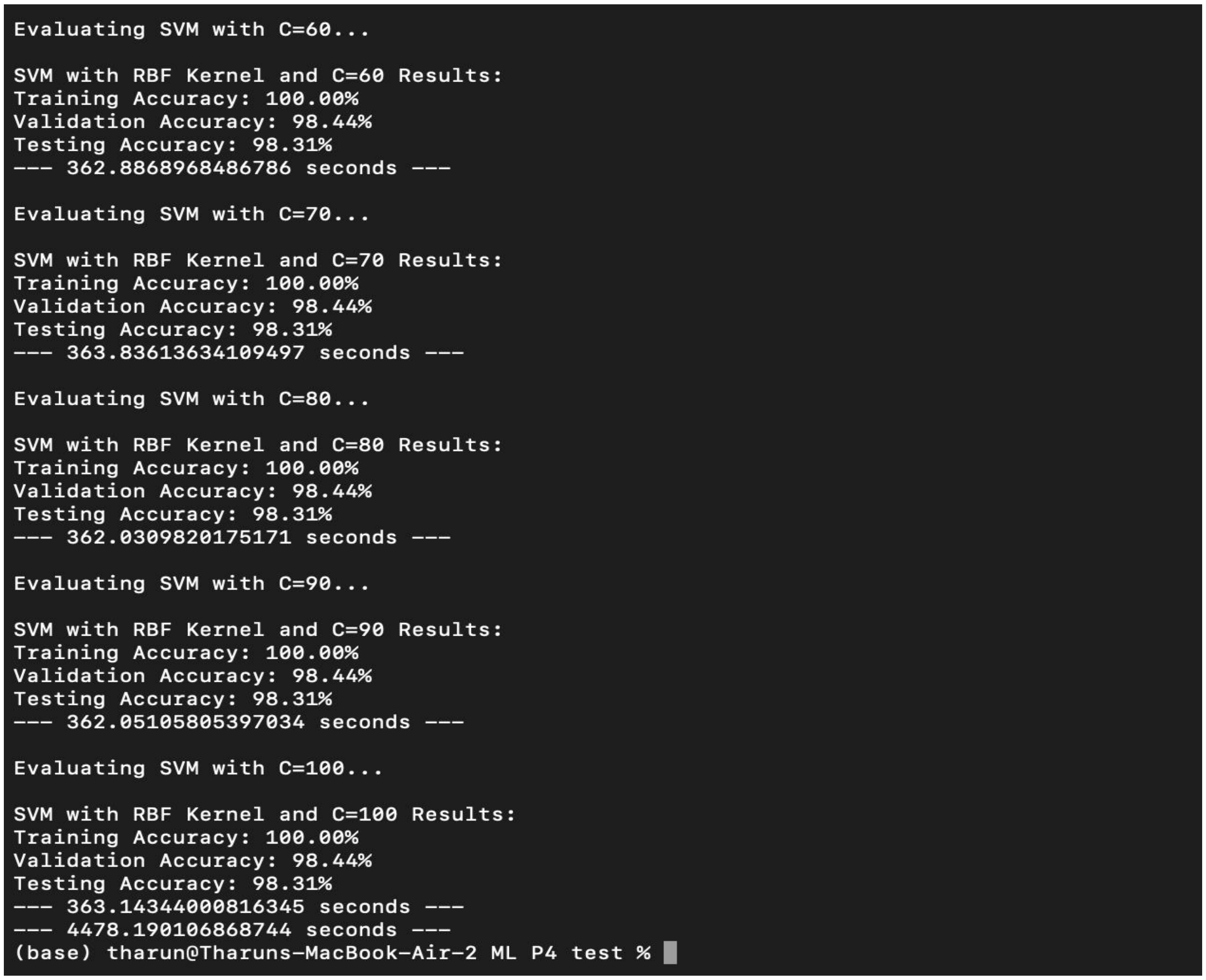
4. **RBF Kernel with Varying C:**



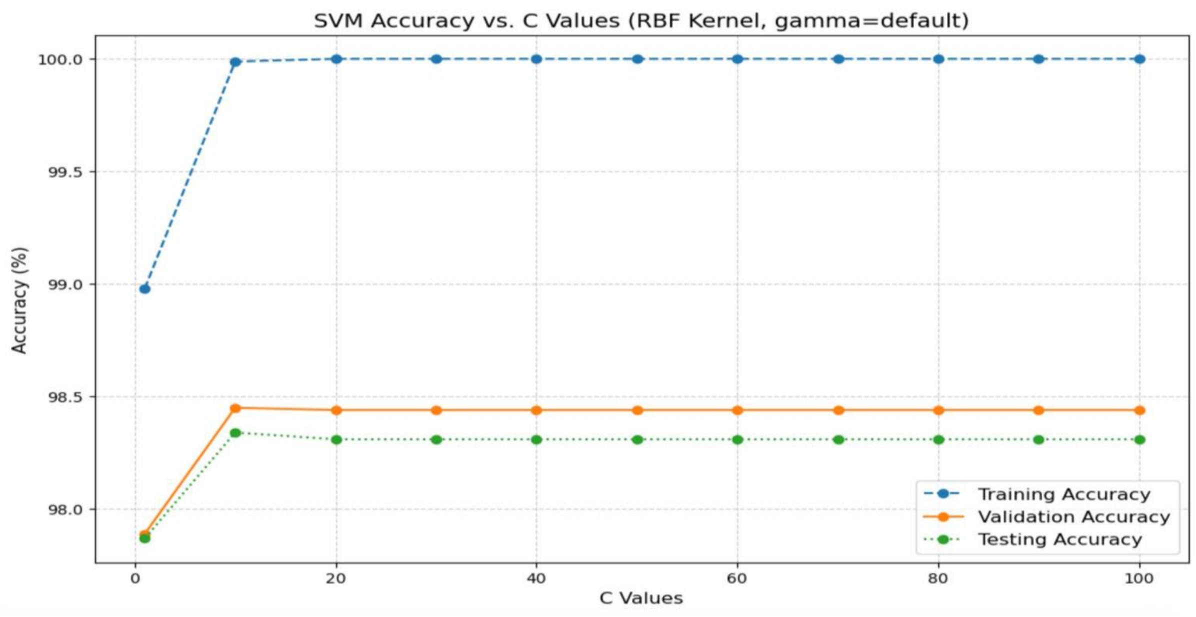
```
------------- SVM with RBF Kernel and Varying C Values -------------

SVM with RBF Kernel and C=1 Results:
Training Accuracy: 98.98%
Validation Accuracy: 97.89%
Testing Accuracy: 97.87%
--- 380.61273097991943 seconds ---
```



```
------------- SVM with RBF Kernel and Varying C Values -------------

Evaluating SVM with C=10...

SVM with RBF Kernel and C=10 Results:
Training Accuracy: 99.99%
Validation Accuracy: 98.45%
Testing Accuracy: 98.34%
--- 1211.9671730995178 seconds ---

Evaluating SVM with C=20...

SVM with RBF Kernel and C=20 Results:
Training Accuracy: 100.00%
Validation Accuracy: 98.44%
Testing Accuracy: 98.31%
--- 363.5877640247345 seconds ---

Evaluating SVM with C=30...

SVM with RBF Kernel and C=30 Results:
Training Accuracy: 100.00%
Validation Accuracy: 98.44%
Testing Accuracy: 98.31%
--- 363.04484391212463 seconds ---

Evaluating SVM with C=40...

SVM with RBF Kernel and C=40 Results:
Training Accuracy: 100.00%
Validation Accuracy: 98.44%
Testing Accuracy: 98.31%
--- 362.4981198310852 seconds ---

Evaluating SVM with C=50...

SVM with RBF Kernel and C=50 Results:
Training Accuracy: 100.00%
Validation Accuracy: 98.44%
Testing Accuracy: 98.31%
--- 363.13499188423157 seconds ---
```



```
Evaluating SVM with C=60...

SVM with RBF Kernel and C=60 Results:
Training Accuracy: 100.00%
Validation Accuracy: 98.44%
Testing Accuracy: 98.31%
--- 362.8868968486786 seconds ---

Evaluating SVM with C=70...

SVM with RBF Kernel and C=70 Results:
Training Accuracy: 100.00%
Validation Accuracy: 98.44%
Testing Accuracy: 98.31%
--- 363.83613634109497 seconds ---

Evaluating SVM with C=80...

SVM with RBF Kernel and C=80 Results:
Training Accuracy: 100.00%
Validation Accuracy: 98.44%
Testing Accuracy: 98.31%
--- 362.0309820175171 seconds ---

Evaluating SVM with C=90...

SVM with RBF Kernel and C=90 Results:
Training Accuracy: 100.00%
Validation Accuracy: 98.44%
Testing Accuracy: 98.31%
--- 362.05105805397034 seconds ---

Evaluating SVM with C=100...

SVM with RBF Kernel and C=100 Results:
Training Accuracy: 100.00%
Validation Accuracy: 98.44%
Testing Accuracy: 98.31%
--- 363.14344000816345 seconds ---
--- 4478.190106868744 seconds ---
(base) tharun@Tharuns-MacBook-Air-2 ML P4 test %
```



The results showed minimal impact of C on validation and testing accuracy, which stabilized at around **98.44%** and **98.31%**, respectively. The training accuracy remained at **100%** across all C values, indicating the model was perfectly fit to the training data.

Comparison of Kernel Types and Gamma Settings:

Linear Kernel vs. RBF Kernel:

The linear kernel achieved a testing accuracy of 93.78%, demonstrating its capability to separate linearly separable data. However, it underperformed compared to the RBF kernel with default gamma, which achieved a testing accuracy of 97.87%. The linear kernel's inability to capture non-linear relationships limited its effectiveness for the complex patterns present in the MNIST dataset. In contrast, the RBF kernel, with its non-linear decision boundaries, was better suited to high-dimensional and overlapping data points.

Effect of Gamma Settings in RBF Kernel:

- **Gamma = 1**: Fixing gamma to 1 led to severe overfitting, with 100% training accuracy but only 17.14% testing accuracy. This happened because a small gamma value created very narrow kernel widths, causing the model to memorize individual training samples instead of learning generalized patterns.
- **Default Gamma**: Using the default gamma setting allowed the kernel width to dynamically adjust based on the input data. This configuration balanced model complexity, achieving 98.98% training accuracy, 97.89% validation accuracy, and 97.87% testing accuracy, making it the most effective setup for this dataset.

Impact of Varying C:

Varying the regularization parameter C from 10 to 100 had minimal impact on validation and testing accuracy, which stabilized at 98.44% and 98.31%, respectively. The training accuracy remained at 100%, indicating that the model had already achieved its optimal capacity to fit the data. Increasing C further only strengthened the model’s preference for minimizing training error but did not improve generalization performance.

In conclusion, the RBF kernel with default gamma provided the best balance between training and testing performance, while fixed gamma values (e.g., gamma = 1) resulted in overfitting. The linear kernel, though computationally simpler, was less capable of handling non-linear complexities inherent in the MNIST dataset. These results highlight the importance of tuning kernel and hyperparameter choices for optimal SVM performance.