Customer Sentiment Analysis

Introduction and Goal

This project utilized customer product reviews from Amazon, Yelp, and IMDB, to classify customer sentiment towards products through Support Vector Machines (SVM), Random Forests, and Multi Layer Perceptrons (MLP).

The **goal** of this project was to train machine learning models that can automatically assign sentiment labels to new customer reviews, streamlining and automating the feedback assessment process.

Data

The data provided for this task consisted of **2,400** single-sentence product reviews, collected from three domains: **imdb.com, amazon.com, and yelp.com.** Each review was assigned a binary label (1 for positive and 0 for negative) indicating the customer sentiment towards the product.

Examples of positive and negative reviews are provided below:

Positive reviews:

1. (amazon) 1 It Works - 2 It is Comfortable
2. (imdb) Gotta love those close-ups of slimy, drooling teeth!
3. (yelp) Food was so gooodd.

Negative reviews:

1. (amazon) DO NOT BUY DO NOT BUYIT SUCKS
2. (imdb) This is not movie-making.
3. (yelp) The service was poor and thats being nice.

Data Preprocessing, Feature Representation

I performed the following steps to clean and transform data for model building:

1. **Text Preprocessing**

   - **Text only:** Some reviews contained numerical data. The numbers were not useful in our case since sentiment analysis aims to decipher user sentiment through text. Hence, numerical data was removed.

     Examples of reviews containing numerical characters are given below:

     – horrible, had to switch 3 times
     – We have tried 2 units and they both failed within 2 months

   - **Lowercasing:** Converted all characters to lowercase to ensure uniformity.
   - **Non-English character removal:** Removed non-ASCII characters to retain English text only.
   - **Punctuation removal:** Eliminated punctuation to simplify the token space.
   - **Whitespace normalization:** Replaced multiple spaces with a single space and stripped leading/trailing whitespace.

   These operations produced a clean textual representation, which I stored in a new `clean_text` column within the training set.

2. **Feature Representation using Term Frequency-Inverse Document Frequency (TF-IDF)**

   The cleaned textual data was then transformed into numerical vectors using TF-IDF approach. TF-IDF is a numerical statistic that reflects the importance of a word in a collection.

   It is a combination of Term Frequency (TF), i.e., how often a term appears in a document, or data observation, in our case; and Inverse

Document Frequency (IDF) which measures how common or rare a term is across the entire dataset.

TF is calculated by dividing the number of times a term appears in a data observation by the total number of words in the observation. Whereas, IDF is calculated by taking the logarithm of the total number of observations in the dataset divided by the number of observations containing the term.[1]

The number of unique words in the training set was 4,679.[2]

To perform feature representation using TF-IDF, I took the following design decisions:

- Used `TfidfVectorizer` from `scikit-learn` to build a function identifying the 1,000 most frequently used words in the dataset.
- Initially, I limited the function to identify top 2,000 words in the dataset. However, this captured words like *design, die, despite, be,* etc., along with helpful words like, *amazing, pathetic*, etc. Classifying such a high number of words as **important** risks overfitting the model on the training set. Therefore, I modified the function to identify top 1,500 words. I was still getting words like *ago, all, as, ask,* etc. Hence, I then restricted the function to identify the top 1,000 words in the dataset. Restricting the vocabulary to 1,000 reduced the number of redundant words. For example, the majority of words captured under this assumption expressed customer sentiment towards a product. Examples of words captured include, but are not limited to, *unreliable, pathetic, horrible, excellent,* etc. All these words indicate user sentiment towards a product.

  Even after restricting the `TfidfVectorizer` to 1,000 words, less useful words, such as, *in, instead, including,* etc., were getting captured. But I decided to not reduce the count further because 1,000 words is less than 25% of all unique words.[3]. I believe at 1,000 words, I hit the sweet spot between overfitting and underfitting.

[1] https://medium.com/@er.iit.pradeep09/understanding-tf-idf-in-nlp-a-comprehensive-guide-26707db0cec5

[2] Used `CountVector` from Sklearn to determine this.

[3] 1000/4679 = 0.214

- As a sanity check, I obtained a sparse feature matrix of shape $(n\_samples, n\_features)$. The output was (2400, 1000), which seems correct.

This representation captured both the frequency and informativeness of words across the dataset and was used as input for the machine learning models.

Bag-of-Words (BoW) Feature Representation

Next, I performed a BoW feature representation to convert each review into a structured numeric format suitable for machine learning.

BoW is a machine learning technique that counts the frequency of a word in a document, or data observation, in our case.

I implemented the BoW model using `CountVectorizer` from the `scikit-learn` library. Below are the steps and corresponding design decisions I took:

1. **Import and Initialization:**
   Imported the `CountVectorizer` tool and initialized it with the following parameters:

   - `max_features=1000`: Restricted the vocabulary to the 1000 most important words.
   - `binary=True`: Created binary vectors indicating word presence (1) or absence (0). Note that this decision was based on the whether an observation contained the 1000 most important words defined earlier.
   - `stop_words='english'`: Removed common English stopwords, such as, *and*, *or*, *the*, etc.
   - `min_df=2`: Excluded very rare words (those appearing in fewer than 2 data observations).
   - `max_df=0.9`: Excluded overly common words (those appearing in more than 90% of the observations.).
   - `ngram_range=(1,1)`: Used unigrams (single words) only. For example, *good looks* got divided into

```
["good", "looks"]
```

2. **Fitting and Transformation:**
   I then applied my vectorizer function to the cleaned training set using:

   ```
   X_BoW = vectorizer.fit_transform(x_train_df['clean_text'])
   ```

3. **Matrix Shape Inspection:**
   Afterwards, I printed the shape of the resulting feature matrix to confirm its dimensions:

   ```
   print("BoW matrix shape:", X_bow.shape)
   ```

   This resulted in a matrix of shape $(2400, 1000)$ indicating 2400 reviews and 1000 vocabulary features.

4. **DataFrame Conversion:**
   For visualization, I converted the sparse matrix into a dense DataFrame using:

   ```
   pd.DataFrame(X_bow.toarray(), columns=vectorizer.get_feature_names_out())
   ```

The result was a consistent, interpretable, and sparse feature representation that captured the presence or absence of relevant words in each review.

I then built three classifiers: **Support Vector Machines, Random Forests, and Multi Layer Perceptrons.**

Support Vector Machines

Support Vector Machines are a supervised machine learning technique. SVMs are used to separate data classes using a hyperplane such that the distance between decision boundary and training patterns is maximized. This reduces

the chances of misclassifying data points that lie very close to the decision boundary.[4]

For the purposes of classifying customer sentiment, I trained the SVM classifier with a Radial Basis Function (RBF)[5] on the pre-processed BoW feature vector. To control the model's complexity and instances of overfitting, I systemtically varied two hyperparameters: regularization strength, C, and the kernel coefficient, γ.

Hyperparameter Tuning

- **Regularization Strength(C):** Controls the bias-variance tradeoff. Larger C values lead to better data classification (low bias) on the training set, but risk overfitting (high variance). On the other hand, smaller C values may lead to higher bias and lower variance, risking underfitting. Hence, the goal is to find a C value that hits the sweet spot between over and underfitting.

- **Kernel Coefficient (γ):** Determines whether points close and far away from the separation line are considered into separation line calculation. In other words, with low γ, points far away from the separation line are considered when calculating the separation line and vice versa. Low γ values will therefore have instances of misclassifications, but lower risk of overfitting.

I explored a grid of values:

- $C \in \{0.01, 0.1, 1, 10, 100\}$
- $\gamma \in \{10^{-3}, 10^{-2}, 0.1, 1, 10\}$

[4]Nandi, K. Asoke, and Ahmed, Hosameldin. *Condition Monitoring with Vibration Signals: Compressive Sampling and Learning Algorithms for Rotating Machines.* Wiley, p. 259, 2019.

[5]RBF measures the similarity between two data points, *center and input*, based on their Euclidean distance, with a higher similarity score for points that are closer to each other and vice versa. RBF is particularly useful here because the data may not be seperabale by a simple linear boundary.

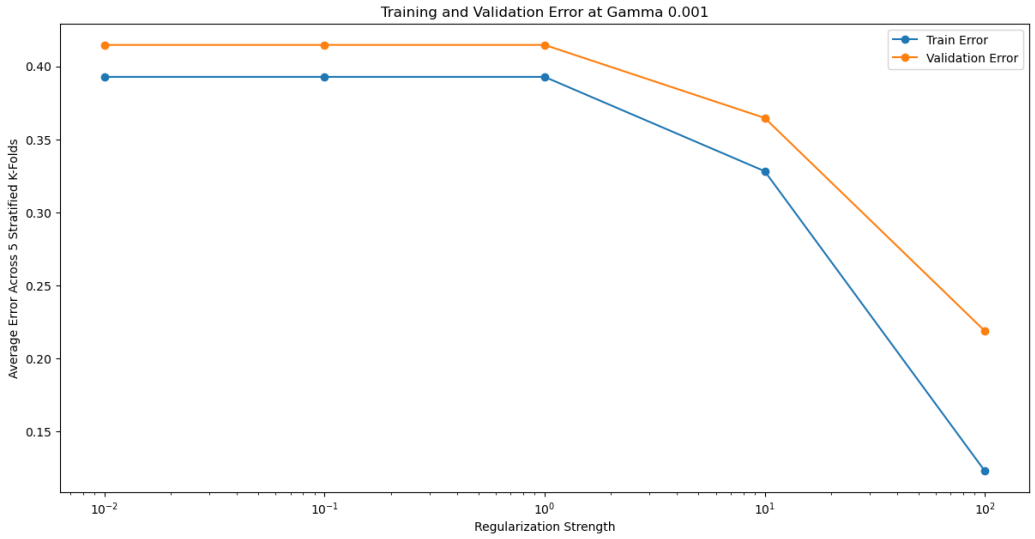
Cross-Validation Methodology

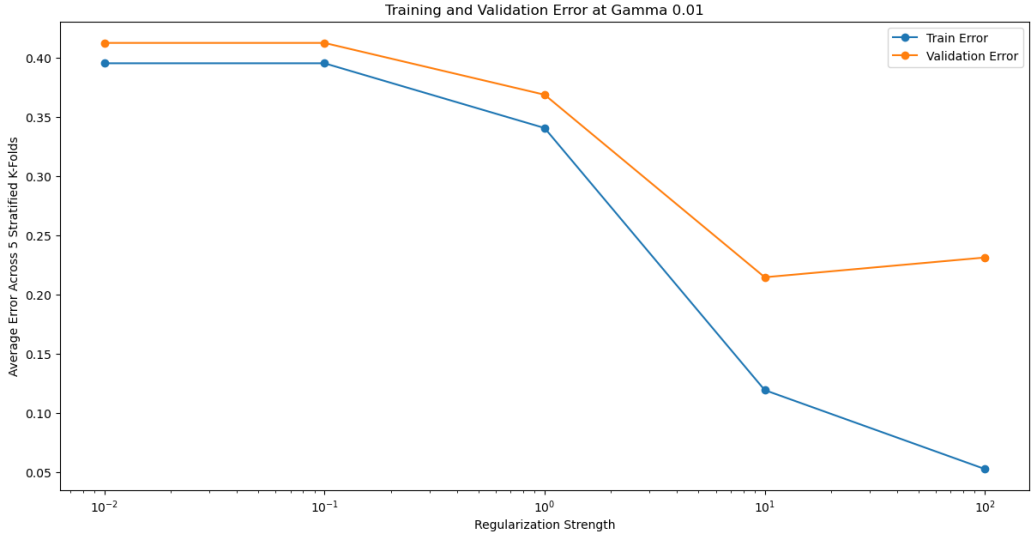
To evaluate the performance of each (C, γ) pair, I performed 5-fold Stratified Cross-Validation, ensuring that each fold preserved the overall class distribution. I recorded both training and validation accuracy across the folds, and then computed the average error, defined as, **1 - accuracy**, across all 5 folds.

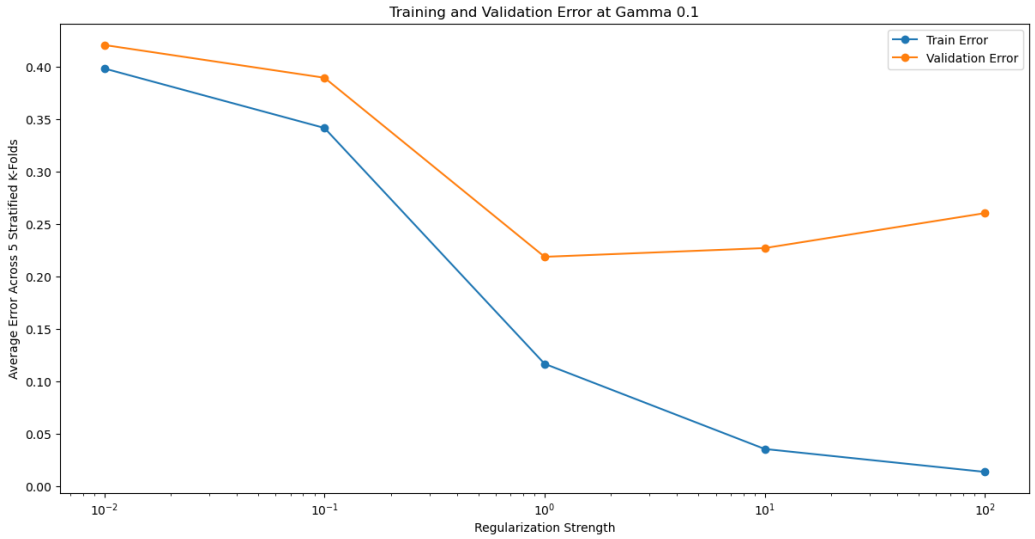
Results and Observations

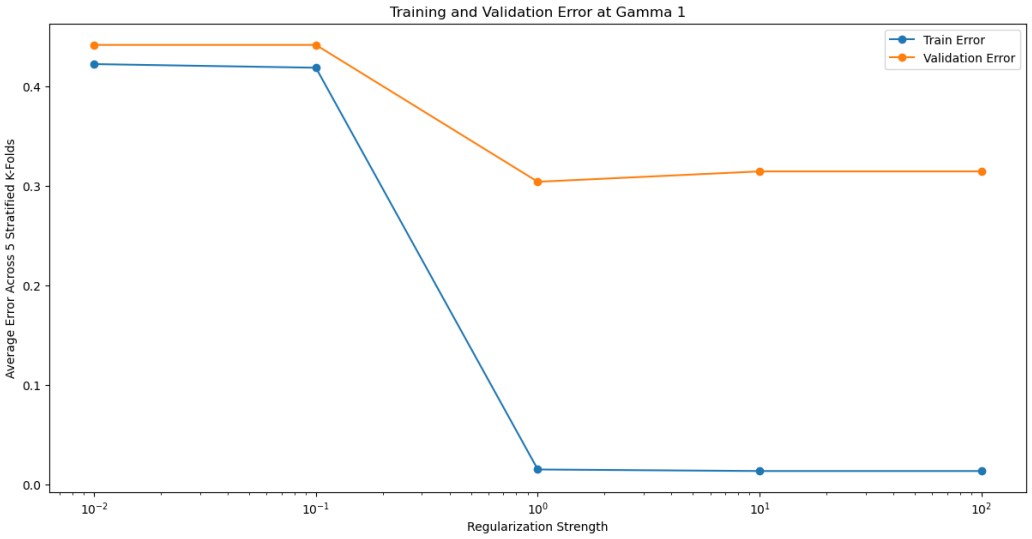
Varying γ Values

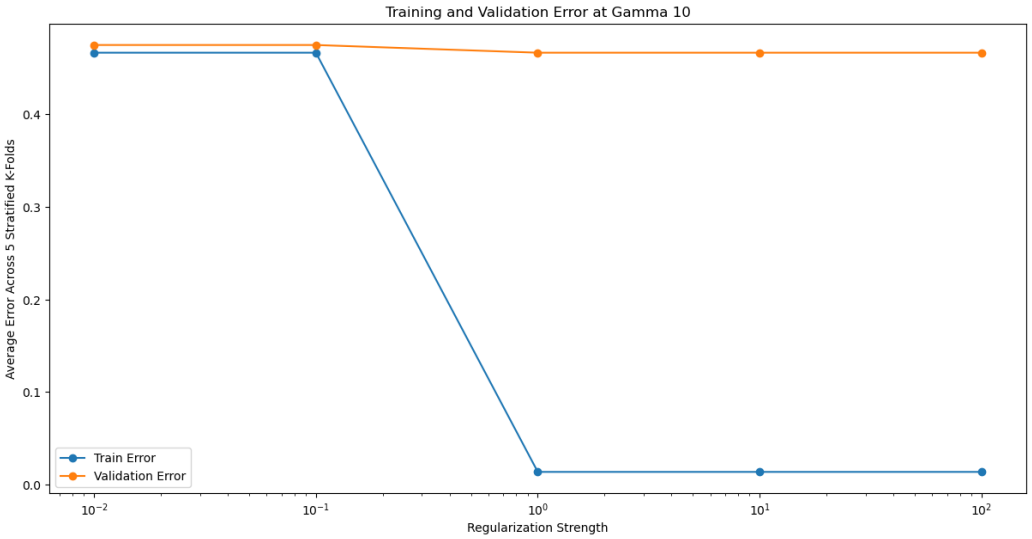
Below, I show figures demonstrating the average error on 5 Stratified Folds across varying γ values:




Training and Validation Error at Gamma 0.01
Train Error
Validation Error
Average Error Across 5 Stratified K-Folds
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05
10^{-2}
10^{-1}
10^{0}
10^{1}
10^{2}
Regularization Strength


Training and Validation Error at Gamma 0.1
Train Error
Validation Error
Average Error Across 5 Stratified K-Folds
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
10^{-2}
10^{-1}
10^{0}
10^{1}
10^{2}
Regularization Strength





Summary of γ - C sweep:

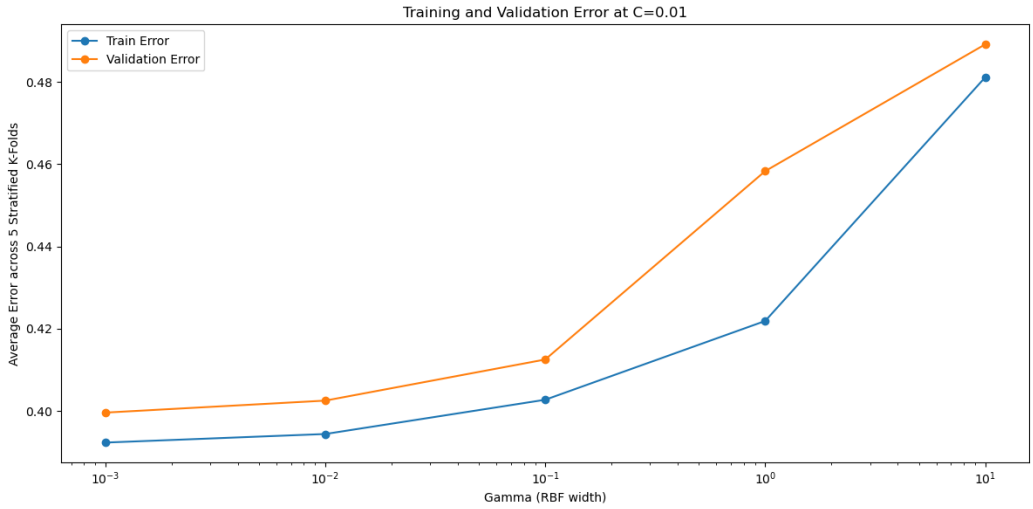
- $\gamma = 0.001$**:** Training and validation errors remain high for low regularization strengths, indicating underfitting. Increasing the regularization strength (C) reduces both errors steadily, suggesting the model benefits from fitting the data more aggressively.
- $\gamma = 0.01$**:** Similar underfitting at small C, but performance improves noticeably at $C \approx 10$, where validation error reaches its minimum.

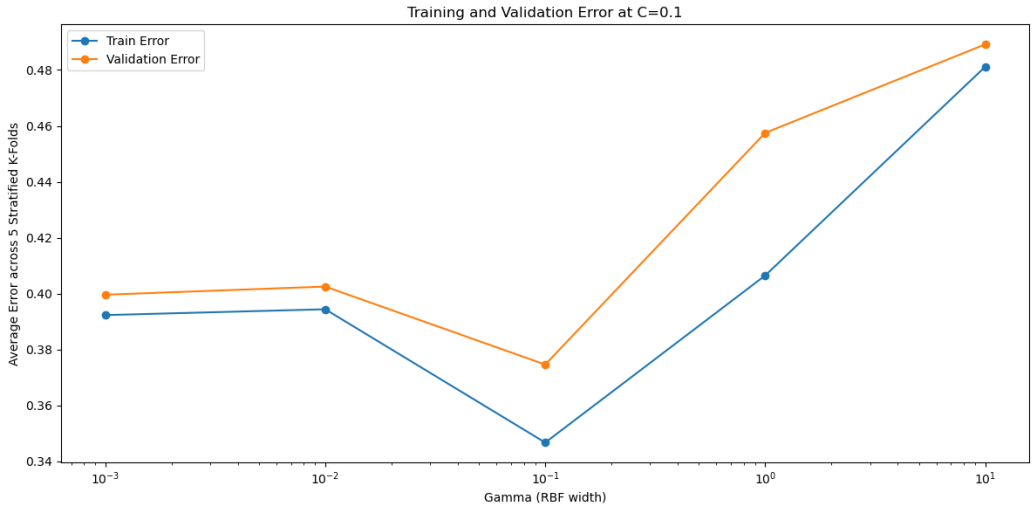
Extremely large C slightly increases validation error, hinting at mild overfitting.

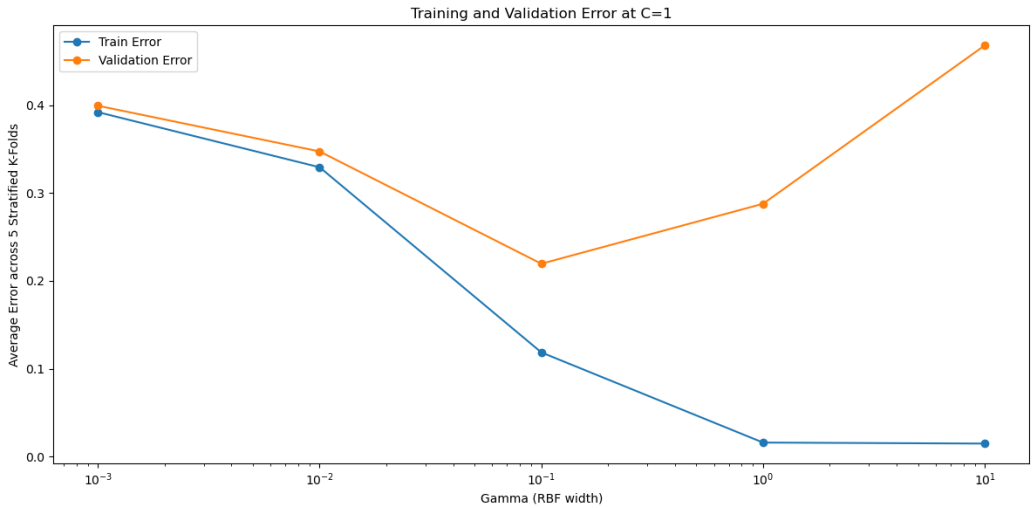
- $\gamma = 0.1$**:** Clear improvement in both training and validation errors as C increases, with the lowest validation error near $C \approx 1$. Validation error begins to rise for very large C, showing the start of overfitting.
- $\gamma = 1$**:** Sharp drop in training error at $C \approx 1$, but validation error remains flat beyond this point. The model begins to overfit heavily at high C.
- $\gamma = 10$**:** Training error reaches near-zero at $C \approx 1$, but validation error stays high across all C values. This indicates severe overfitting caused by the excessively localized RBF kernel.
- **Conclusion:** Moderate γ (around 0.01 to 0.1) with regularization strengths in the range $C \approx 1$ to 10 tend to provide the best balance between bias and variance. Very high γ or very low C generally degrade performance.

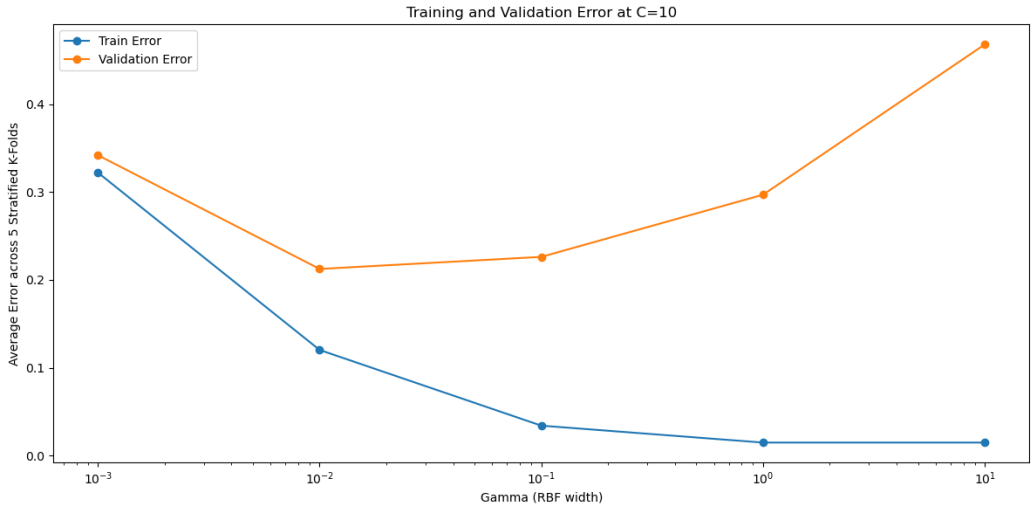
Varying C Values

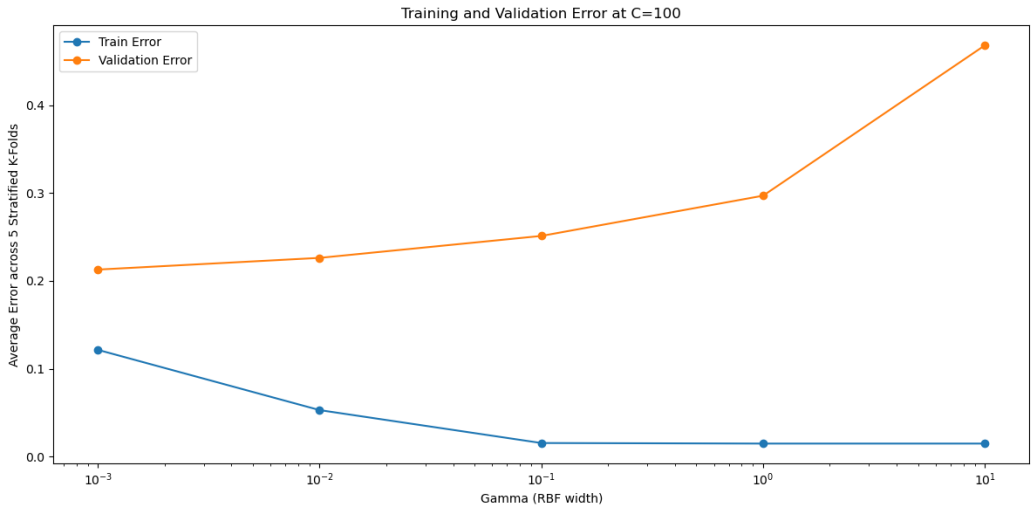
Below, I show figures demonstrating the average error on 5 Stratified Folds across varying C values:




Training and Validation Error at C=0.1
Train Error
Validation Error
Average Error across 5 Stratified K-Folds
0.48
0.46
0.44
0.42
0.40
0.38
0.36
0.34
10^{-3}
10^{-2}
10^{-1}
10^{0}
10^{1}
Gamma (RBF width)


Training and Validation Error at C=1
Train Error
Validation Error
Average Error across 5 Stratified K-Folds
0.4
0.3
0.2
0.1
0.0
10^{-3}
10^{-2}
10^{-1}
10^{0}
10^{1}
Gamma (RBF width)





Summary of C - γ Sweep:

- **Low C (0.01, 0.1):** The model struggles to fit the data across all values of γ, with both training and validation errors staying high. There is a small improvement at moderate γ, but overall the capacity is too limited.
- **Moderate C (1, 10):** Performance improves noticeably, with a clear sweet spot around $\gamma \approx 10^{-1}$. Here, validation error is at its lowest and

training error is still reasonable. Very small γ values underfit, while very large ones start to overfit.

- **High C (100):** The model can almost perfectly fit the training data for moderate and large γ, but validation accuracy only stays high at smaller to moderate γ. At very large γ, the model clearly overfits.
- **Conclusion:** The best balance between bias and variance happens with a moderate C and a moderate γ. Going too far in either direction—whether too small C or too large γ—leads to worse performance.

Random Forests

Following SVM, I implemented a Random Forest classifier to compare against the SVM.

Random Forests are an ensemble learning method that combine the predictions of multiple decision trees to improve generalization and reduce overfitting.

Hyperparameter Tuning

I varied two key hyperparameters:

- **n_estimators:** Number of trees in the forest. The more trees a forest has, the better its accuracy. However, increasing the number of trees also increases the risk of overfitting the model on training set and it eventually fails to classify test observations correctly. Hence, I tested a range of values, such as, {10, 50, 100, 200, 300}.
- **max_depth:** Maximum depth of each tree. This determines the number of splits each tree is allowed to make. Higher number of splits means more accuracy. However, this, again, risks overfitting. Hence, I tested a range of values, such as, {3, 5, 10, 15, 20}.

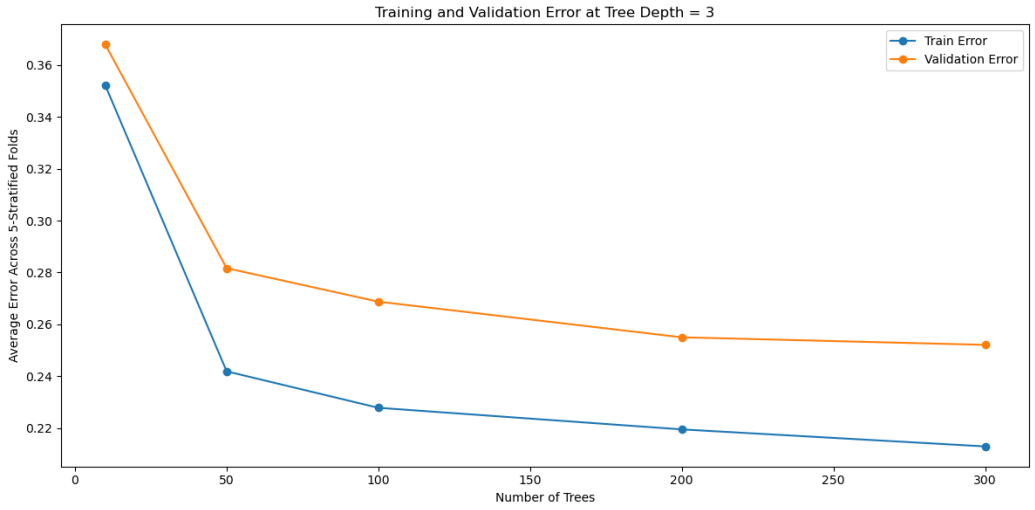
Like SVMs, I performed 5-fold Stratified Cross-Validation for each combination of parameters. For every fold, I recorded both training and validation accuracy, and computed the average error across folds.[6] This helped assess the impact of model complexity on over and underfitting.

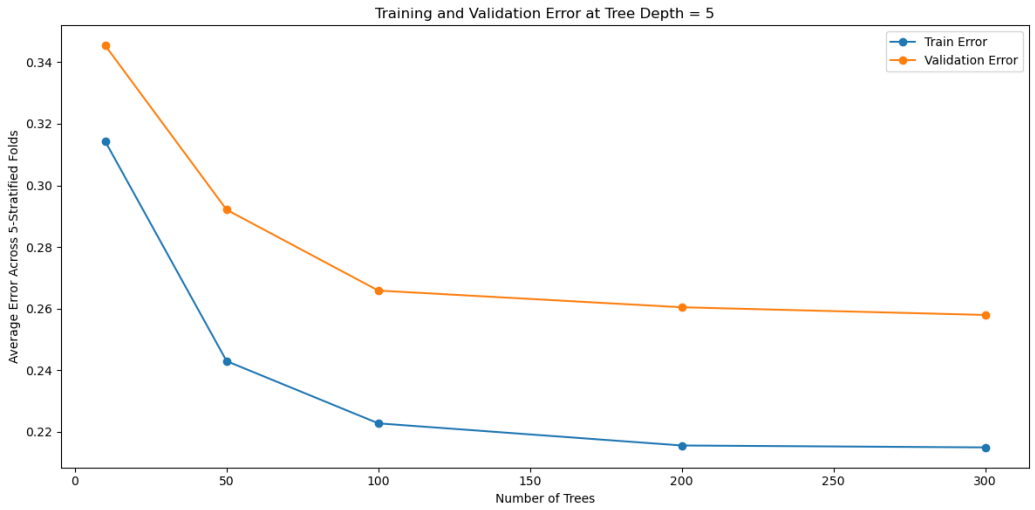
[6] Average Error = 1 - Accuracy

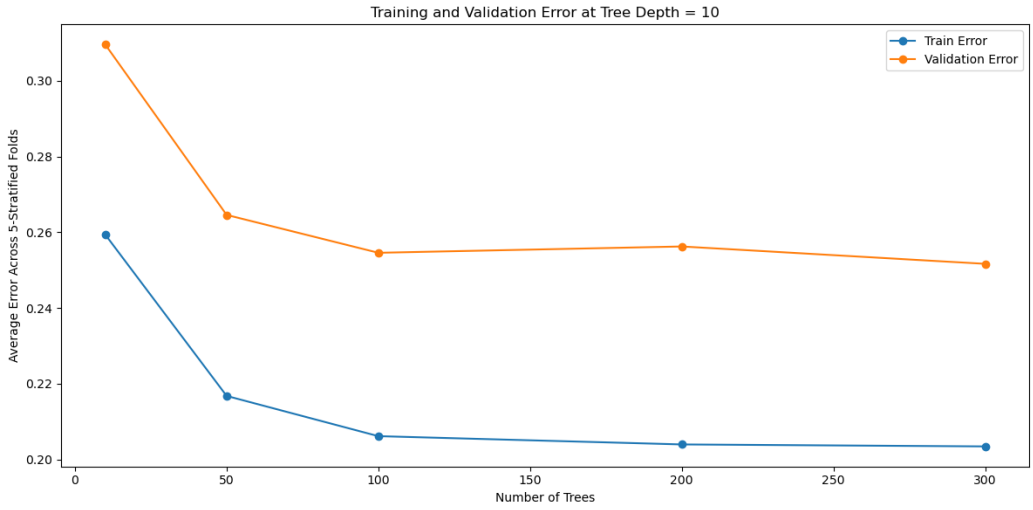
Results and Observations

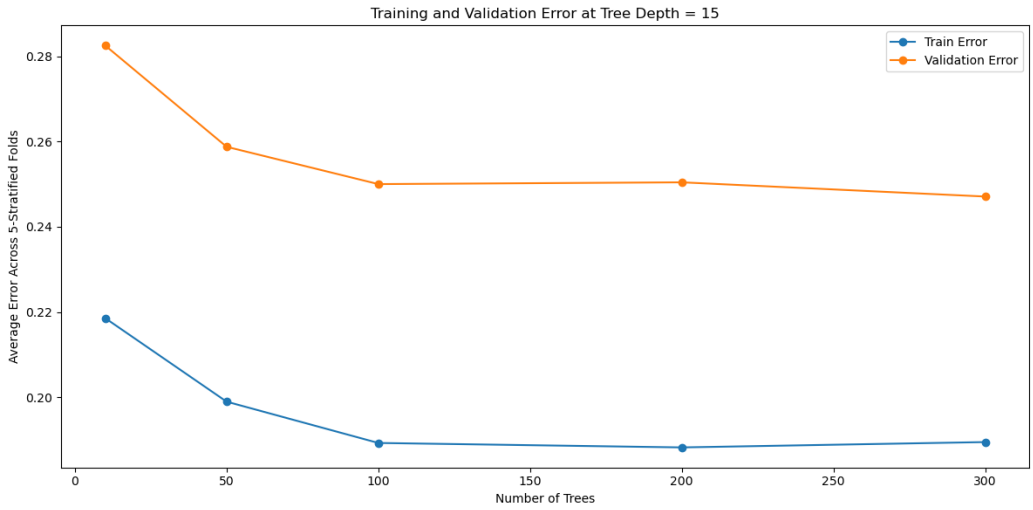
Varying Tree Depths

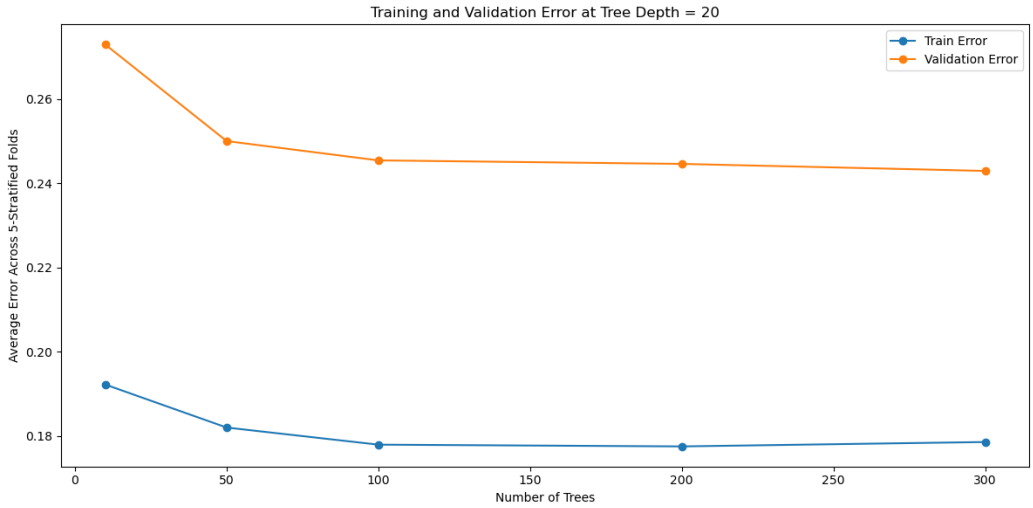
Below, I show figures demonstrating the average error on 5 Stratified Folds across varying tree depths:






Training and Validation Error at Tree Depth = 10
Train Error
Validation Error
Average Error Across 5-Stratified Folds
0.30
0.28
0.26
0.24
0.22
0.20
0
50
100
150
200
250
300
Number of Trees

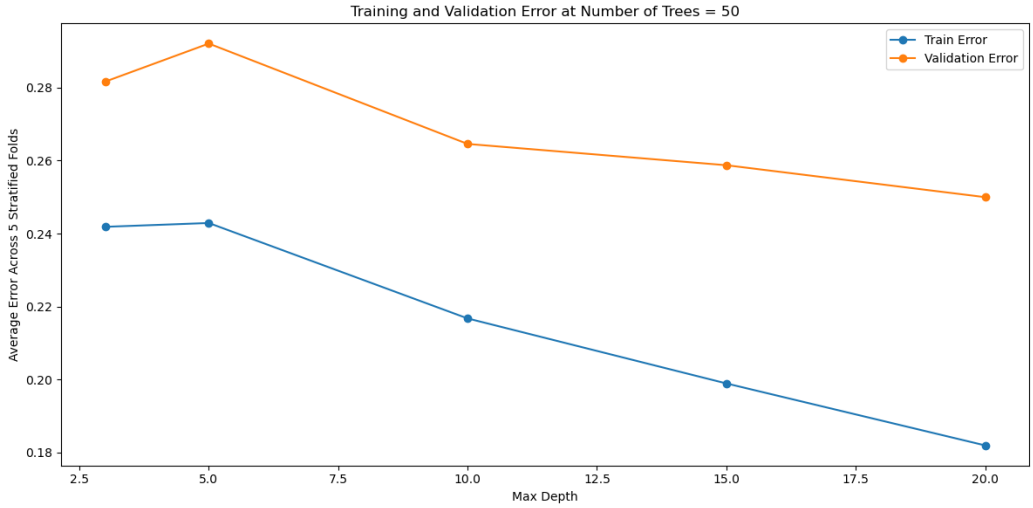

Training and Validation Error at Tree Depth = 15
Train Error
Validation Error
Average Error Across 5-Stratified Folds
0.28
0.26
0.24
0.22
0.20
0
50
100
150
200
250
300
Number of Trees

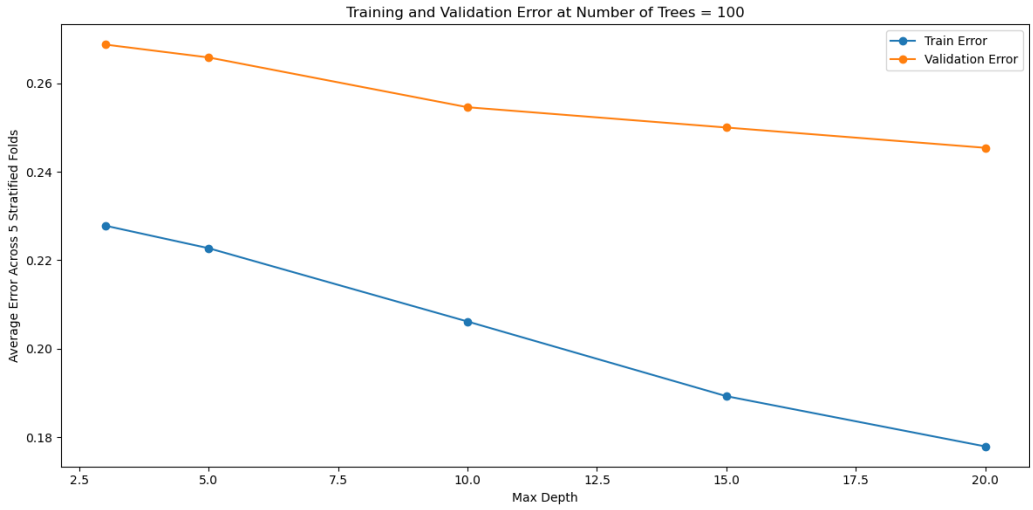


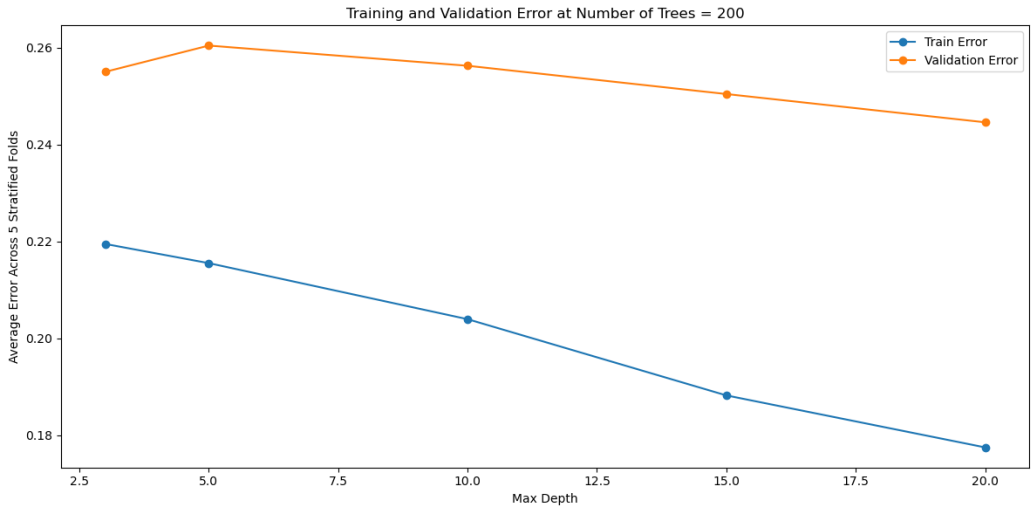
Summary of Tree Depth - Number of Trees Sweep

- **Depth = 3:** Both training and validation errors drop steadily as the number of trees increases, with no signs of overfitting up to 300 trees. Gains in error reduction diminish beyond ≈ 100 trees.
- **Depth = 5:** Similar trend to depth 3, but starting from slightly lower error values. Validation error plateaus after ≈ 200 trees.
- **Depth = 10:** Training error decreases quickly, reaching a plateau near 150 trees. Validation error improves initially but shows a slight upward creep for large tree counts, hinting at mild overfitting.
- **Depth = 15:** Training error reaches very low levels by 100 trees and plateaus. Validation error plateaus between ≈ 100 and ≈ 200 trees. It then reduces slightly at 300 trees, while training error increases.
- **Depth = 20:** Training error is lowest across all depths, but validation error remains almost flat after 50–100 trees. This suggests the model is near the limit of performance and may be overfitting at deeper depths.
- **Conclusion:** Increasing tree depth reduces bias but risks overfitting at high depths. For this dataset, depths between 5 and 15 with ≈ 100 trees provide a strong trade-off between bias and variance.

Varying Number of Trees







Summary of Tree Depth–Number of Trees Sweep

- **10 Trees:** Increasing depth consistently lowers both training and validation errors. The gap between them stays moderate, suggesting little overfitting even at high depths.
- **50 Trees:** Training error decreases with depth, but validation error peaks slightly at depth = 5 before dropping again. Overall performance improves at deeper depths, with a smaller train–validation gap.
- **100 Trees:** Both errors decline steadily with depth, with the valida-

tion curve flattening at higher depths. The small gap suggests stable generalization.

- **200 Trees:** Training error drops steadily, while validation error changes very little beyond depth = 10. The consistent gap implies a mild risk of overfitting at the deepest levels.
- **300 Trees:** Patterns are similar to 200 trees, i.e., training error continues to fall, while validation error remains almost flat past depth = 10. Benefits from deeper trees are marginal for validation.
- **Conclusion:** More trees and greater depth both reduce bias (training error) but do not always improve validation performance. For this dataset, depths of 10–15 with 100–200 trees provide a good balance between bias reduction and overfitting risk.

Multi Layer Perceptrons

Lastly, I implemented a deep neural network using a multilayer perceptron (MLP)[7] architecture to classify sentiment from Bag-of-Words features. The network used fully connected (`Dense`) layers with ReLU activations for the hidden layers and a sigmoid activation in the output layer to produce a probability for the classes.

Hyperparameters

For this model, I varied the two hyperparameters, the number of Hidden Layers and Learning Rate, systematically:

- **Number of hidden layers:** {1, 2, 3, 4}
- **Learning rate:** $\{1\times10^{-4}, 1\times10^{-3}, 1\times10^{-2}\}$, using the Adam optimizer. [8]

[7] MLP is a type of Neural Network which consists of multiple layers of nodes, including an input layer, one or more hidden layers, and an output layer.

[8] Aaptive Momemnt Estimation (Adam) adjusts the learning rate for each parameter individually and automatically.

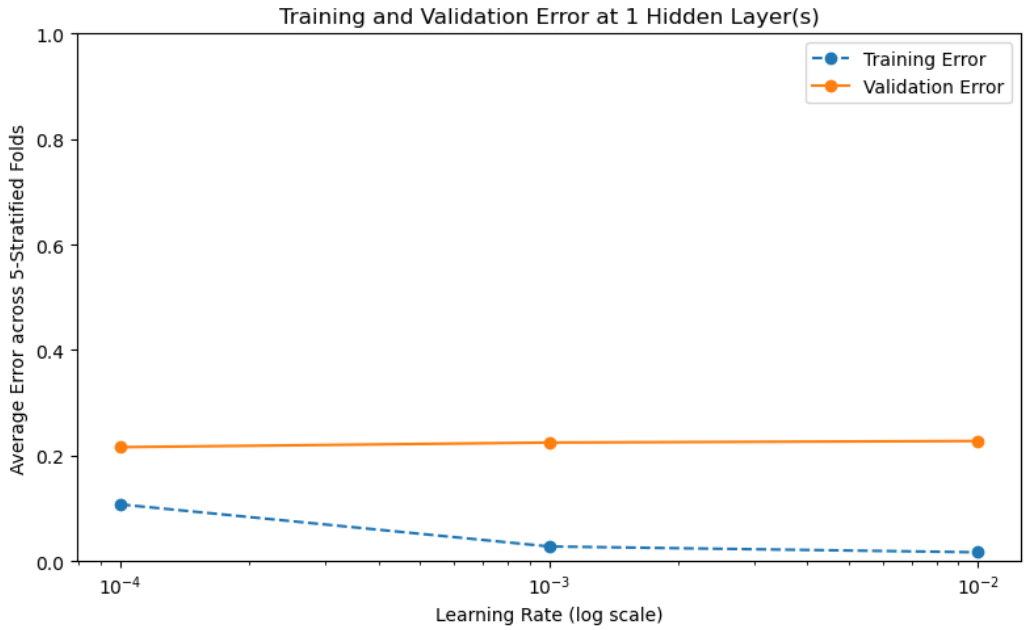
Each hidden layer consisted of 128 neurons. The binary cross-entropy loss function was used, and all models were trained for 15 epochs with a batch size of 32.[9]

Cross-Validation

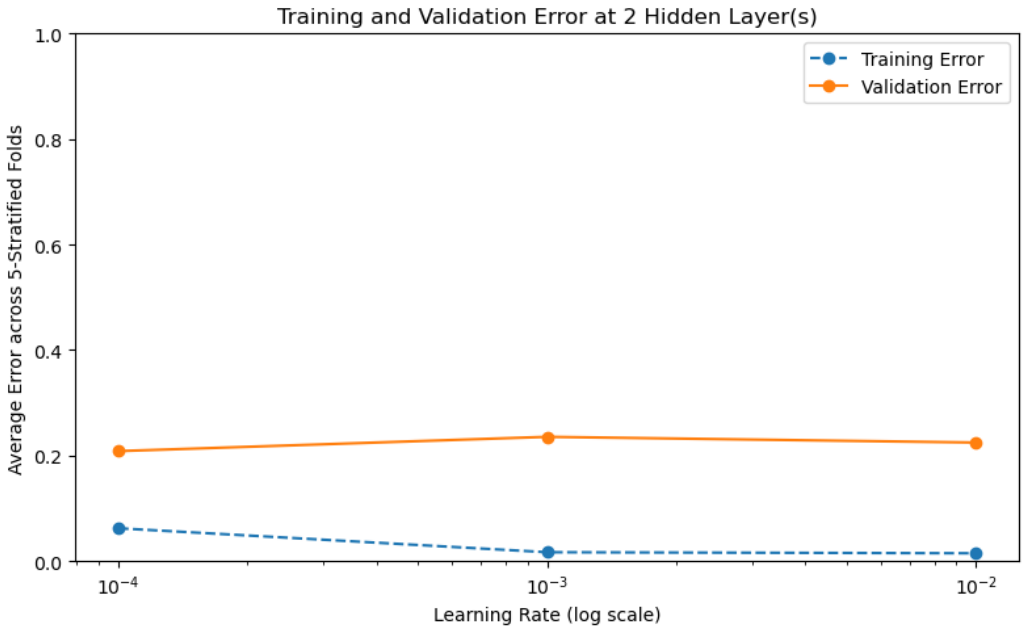
I used 5-fold Stratified Cross-Validation to evaluate each configuration.

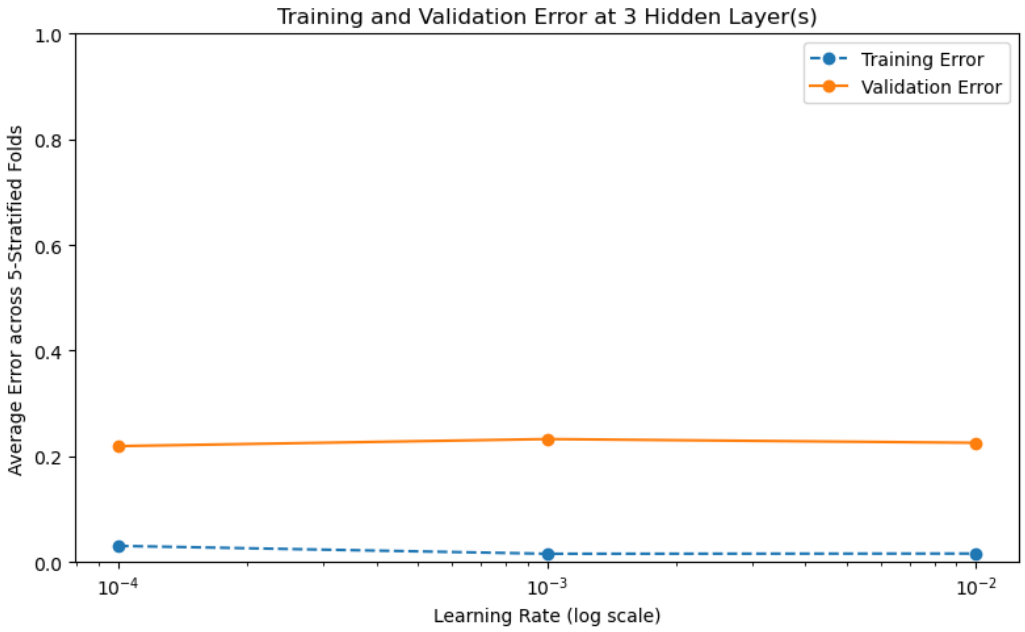
Results and Observations

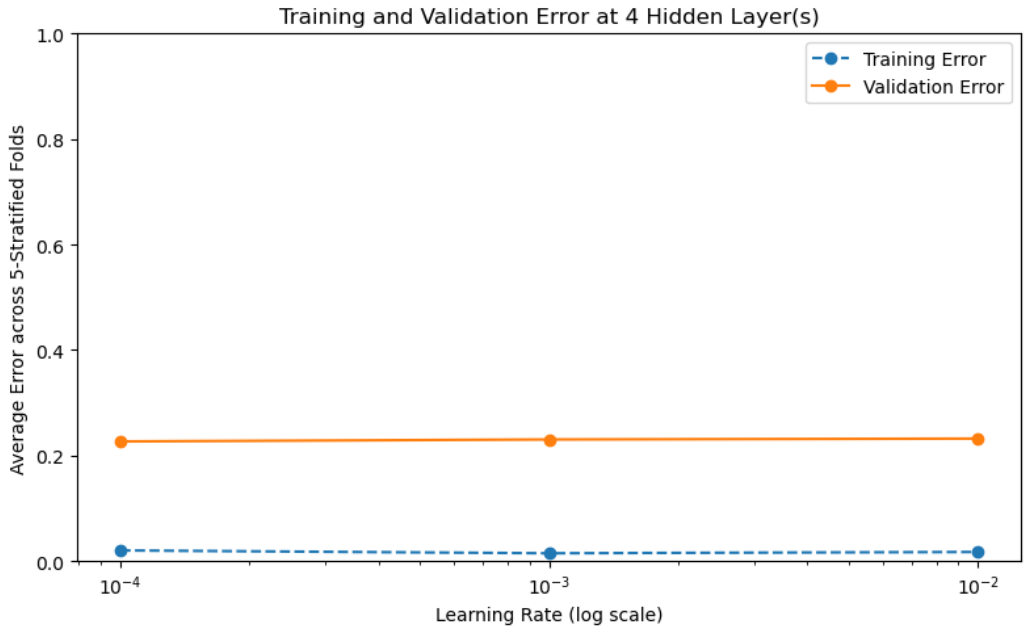
Varying The Number of Hidden Layers:



[9]Epochs mean that the model will see the entire training set 15 times in total (with weights updated). Similarly, batch size means that the model takes n number of observations from the dataset, compute predictions/loss/weight updates and then moves onto the next n batch size


Training and Validation Error at 2 Hidden Layer(s)
Training Error
Validation Error
Average Error across 5-Stratified Folds
1.0
0.8
0.6
0.4
0.2
0.0
10^{-4}
10^{-3}
10^{-2}
Learning Rate (log scale)


Training and Validation Error at 3 Hidden Layer(s)
Training Error
Validation Error
Average Error across 5-Stratified Folds
1.0
0.8
0.6
0.4
0.2
0.0
10^{-4}
10^{-3}
10^{-2}
Learning Rate (log scale)

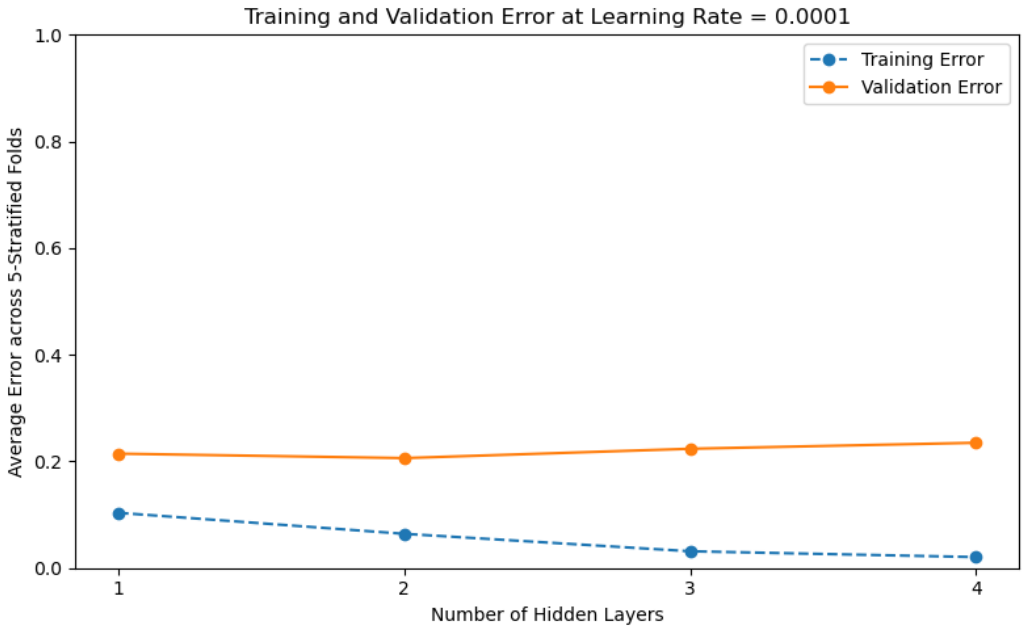


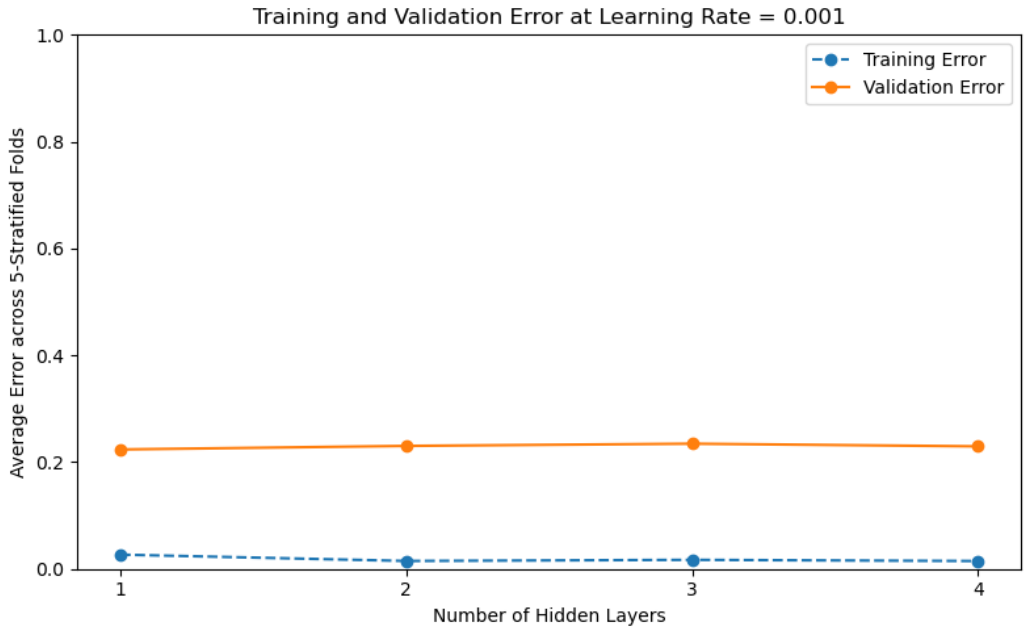
Summary of Hidden Layers - Learning Rate Sweep

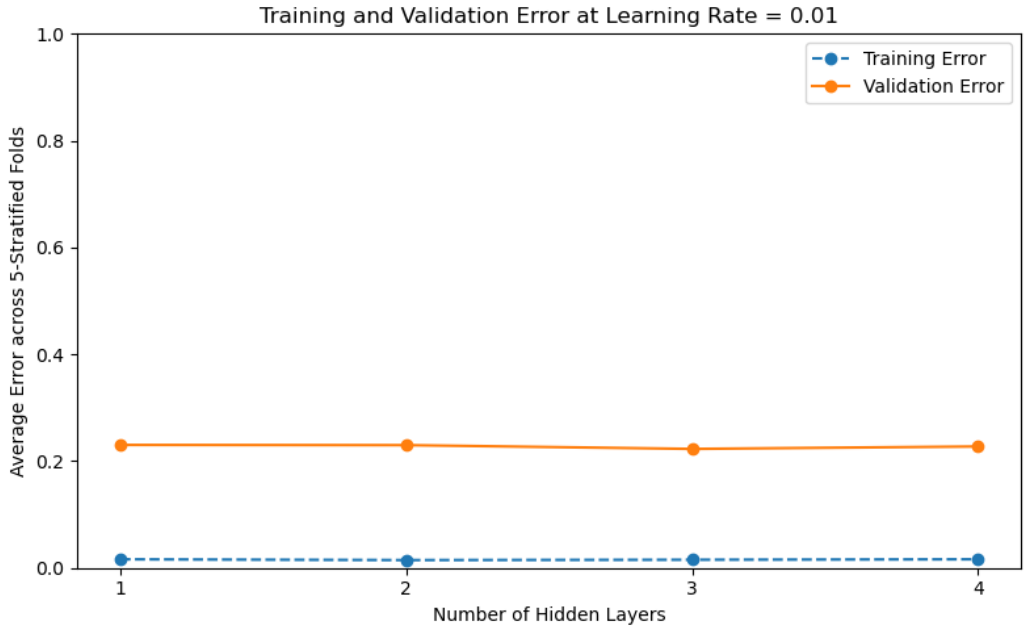
- **1 Hidden Layer:** Training error decreases steadily as the learning rate increases, but validation error remains nearly constant around 0.22. This suggests the model is already fitting well, and changes in learning rate have little effect on generalization.
- **2 Hidden Layers:** Similar to 1 layer, but with slightly lower validation error at the smallest learning rate. Training error drops rapidly with higher learning rates, though validation trends remain flat.
- **3 Hidden Layers:** Training error is very low across all learning rates. Validation error shows minimal variation, indicating that model depth is sufficient to capture the patterns regardless of learning rate choice.
- **4 Hidden Layers:** Training error is nearly zero for all learning rates, and validation error remains stable at ≈ 0.23. Adding more depth does not improve generalization.
- **Conclusion:** Increasing depth beyond 2 layers provides little to no gain in validation performance. Learning rate has minimal influence on

validation error in this setup, although it slightly accelerates training convergence.

Varying Learning Rate:







Summary of Learning Rate - Hidden Layer Sweep:

- **Low learning rate** (10^{-4})**:** Training error decreases with more hidden layers, but validation error remains consistently higher, indicating

slight underfitting. The gap between training and validation error increases with more layers.

- **Moderate learning rate (10^{-3}):** Both training and validation errors are low and stable across all depths. This learning rate provides the best trade-off, suggesting that 10^{-3} is close to the optimal value for this task.
- **High learning rate (10^{-2}):** Training error remains very low, but validation error shows no significant improvement over 10^{-3}. Slight fluctuations in validation error suggest potential instability at higher learning rates.
- **Conclusion:** A learning rate of 10^{-3} yields the best generalization across different network depths. Increasing the number of hidden layers reduces training error, but validation error saturates, suggesting diminishing returns from additional depth.

Best Model

The best model in this case is the Multi Layer Perceptrons, where the average error on training and validation sets is $\approx$0 and $\approx$0.2, respectively. This is lower than the error achieved using Support Vector Machines and Random Forests. Hence, MLP is the best performing model for our purposes.

MLP outperformed SVM and Random Forests here because MLPs are capable of learning intricate patterns and detecting non-linear boundaries more efficiently.

Test Performance Results:

To evaluate the performance of the best performing model, i.e., Multi Layer Perceptrons, I fed it 600 test reviews for classification. The MLP correctly classified all the reviews, achieving an accuracy of 100%.

Conclusion and Next Steps

In conclusion, I was given a dataset consisting of customer reviews on various products from three different domains. I first standardized the dataset to include only text, characters from the english language, etc. I then performed feature representation using TF-IDF, where I identified the 1,000 most commonly occurring words in the dataset. I then performed a Bag-of-Words (BoW) transformation on the data, where I assigned every data observation 0 or 1 based on whether they contained any of the most important 1,000 words. Lastly, I trained three different classifiers, namely SVM, Random Forests, and MLP, on the BoW feature data. I then validated these classifiers using the original training set. Once I had identified the best performing model, i.e., MLP, I ran it on an unseen test dataset of 600 reviews and it classified all data points correctly.

The next step would be to generalize this model for larger datasets; our dataset was relatively smaller, with only 2,400 observations. Additionally, we can, perhaps, analyze multilingual data for broader model applicability.