Introduction to Machine Learning (CS 135)
Project 02
Name: Obafemi Jinadu | Tufts ID: 1406297

Part one: **Classifying Reviews Sentiment with Bag-of-Words Features**
An outline of how I went about the project:
- Exploratory analysis
- Obtaining a Baseline Model
- Preprocessing Pipeline
- Logistic Regression Model
- MLPClassifier Model
- Support Vector Machine Classifier (SVC)
- Summary of Best Performing Model
- Cross-Validation and Leaderboard Comparison

Exploratory Analysis

I started off by carrying out some exploratory analysis on the dataset of single-sentence reviews, collected from three domains i.e., imdb.com, amazon.com, and yelp.com, the description of the dataset is given below:

- x_train_df – (2400 by 2) where we have the first column representing the domain/website name and the second column is the review in text. Meaning we 2400 single sentence reviews across the 3 domains.
- y_train_df – (2400 by 1) this dataframe contains a binary label indicating sentiment, 1 for positive and 0 for negative sentiments. And we have 2,400 entries corresponding to the reviews in the x_train_df.

For better analysis, I merged both dataframes i.e., x_train_df and y-train_df. After merging, I then split the merged dataframe into 3, and each of the 3 dataframes represents a domain. A summary of the exploration is given in the table below:

Table 1: Results of splitting dataframe into respective domains (training data)

Domain	**Datapoints/Single-sentence Text Review Count**	**Sentiment Breakdown**
Amazon	800	400 positive and 400 negative sentiments
Imdb	800	400 positive and 400 negative sentiments
Yelp	800	400 positive and 400 negative sentiments

From the above distribution, we observe that the training dataset is equally distributed on all counts i.e., equal splits of 800 per domain and within each domain equal counts of positive and negative sentiment labels.

For the testing data:

- x_test – (600 by 2) where we have the first column representing the domain and the second column representing the single-sentence review of which we have 600 of these datapoints.

Further exploration was also carried out on the testing data, since our target (objective) is to find its binary label representing each datapoints sentiment, we do not have a y_test (as it is what the project is to predict). The x_test data is also split into 3 dataframes where each dataframe represents a domain. A summary of the exploration is given in the table below:

Table 2: Results of splitting dataframe into respective domains (testing data)

Domain	Datapoints/Single-sentence Text Review Count
Amazon	200
Imdb	200
Yelp	200

Similarly, from the testing data distribution, we observe that the dataset is equally distributed across each domain with each domain having 200 reviews and the 3 domains summing up to the datapoint of 600 entries.

The performance metrics considered for all models leverages 10-fold cross-validation with stratified sampling. Also, all results in the development of this project were obtained using stratified 10-fold cross-validation and the performance metrics considered are:

- Error rate
- Logistic Loss/Cross-Entropy Loss
- Area Under the Receiver Operating Characteristic (AUROC)

The stratified kfold cross-validation was used because it ensures that the proportion of the feature of interest (sentiment label) during each train, test split (fold) is the same as the original/complete dataset. This ensures that no value is over/under-represented in the training and test set, and it allows for more reliable performance estimates. An alternative to avoiding over/under-representation of the feature of interest would be to use the regular Kfold and set the shuffle argument to true. Both approaches produced almost identical results.

Baseline Model

A simple baseline model was built, which is a model built with defaults i.e., without any pre-processing, or model hyper-parameter tuning with grid search or any other means. The baseline model serves as a means of comparison to other models to be explored.

A stratified 10-fold cross-validation was used, to get a more reliable estimate of the training and testing (true) errors. In this case, since we have 2,400 data points/rows. The stratified 10-fold cross-validation is essentially 10 different models with each model having 2,160 training data and 240 testing data and, in the end, the averages of the model performance metrics are obtained.

This average of the 10 different models would give a more accurate idea of how the model would do in the real world as we have a more diverse and robust range of data distribution for each model to be averaged.

Also, for each model, the train and test data preserve the original label proportion (as explained with the stratified kfold cross-validation). To build the baseline model, the default CountVectorizer() was used to generate the bag of words (BoW) features which produced 4,510 features serving as the data which the default logistic regression model was built.
The logistic regression model was used as the baseline because of its simplicity.
The following results (averaged over 10-fold cross-validation) were obtained for the baseline model:

Table 3: Baseline Model Results

	Training set			**Testing set**			**Leaderboard**	
Approach	Error Rate	Log-Loss	AUROC	Error Rate	Log-Loss	AUROC	Error Rate	AUROC
Baseline	0.0204	0.3902	0.9746	0.175	0.4887	0.8898	0.1833	0.8896

The baseline performance values were not significantly different from the leaderboard perform ance, indicating there is no overfitting. However, further exploration was carried out to further i mprove the model.

1) <u>Preprocessing Pipeline</u>

It is worth mentioning that the Support Vector Classifier model along with the TfidfVectorizer worked best with the pipeline created. The preprocessing pipeline steps are outlined below:

- Step 1 - Sentence Contraction and regular expression word substitution
- Step 2 - Parts-of-speech (POS) Lemmatization
- Step 3 - Building a tokenizer function for TfidfVectorizer ()

<u>Sentence Contraction and regular expression word substitution</u>

This preprocessing step starts by expanding contractions, contractions are words or combinations of words shortened by dropping letters and replacing them with apostrophes e.g., “isn’t”, “we’ve” their expanded respective versions are “is not” and “we have”. This was done to produce more reasonable word tokens. For example, “we’ve” won’t be split into “we” and “ve” as “ve” does not really mean anything on its own compared to its expanded version “have”.
This was implemented using the contractions library in python.
The next thing that was done was to replace certain texts using regular expressions. This was done to reduce the bag of word features without compromising data by tagging similar texts to a word some examples are given below:

- If a text starts with a dollar sign and contains a number, it would be tagged as ” money”
- If a text starts with a number and ends with “pm”, “am”, “mins”, “hour”, “sec” etc. it would be tagged as “time”.
- If a text starts with a number and ends with “lb”, “g “ or ”kg” etc. it would be tagged as “weight”.
- If a text of numbers starts with either 1 or 2 and contains 2 more digits (making 4 altogether) it would be tagged as “year”

This also accounted for some slang like “bucks” tagging this slang to money.
This preprocessing step reduced the bag of words (BoW) features obtained from the inverse document frequency (TfIdfVectorizer) to 4,418.

Examples of some of these sentence transformations copied as-is from the preprocessed list and original training_text_list are given in the table below:

Table 3: Preprocessing step 1 – Output from Applying Word Contraction and Regular Expressions

Original Sentence	Transformed Sentence
'Waste of 13 bucks.'	'Waste of money'
'$50 Down the drain.'	'money Down the drain'
'At around time I buy it , at around time I start to watch	"At around 4 pm I bought it, at around 8pm I started to watch ”

Parts-of-speech (POS) Lemmatization

Lemmatization was applied to group together different inflected forms of a word to their root/common word (by default, lemmatization just tags nouns). For, better lemmatization outputs, parts-of-speech tagging was used as lemmatization leverages context information to perform these word mappings. For example, the words “going”, “gone” and “went” would be mapped to “go”. Unlike stemming that essentially just chops off parts of the word to achieve their root word mappings. The POS lemmatization was implemented using the WordnetLemmatizer from the nltk package.

Examples of some of these sentence transformations after preprocessing steps one and two were applied, copied as-is from the preprocessed list and original training_text_list are given in the table below:

Table 4: Preprocessing step 1 – POS Lemmatization Output

Original Sentence	Transformed Sentence
"THAT one didn't work either."	'THAT one do not work either'
'Not impressed.'	'Not impress',
"Don't make the same mistake I did."	'Do not make the same mistake I do'

This approach in addition to the first step further reduced the bag of words features to 3,870.

Building a tokenizer function for TfidfVectorizer()

The tokenizer function serves as the tokenizer argument to the TfidfVectorizer, this function implemented dictates how the words are tokenizer. The function takes in the training_text_list and first uses regular expression to parse words separated by space and replaces anything that is not a letter, number, hyphen, or a dollar sign. It then unifies every word to lower case and splits every word into individual items of a list.

It then prunes out words that are less than two letters. It finally stems the output using the PorterStemmer library. This reduced the bag of word features to 3,524. The model performance was tested at each preprocessing stage using SVM Classifier (SVC) and is captured in table below:

Table 5: Comparing Results of Preprocessing Steps on Base SVC Model

		Training set			Testing set			Leaderboard	
Approach	BoW Feature Size	Error Rate	Log-Loss	AUROC	Error Rate	Log-Loss	AUROC	Error Rate	AUROC
Baseline	4,510	0.0204	0.3902	0.9746	0.175	0.4887	0.8898	0.1833	0.8896
Step 1	4,418	0.00565	0.0357	0.9989	0.175	0.4041	0.9009	0.17167	0.91162
Step 1 + Step 2	3,870	0.00694	0.04405	0.9987	0.1642	0.3857	0.9103	0.175	0.91045
Step 1 + Step 2 + Step 3	3,524	0.00829	0.04733	0.9986	0.1599	0.39063	0.9103	0.16167	0.9141

- Step 1 - Sentence Contraction and regular expression word substitution
- Step 2 - Parts-of-speech (POS) Lemmatization
- Step 3 - Building a tokenizer function for TfidfVectorizer ()

From table 5, For the cross-validation results, we observe that leveraging all the preprocessing s teps i.e., steps 1, 2, and 3 on the base SVC model gives the overall best performance, most gene ralized performance on the cross-validation training and testing results, and on the leaderboard as well.
On the training set, we observe that as we go through each step on the table the performance g radually declines, as the error rate and log-loss gradually increase and the AUROC reduces. Whil e on the testing set, the reverse happens (the model performance improves). This implies that t he effect of overfitting gradually reduces as we gradually increase the preprocessing steps appli ed and the model becomes more generalized.
On the leaderboard, we see that applying all preprocessing steps gives the best leaderboard res ult. However, using step 1 alone seems to give better leaderboard results compared to when st eps 1 and 2 are applied. This inconsistency could be due to a higher tendency of overfitting of t hese models.

Also, n_gram and min_df parameters were explored on the preprocessed data.

Increasing the min_df i.e., dropping out words based on how rarely they occur in documents. For example, when min_df is 2, we drop any number that occurs in less than two documents and so on. The min_df value was varied from 2 to 5 on the SVC base model. We see that increasing the min_df reduces the number of features in the BoW feature vector and as the number of features reduces, the training error rate also increased, the testing error rate increased in this trend as well. Underfitting occurres as min_df value was increased. Therefore the default min_df of 1 was used.

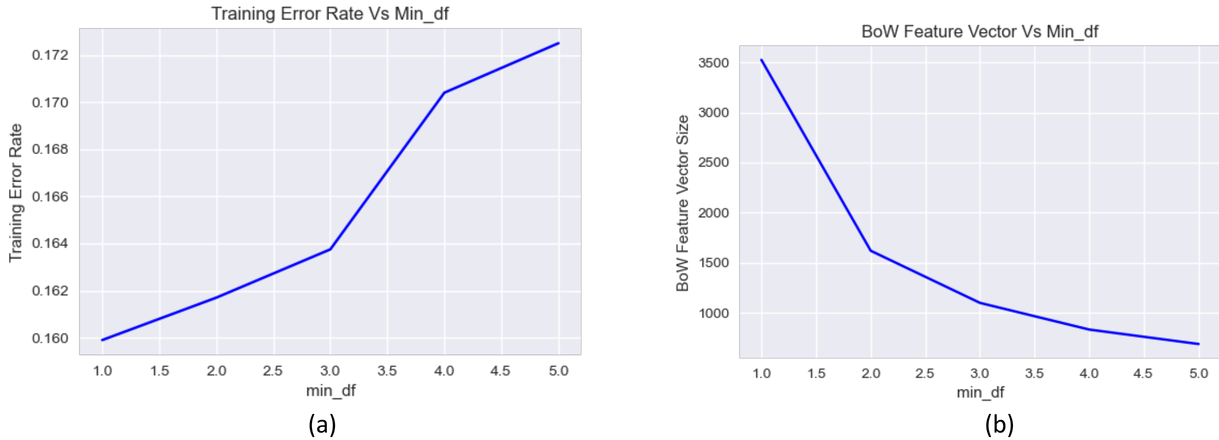


Fig. 1 a) Training Error Rate Vs Min_df b) BoW Feature Vector Size Vs. Min_df

The n_gram value was also empirically tested. Along with the default unigrams, bigrams and trigrams were also considered. They produced promising results. However, not as good as the unigram as the bigrams and trigrams seem to overfit on the data and added more complexity by producing large feature vectors. The results are captured in the table below:

Table 5: Comparing Results of n_grams

			Training set			**Testing set**			**Leaderboard**	
N_gram	BoW Feature Size	Min_df	Error Rate	Log-Loss	AUROC	Error Rate	Log-Loss	AUROC	Error Rate	AUROC
(1,1)	3,524	1	0.00829	0.04733	0.9986	0.1599	0.39063	0.9103	0.16167	0.9141
(1, 2)	4,597	2	0.000417	0.0085	0.9997	0.1587	0.3777	0.9124	0.175	0.90626
(1, 3)	5,985	2	0.000417	0.0076	0.9997	0.1604	0.38014	0.9113	0.172	0.909

From the table, we see that the unigram gives a better error rate and AUROC performance on the leaderboard and gives almost the same results as the bigram and trigram on the testing set. However, on the training set, the bigram and trigram perform better which indicates some overfitting on the models built on the bigram and trigram. Due to the apparent overfitting of the bigrams and trigrams along with their complexity (as unigram as the most compact feature vector size). I decided to go with unigrams and min_df of 1.

2) Logistic Regression

To begin this phase of the project, the bag of words (BoW) feature vector obtained from the preprocessing pipeline explored above was used to build a default logistic regression model. To evaluate its performance on training and validation data, a stratified 10-fold cross-validation was used. The results are given in the table below:

Table 6: Comparing Preprocessed BoW on Base Logistic Regression with Baseline

		Training set			**Testing set**			**Leaderboard**	
Approach	BoW Feature Size	Error Rate	Log-Loss	AUROC	Error Rate	Log-Loss	AUROC	Error Rate	AUROC
Baseline	4,510	0.0204	0.3902	0.9746	0.175	0.4887	0.8898	0.1833	0.8896
Base Logistic Regression Model	3,524	0.06722	0.3892	0.9724	0.1725	0.3892	0.90168	0.17833	0.90392

Comparing the output of the logistic regression model built on the processed BoW pipeline, to the model built on the unprocessed BoW feature vector, we see an improvement in the testing set data and a slight decline in the performance on the training set. This indicates that the model built on the processed BoW pipeline generalizes better on new examples than the unprocessed version. These performance values were not significantly different from the leaderboard performance, indicating there is no overfitting. However, further exploration was carried out to further improve the model.

Hyper-Parameter Tuning

To refine and optimize the model, hyperparameter tuning was explored.
The tuning was carried out via GridSearch to get the best possible model from a set of possibilities which include:

- penalty: L1, L2, and elasticnet.
- Solver: newton-cg, lbfgs, liblinear, sag, saga.
- C: ranging from 10^{-2} to 10^{3}

These ranges of hyperparameter values were selected to keep models relatively simple, not so computationally expensive (particularly for the C values as too large C values overfit the model) and to be exhaustive in experimentation at the same time.
From this set of possibilities, the output from grid search as the best hyperparameters was:

C: 10.0, Penalty: L2, Solver: saga.

This improved the model performance on the training and testing data (verified through stratified cross-validation) for all metrics except the testing AUROC. Where it is lower than the base logistic regression model by $1\text{x}10^{-4}$ which is somewhat negligible. On the leaderboard, it performed better than both the baseline model and the base logistic model (i.e., the default logistic model built on preprocessed data). As before, stratified 10-Fold Cross-Validation was carried out to evaluate model performance reliably.

Table 7: Results of Logistic Regression Model on Hyperparameter Tuning

		Training set			**Testing set**			**Leaderboard**	
Approach	BoW Feature Size	Error Rate	Log-Loss	AUROC	Error Rate	Log-Loss	AUROC	Error Rate	AUROC
Baseline	4,510	0.0204	0.3902	0.9746	0.175	0.4887	0.8898	0.1833	0.8896
Base Logistic Regression Model	3,524	0.06722	0.3892	0.9725	0.1725	0.4756	0.90168	0.17833	0.9039
Logistic Regression Model + Hyperparameter Tuning GridSearch	3,524	0.01338	0.1618	0.9961	0.1717	0.40205	0.90156	0.17667	0.9056

Hyper-Parameter Selection to Control Model Complexity and/or Tendency to Overfit

The hyperparameters selected for varying and exploration are:

1. The Inverse of Regularization Strength, C: This was selected because it controls the level of regularization applied, smaller C values specify stronger regularization on weights.

2. Penalty/Norm of Penalty: This specifies the type of penalty norm applied to drive the weights towards 0 (or possibly to 0 in the case of L1 norm for low enough regularization strength or high enough C value). Available penalty values are on sklearn are:
   - L1
   - L2
   - Elasticnet
   - None

3. L1-Ratio: This works only with the elasticnet penalty - which combines the L1 and L2 norm penalty. The L1-Ratio value ranges from 0 to 1 i.e. 0 <= L1_ratio<= 1.

When L1_ratio is set to 0, we have the L2 penalty and when it is set to 1, we have the L1 penalty. When it is between 0 and 1. The penalty is a combination of L1 and L2. This hyperparameter was selected because it varies the penalty norm, essentially showing us what happens when we select either an L1 or L2 penalty and also when we combine them.

To implement this, I wrote a python function that takes in the training data (x_train), the corresponding label data (y_train), and the range of hyperparameter values in a list. The function contains 2 for-loops. The first for-loop parses through the hyperparameter list and the second for-loop performs the stratified 10-fold cross-validation. Outside the first for-loop, empty lists are initialized and these lists correspond to the performance metrics we wish to track (error rate, log-loss, AUROC). This list will essentially store the values of aggregated metric scores (average and standard deviation values) for each hyperparameter distinct value.

Inside the first for loop, another set of empty lists is initialized, these lists get the training and testing performance metrics for every fold per hyperparameter distinct value and perform numpy averaging and standard deviation operations on this list after all iterations in the second for loop. These scalar values are then stored in the set of lists initialized outside the first loop by

appending. The lists inside the first for-loop resets after each cross-validation loop cycle completes to capture the cross-validated performance metric of the next distinct hyperparameter value. After all iterations of both for-loops, the lists containing the aggregated metric values for every distinct hyperparameter value together with the original hyperparameter list are zipped and stored in a pandas dataframe. Each dataframe entry corresponds to the average and standard deviation (to estimate uncertainty) error, AUROC, and cross-entropy loss of training and validating data across the 10 folds for each value of the chosen hyperparameters. For every hyperparameter, this function is adjusted and re-used accordingly (this function is used for all hyperparameters explored for all models built in this project i.e the Logistic Regression, MLPClassifier, and SVC).
An example of the pandas dataframe generated from the function is given in the image below:

	Penalty	Mean Train Error	Training Error Rate Standard Deviation	Mean Test Error	Testing Error Standard Deviation	train_loss	test_loss	train_auc	test_auc
0	none	0.001157	0.000427	0.188750	0.024654	0.019751	0.577673	0.999992	0.885866
1	l1	0.171343	0.002540	0.209167	0.022807	0.419497	0.458710	0.913106	0.878319
2	l2	0.066574	0.002509	0.175000	0.019365	0.389178	0.476456	0.972415	0.901400
3	elasticnet	0.131204	0.002756	0.199583	0.022240	0.429987	0.476622	0.934526	0.887026

Fig 2: Results of Hyperparameter Variation

The uncertainty of the results was estimated by obtaining the standard deviation across the 10-fold cross-validation tests. For each of the 5 distinct values, stratified 10-fold cross-validation was applied and the standard deviation for each distinct value (corresponding to the standard deviation of the 10-fold cross-validation for said distinct hyperparameter value) was captured, and on average we have:

Table 8: Results of Showing Standard Deviation on Average of Training and Testing Error (Logistic Regression)

Hyperparameter	Standard Deviation on Training Error Rate Results	Standard Deviation on Testing Error Rate Results
Inverse of Regularization Strength, C	0.02264	0.0383
L1_Ratio	0.00394	0.0221
Penalty	0.00206	0.0223

The low standard deviation shows the average performance is relatively stable and consistent from one cross-validation test to another.

Figures for the Tested Hyperparameters (Logistic Regression)

From the dataframe returned by the function output, plotting the tested hyperparameter's performance on at least 5 distinct values was possible, the plots for each parameter varied is given below:

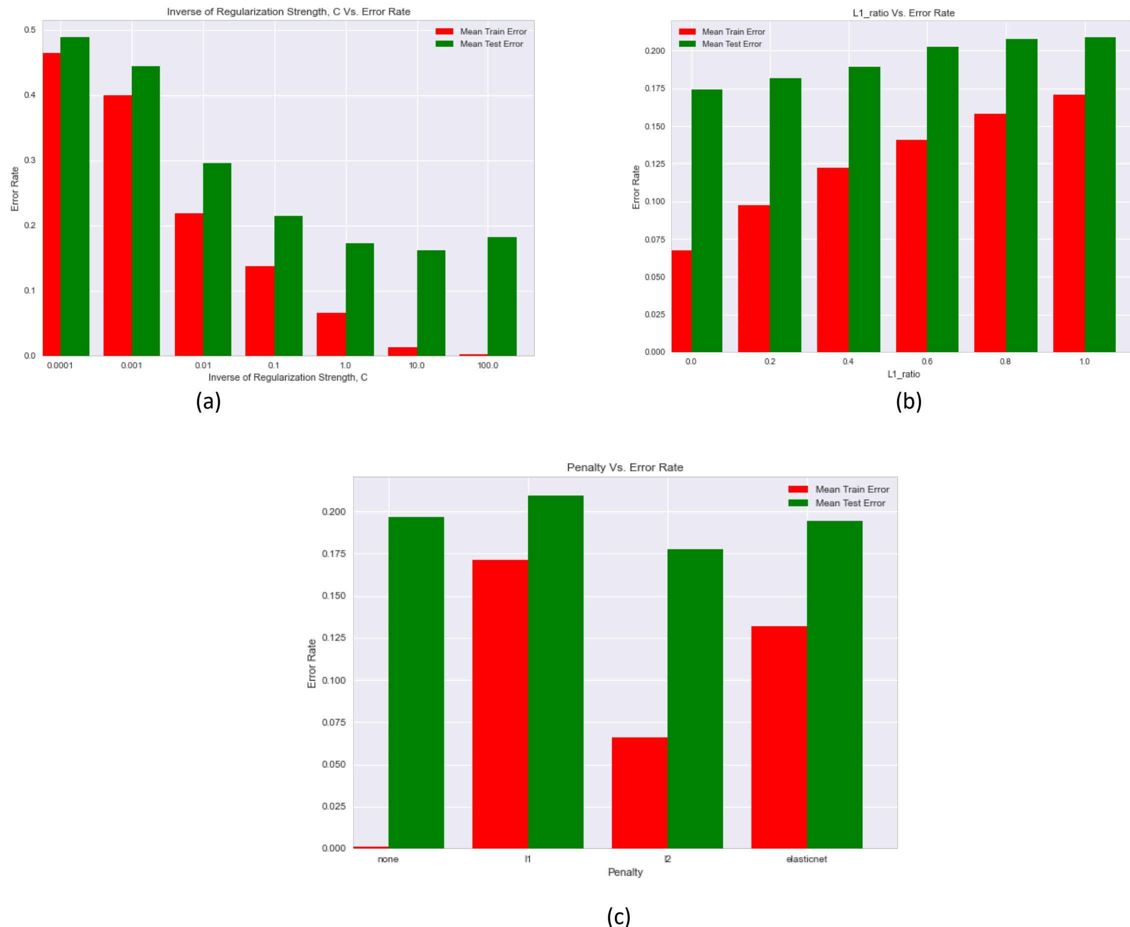


Fig. 3 a) Training and Testing Error Rates vs. Inverse of Regularization Strength, C averaged across 10 folds for each distinct C value. b) Training and Testing Error Rates vs. L1_Ratio averaged across 10 folds for each distinct value L1_Ratio Value. c) Training and Testing Error Rates vs. Penalty Type averaged across 10 folds for each distinct penalty type.

Observing the figures, starting with fig. 3a the inverse regularization strength against error rate, we see that at C = 0.0001 to C = 10.0, the model improves (as C is increasing the regularization on the weights becomes weaker and underfitting is reduced). The model performs best at C = 10 (this supports the GridSearch results). However, as C increases to 100. We observe training error reduces while the testing error gets worse indicating overfitting at C = 100 and above.

Also, for fig. 3b Varying the L1-ratio, we see that when the value of L1-ratio is 0.0 (this is equivalent to L2 norm) we have the best result and as we increase the L1-ratio value, the

performance declines (training and testing error rates increase) and we have the worst performance on training and testing at L1-ratio = 1.0(which is equivalent to L1 norm). This also corroborates the results obtained from GridSearch.

Finally, for fig. 3c varying the penalty types, is closely related to fig. 3b in interpretation. We observe that when the penalty is 'none' we have the best training error rate performance and one of the worst testing performances (high error rate). This strongly indicates that with no penalty applied we have overfitting. We have the best performance when the penalty is set to L2 norm which corroborates with fig(b) and the results obtained from GridSearch.

3) <u>Multilayer Perceptron (MLP)</u>

Carrying out the same analysis on a base MLP model using BoW feature vectors obtained from the preprocessing pipeline. To evaluate its performance on training and validation data, a stratified 10-fold cross-validation was used. The results are given in the table below:

Table 9: Comparing Preprocessed BoW on Base MLP Model with Baseline

		Training set			**Testing set**			**Leaderboard**	
Approach	BoW Feature Size	Error Rate	Log-Loss	AUROC	Error Rate	Log-Loss	AUROC	Error Rate	AUROC
Baseline	4,510	0.0204	0.3902	0.9746	0.175	0.4887	0.8898	0.1833	0.8896
Base MLP Model	3,524	0.00	0.0041	1.00	0.2092	0.7266	0.8777	0.2117	0.8739

Comparing the output of the MLP model built on the processed BoW pipeline, to the baseline model, we see a decline in the testing set and leaderboard performance and we also see a 0-error rate and AUROC of 1.0 on the training set strongly indicating overfitting (the base MLP Model does not generalize well on new data). The testing performance of both models is not largely different from the leaderboard performance.

<u>Hyper-Parameter Tuning</u>

To refine and optimize the model, hyperparameter tuning was explored.
The tuning was carried out via GridSearch to get the best possible model from a set of possibilities which include:

- Activation: Logistic, Tanh, and Relu.
- Hidden Layer Sizes: (50,), (100,), (50, 100), (150,), (100, 10)
- Alpha: ranging from 10^{-2} to 10^{3}

These ranges of hyperparameters were selected to keep models relatively simple, and not so computationally expensive, particularly in the case of hidden layer sizes. Values above this were too complex and made models built on them overfit. They were also selected to be exhaustive. From this set of possibilities, the output from grid search as the best hyperparameters:

Activation = Tanh, Hidden Layer = (100,), Alpha = 1.0.

This improved the model performance on the training and testing data (verified through stratifi ed cross-validation) for all metrics. On the leaderboard, it performed better than the baseline m odel and the base logistic model, and base MLP model. As before, stratified 10-Fold Cross-Valid ation was carried out to evaluate model performance reliably.

Table 10: Result Exploring Hyperparameter Tuning on the MLP Model

		Training set			**Testing set**			**Leaderboard**	
Approach	BoW Feature Size	Error Rate	Log-Loss	AUROC	Error Rate	Log-Loss	AUROC	Error Rate	AUROC
Baseline	4,510	0.0204	0.3902	0.9746	0.175	0.4887	0.8898	0.1833	0.8896
Logistic Regression Model + Hyperpara meter Tuning GridS earch	3,524	0.01338	0.1618	0.9961	0.1717	0.40205	0.90156	0.17667	0.9056
Base MLP Model	3,524	0.00	0.0041	1.00	0.2092	0.7266	0.8777	0.2117	0.8739
MLP Model + Hyper parameter Tuning GridSearch	3,524	0.0346	0.2582	0.9898	0.1725	0.4188	0.9070	0.1667	0.9081

Hyper-Parameter Selection to Control Model Complexity and/or Tendency to Overfit

The hyperparameters selected for varying and exploration are:

1. Alpha: This was selected because it controls the regularization term (on the L2 norm), Increasing alpha could fix overfitting (by encouraging smaller weights), and decreasing alpha could fix underfitting (by encouraging large weights).
2. Hidden Layer Size: This was selected because this hyperparameter could significantly affect the model complexity and lead to overfitting (if the model is too deep) and also shallow models could trade off simplicity for performance (underfitting).

The same function used by the logistic model to vary the hyperparameters was leveraged here to generate the pandas dataframe for visualization.

Similarly, the uncertainty of the results was estimated by obtaining the standard deviation across the 10-fold cross-validation tests for each of the 5 distinct values for each hyperparameter. A summary is given in the table below:

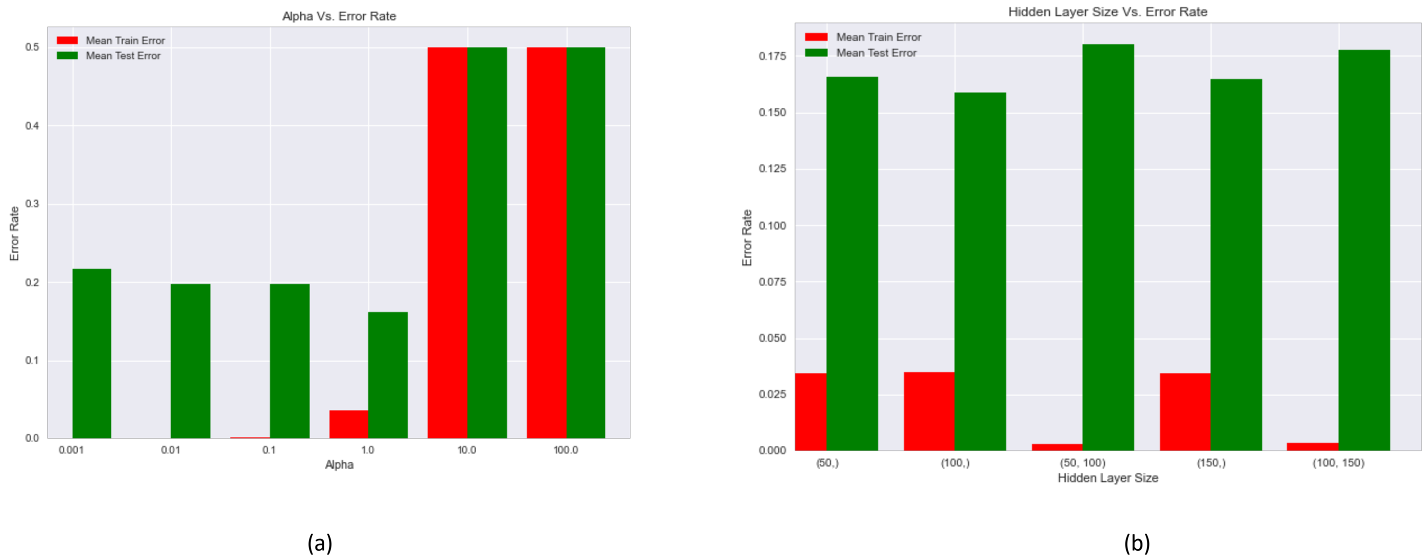
Table 11: Results of Showing Standard Deviation on Average of Training and Testing Error (MLP)

Hyperparameter	Standard Deviation on Training Error Rate Results	Standard Deviation on Testing Error Rate Results
Alpha	0.00125	0.02268
Hidden Layer Size	0.001535	0.02331

The low standard deviation shows the average performance is relatively stable and consistent from one cross-validation test to another.

Figures for the Tested Hyperparameters (MLP)

From the dataframe returned by the output, plotting the tested hyperparameter's performance on at least 5 distinct values was possible, the plots for each parameter varied are given below:



(a) (b)

Fig. 4 a) Training and Testing Error Rates vs. Alpha averaged across 10 folds for each alpha distinct value. b) Training and Testing Error Rates vs. Hidden Layer Size averaged across 10 folds for each distinct value Hidden Layer Size.

From fig. 4a, we see that a model with 0 error rate on training and a little over 0.2 on the testing set, at alpha = 0.001 as we increase alpha, the model seems to improve on the testing set and decline a training set at Alpha = 1.0, we have the best testing error rate performance (agreeing with GridSearch) and poorer but moderate training performance (indicating a more generalized model i.e. overfitting is reduced). Beyond Alpha = 1.0 However, the performance on both training and testing set declines.

For fig. 4b, we see that selecting a hidden layer of (100,) gives the best performance (agreeing with GridSearch) and hidden layer sizes of (50, 100) and (100, 150) give the worst performance on testing and the best performance on training. This shows overfitting. For hidden layer sizes of (50,) and (150,) they give a moderate performance on training and testing. However, overall selecting a hidden layer size of (100,) gives the best performance.

4) Support Vector Classifier (SVC)

The third model considered was a support vector machine classifier. Carrying out the same analysis on a base MLP model on BoW feature vector obtained from the preprocessing pipeline. To evaluate its performance on training and validation data, a stratified 10-fold cross-validation was used. The results are given in the table below:

Table 12: Result of Base SVC Model

		Training set			Testing set			Leaderboard	
Approach	BoW Feature Size	Error Rate	Log-Loss	AUROC	Error Rate	Log-Loss	AUROC	Error Rate	AUROC
Baseline	4,510	0.0204	0.3902	0.9746	0.175	0.4887	0.8898	0.1833	0.8896
Base SVC Model	3,524	0.00829	0.04733	0.9986	0.1599	0.39063	0.9103	0.16167	0.9141

Comparing the output of the SVC model built on the preprocessed BoW pipeline, to the baseline model, we see a significant improvement in performance on the training and testing set on all performance metrics considered, we also see a significant improvement on the leaderboard.

Hyper-Parameter Tuning

To refine and optimize the model, hyperparameter tuning was explored.
The tuning was carried out via GridSearch to get the best possible model from a set of possibilities which include:

- The inverse of Regularization Strength, C: ranging from 10^{-2} to 10^{2}
- Gamma: ranging from 10^{-2} to 10^{2}
- Kernel: RBF, Poly, Sigmoid, Linear

These ranges of hyperparameters were selected to keep models relatively simple. Values above these were too complex and started to overfit (particularly for C values). They were also selected to be exhaustive (for the case of kernel function).
From this set of possibilities, the output from grid search as the best hyperparameters:

C : 1.0, Kernel : RBF, Gamma: 1

This improved the model performance on the training and testing on all metrics as verified thro ugh stratified cross-validation (based on the training's perfect metrics However, the model sho ws signs of overfitting). On the leaderboard, it performed worse in terms of error rate and AUR OC value, even performing worse than the base SVC model on the leaderboard. This could be d ue to the model's tendency to overfit as can be observed in the training set performance.

Table 13: Result Exploring Hyperparameter Tuning on the SVC Model

		Training set			Testing set			Leaderboard	
Approach	BoW Feature Size	Error Rate	Log-Loss	AUROC	Error Rate	Log-Loss	AUROC	Error Rate	AUROC
Baseline	4,510	0.0204	0.3902	0.9746	0.175	0.4887	0.8898	0.1833	0.8896
Logistic Regression Model + Hyperpara meter Tuning GridS earch	3,524	0.01338	0.1618	0.9961	0.1717	0.40205	0.90156	0.17667	0.9056
Base SVC Model	3,524	0.00829	0.04733	0.9986	0.1599	0.39063	0.9103	0.16167	0.9141
SVC Model + Hyper parameter Tuning GridSearch	3,524	0.00	0.010	1.0	0.1571	0.3706	0.9177	0.17	0.9158

Hyper-Parameter Selection to Control Model Complexity and/or Tendency to Overfit

The hyperparameters selected for varying and exploration are:

1. The inverse of Regularization Strength, C: This was selected because it controls the level of regularization applied, smaller C values specify stronger regularization on weights.

2 Kernel: This was selected because this hyperparameter could significantly affect the model complexity, the kernel function separates data meaningfully and this could sometimes lead to transformations of weights into higher dimensions.

3 Gamma: This was selected because it serves as the kernel coefficient, the model tends to be sensitive to gamma, if gamma is too large the radius of influence of the support vectors would only support the vector itself and no amount of C will prevent overfitting and if gamma is too small the model gets too constrained that it cannot capture the data complexity/shape of the data.

The same function used by the logistic and MLP models to vary the hyperparameters was leveraged here to generate the pandas dataframe for visualization.
Similarly, the uncertainty of the results was estimated by obtaining the standard deviation across the 10-fold cross-validation tests for each of the 5 distinct values for each hyperparameter. A summary is given in the table below:

Table 14: Results of Showing Standard Deviation on Average of Training and Testing Error (SVC)

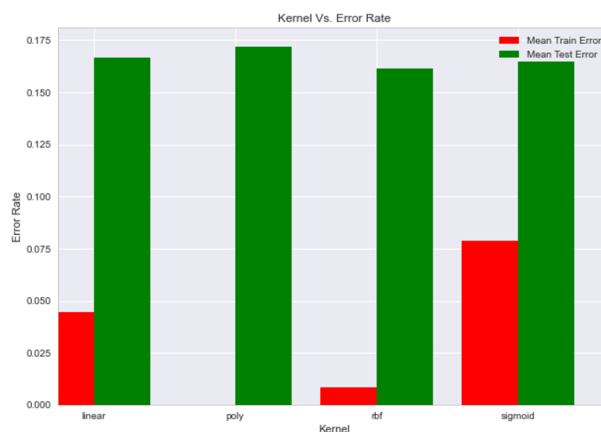
Hyperparameter	Standard Deviation on Training Error Rate Results	Standard Deviation on Testing Error Rate Results
Inverse of Regularization Strength, C	0.0132	0.0259
Kernel Function	0.00113	0.0229
Gamma	0.02836	0.0121

The low standard deviation shows the average performance is relatively stable and consistent from one cross-validation test to another.

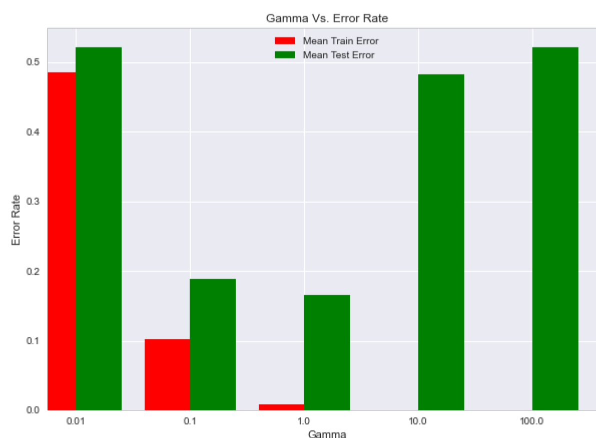
Figures for the Tested Hyperparameters (SVC)

From the dataframe returned by the output, plotting the tested hyperparameter's performance on at least 5 distinct values was possible, the plots for each parameter varied are given below:

(a)



(b)



(c)

Fig. 5 a) Training and Testing Error Rates vs. Inverse of Regularization Strength, C averaged across 10 folds for each C distinct value. b) Training and Testing Error Rates vs. kernel function averaged across 10 folds for each distinct kernel function Value. c) Training and Testing Error Rates vs. Gamma averaged across 10 folds for each distinct gamma value.

From fig. 5a, at C = 0.01 the model appears to underfit (meaning its stronger regularization on weights is applied) as we see a poorer performance in training and testing error rates. As we increase C the model starts to improve until we get to a C value of 1.0. Beyond this value, the model overfits as can be seen with the 0-error rate performance in training and the increasing error rate in testing. C =1.0 gives the best model performance.

From fig. 5b exploring the kernels we observe that the RBF kernel gives the best and most generalized performance (with the lowest testing error rate and a moderate training error rate

as well). The kernel poly, however, gives the worst performance as it is clearly overfitting the model with the best training performance and the worst testing performance.

N.B: 5 distinct values could not be generated for the kernel as the 5th kernel available on sklearn called 'precomputed' only works with square matrix data which is not the case for the BoW feature vector which is 2400 by 3524. Therefore, this kernel could not be explored.

From fig. 5c: we see that at gamma = 0.01, we have an underfit model and as we increase gamma up to 1 (which gives the best performance) the model improves. Beyond gamma as 1, the model gets worse (overfitting) as the testing error rate increases while the training error rate is 0.

5) Models Summary

In general, the support vector machine (SVM) classifier models explored give the best performance, better than the Logistic Regression Model and the MLP Model in all performance metrics considered (i.e., error rates, cross-entropy loss, AUROC) in the training and testing set and also on the leaderboard.
Narrowing down between the SVM classifier models, the SVC model with hyperparameter tuning does perform better on the testing set and on the training set with a 0-error rate, 1.0 AUROC, and a cross-entropy loss of 0.01 on the training set which clearly indicates overfitting and the potential to generalize poorly on new examples when compared to the base SVC model. Therefore, I considered the base SVC model as the best-performing model. The results are captured in the table below:

Table 15: Results of Models Explored

		Training set			Testing set			Leaderboard	
Approach	BoW Feature Size	Error Rate	Log-Loss	AUROC	Error Rate	Log-Loss	AUROC	Error Rate	AUROC
Baseline	4,510	0.0204	0.3902	0.9746	0.175	0.4887	0.8898	0.1833	0.8896
Logistic Regression Model + Hyperparameter Tuning GridSearch	3,524	0.01338	0.1618	0.9961	0.1717	0.40205	0.90156	0.17667	0.9056
MLP Model + Hyperparameter Tuning GridSearch	3,524	0.0346	0.2582	0.9898	0.1725	0.4188	0.9070	0.1667	0.9081
Base SVC Model	**3,524**	**0.00829**	**0.04733**	**0.9986**	**0.1599**	**0.39063**	**0.9103**	**0.16167**	**0.9141**
SVC Model + Hyperparameter Tuning GridSearch	3,524	0.00	0.010	1.0	0.1571	0.3706	0.9177	0.17	0.9158

The SVC models performed best, though they seem to have the tendency to overfit as can be observed from their almost flawless performance in training compared to their testing performance. Comparing the SVC models, the base SVC model seems better at avoiding overfitting than the SVC model built via hyperparameter tuning.

I further analyzed the results of the base SVC model. This was achieved by concatenating the y_train dataframe to the x_train dataframe and also generating predictions (based on the base SVC model) in an array and adding the predictions as a column to the concatenated dataframe. I then created a final column for error checking (naming it error bit). If the y_train value for a row and the corresponding prediction value are not equal assign a 1 to the 'error bit' column else assign a 0. This helped in filtering out the misclassifications. A summary of the errors is given below:

Table 16: Quantifying Model Misclassifications by the Base SVC Model (Best Model)

SVC Model	Misclassification Count by Domain	Total Misclassifications
Base SVC	Amazon = 9, IMDB = 24, Yelp = 15	48

I could observe a pattern with the misclassifications, most of the misclassifications were due to context. Words that generally have negative meanings used in positive contexts were classified as negative sentiments by the model, some of these words "unfortunately", "not", "disappointed","terrible".

Some of the examples of misclassified examples from the training dataset are given in the table below:

Table 17: Specific Misclassification Examples

Text From Training Data	Correct Label	Predicted Label
"The soundtrack wasn't terrible, either."	1	0
"I don't think you will be disappointed. "	1	0
"Not too screamy not to masculine but just right..."	1	0
"A film not easily forgotten."	1	0
"The kids play area is NASTY!"	0	1

6. Applying my best classifier, the base SVC model to the x_test.csv and storing the outcome as probabilistic predictions and submitting it to the leaderboard. The performance on the leaderboard is captured in the table below:

Table 17: Comparing Base SVC Model (Best Model) Cross-validation Results to Leaderboard Results

		Training set			Testing set			Leaderboard	
Approach	BoW Feature Size	Error Rate	Log-Loss	AUROC	Error Rate	Log-Loss	AUROC	Error Rate	AUROC
Base SVC Model	**3,524**	**0.00829**	**0.04733**	**0.9986**	**0.1599**	**0.39063**	**0.9103**	**0.16167**	**0.9141**

The cross-validation performance is not significantly different from the leaderboard performance, the cross-validation testing error is 0.1599 (which corresponds to an accuracy of 84.01%) and the leaderboard error rate is 0.1617(which corresponds to an accuracy of 83.8%). This is a difference of 0.00178 in error rate (0.00021 or 0.21% accuracy).

In terms of AUROC also, the cross-validation performance and leaderboard performance are not largely different from the cross-validation result, 0.9103, and the leaderboard result, 0.9141 differ by 0.0038. This implies that the model generalizes relatively well on new data and the slight difference in numbers might be because the test/leaderboard data might slightly vary in distribution. This analysis generally applies to all my leaderboard submissions of the different models explored. The submission ranked 5th in terms of error rate and 3rd in terms of AUROC.