Sam Miller, Sami Hakkareinen
CS135
March 2, 2024

Project 1 Report

Problem 1

1A : Bag-of-Words Design Decision Description:

For our Bag-of-Words feature representation we chose to use 'CountVectorizer' from SciKitLearn. CountVectorizer transforms textual data into a numerical representation of the text, and the result will take on the form of a feature vector x_n, of a standard length equal to the size of the dictionary. We chose to quantify the text values as **counts** rather than binary, because we are more focused on the quantity of words relating to a given sentiment rather than if a word is present or not. So, the vocabulary represents how many times a word appears in the documents, and based on the vocabulary, each document is converted into a feature vector.

During the preprocessing phase, we utilized the CountVectorizer to "clean" the text. This involved converting all text to lower case and removing punctuation. Such normalization ensures that words are consistently represented in the BoW model, regardless of their appearance in the original documents. Further, we eliminated all standard English "stop words" from the vocabulary and specified that our vectorizer is to only work with unigrams. Only including unigrams (single words) in our vocabulary makes it significantly smaller and less complex, but it was a forced limitation for this step. Any out-of-vocabulary words in a test set will be ignored. This is standard practice with the CountVectorizer object.

After running the training data on the CountVectorizer, the vocabulary was 4255. When trained with a Vectorizer that did not ignore "stop words" when creating a vocabulary, the vocabulary size was 4510. We chose to eliminate stop words from the vocabulary because they tend to skew results because of their high frequency.

1B : Cross Validation Design Description:

In order to perform cross validation, we utilized "GridSearchCV" from SciKitLearn. The classifier pipeline we used was simply just a "LogisticRegression" object from SciKitLearn. Within GridSearchCV, we set the parameter cv=5, meaning we used a 5-fold CV approach. This meant our Logistic Regression model was cycled 5 times, each time grouping a randomly selected ⅘ of the data to be used for the train set and the other ⅕ of the data used for the validation set. Because our original training set is 2400 documents, in our 5-fold approach this will result in 1920 training values and 480 validation values.

The metric we used to optimize heldout data was area under the ROC curve. We did this by setting the parameter *scoring='roc_auc'* within the GridSearchCV call. This metric was selected for its ability to quantify the model's performance across all classification thresholds, thereby offering a balanced measure of sensitivity (true positive rate) and specificity (true negative rate). ROC AUC is particularly useful in binary classification tasks for its insensitivity to changes in the class distribution, making it an ideal

choice for evaluating our sentiment analysis model's ability to distinguish between positive and negative sentiments accurately.

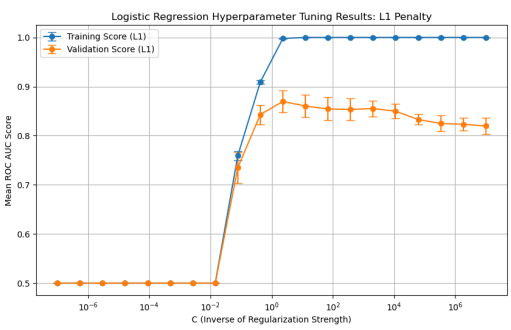
When identifying hyperparameter configuration, one feature of the GridSearchCV we were able to utilize is “param_grid”, where we chose to tune “C” and “penalty” parameters within the LogisticRegression. We plotted the results of this hyperparameter tuning, which is included in part 1C, but ultimately we built one final model for which we would run our test set by using the “best_estimator_” attribute of the GridSearchCV object. We made sure that our outcome of step C aligned with this attribute.

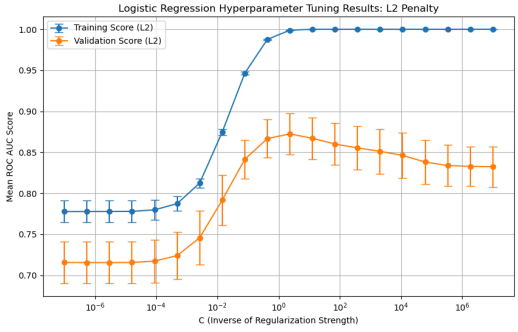
1C : Hyperparameter Selection For Logistic Regression Classifier

The LogisticRegression classifier has several potential advantages. Some that come to mind are its ability to control the **strength of regularization** to avoid overfitting**,** as well as selecting which **type regularization (L1 and L2 penalties)**.

The hyperparameters we built our grid on are **C values,** which relate to the strength of a model’s regularization, and **penalties** (L1 or L2), denoting how this regularization is applied. In order to handle both **L1 and L2 regularization**, we chose the ‘liblinear’ solver. We also chose this because it is optimal for handling small data sets. As for our **C values,** we searched **values across 20 log spaced values ranging from 10^{-6} to 10^{6}.** Generally, a higher C value will lead to a lower regularization strength, which leads to a more complex model that may fit the training data very well, but doesn’t do well on testing data. Whereas a lower C value means a high regularization strength, which makes the decision boundary smoother, leading to a simple model. This is a common reason for underfitting data. So, exploring a wide range of C values as well as analyzing these C values for both L1 and L2 penalty types is a reasonable way to explore the transition between underfitting and overfitting models.

During the training process we encountered convergence issues when trying to fit the Logistic Regression, seeing warning signs suggesting us to increase the max number of iterations. So, we did this and were able to get around this issue.





Grid_search.best_params_: {'C': 2.069, 'penalty': 'l2'}

We can see similar optimal C values for both L1 and L2 regularization penalties. This optimal value is around 10^{0}, or **C = 1**. To be exact, the optimized value for our model was **C = 2.069**. Underfitting and

overfitting can clearly be seen in these figures, as the AUROC score for the validation set behaves poorly as C significantly increases or decreases. The peak in the middle is the “sweet spot”.

Further, analyzed as L1 vs L2, the AUROC for the L2 Penalty was better at every instance than the L1 penalty. This can be seen in the data below, as L2 penalty at each C instance has a higher AUROC for the validation set:

	param_C	param_penalty	mean_test_score
0	2.069138	l2	0.872196
1	0.483293	l2	0.870043
2	8.858668	l2	0.865179
3	2.069138	l1	0.859774
4	0.112884	l2	0.859040
5	37.926902	l2	0.851804
6	8.858668	l1	0.848917
7	0.026367	l2	0.846109
8	0.006158	l2	0.835675
9	162.377674	l2	0.835238
10	0.001438	l2	0.831738

Thus, the L2 is the preferred penalty, and a C value of 2.069 is preferred, and this is backed by decisive evidence from the figures provided, the table, and the GridSearchCV.

1D : Analysis of Predictions for the Best Classifier

In order to analyze which documents failed, we used the following technique:

- Create a 5 fold example of the training data using the StratifiedKFold library. This will replicate the “Fold Pattern” that is used in GridSearchCV.
- After obtaining this object, split the train data apart using this fold. This will give a set of indices that are a part of the training set, and a set that represents the validation set.
- This validation set is the “heldout data”
- Reconstruct the x_train and y_train sets with these indices of the validation set
- Run a prediction on a model with this reconstructed test set, and print out the instances of false predictions.

By following all these steps, we were able to ensure that we were isolating the “heldout” data and printing out False positives and False Negatives from this set of data.

False Positive Examples:	
Example 1:	“None of the three sizes they sent with the headset would stay in my ears.”
Example 2:	“My father has the V265, and the battery is dying.”
Example 3:	“I really wanted the Plantronics 510 to be the right one, but it has too many issues for me.The nice”
Example 4:	“Att is not clear, sound is very distorted and you have to yell when you talk.”

False Positive Analysis:

The Logistic regression classifier's false positives often stem from an inability to interpret contextual cues and negations accurately. For instance, sentences that contain technically neutral or positive terms, such as

product names or features, alongside a negative context ("None of the three sizes... would stay in my ears") lead to misclassification. This suggests the classifier might overemphasize certain words without fully understanding their context. An analysis of the source of these reviews did not distinctly show a bias towards one type of review (e.g., Amazon, IMDb, or Yelp), indicating that the issue is more related to linguistic structure than content type. Interestingly, sentences with negation words ("not", "no", etc.) or those expressing dissatisfaction in a nuanced manner ("it has too many issues for me") were particularly challenging for the model. There was no clear pattern regarding sentence length affecting performance, suggesting that the classifier's mistakes are more about context and negation than about the complexity or length of sentences.

False Negative Examples:	
Example 1:	"i'm glad i found this product on amazon it is hard to find, it wasn't high priced."
Example 2:	"'Virgin Wireless rocks and so does this cheap little phone!"
Example 3:	"I got it because it was so small and adorable."
Example 4:	"'I really like this product over the Motorola because it is allot clearer on the ear piece and the mic."

False Negative Analysis:

Similarly, the false negatives reveal challenges in recognizing positive sentiment when expressed subtly or through comparative statements. Phrases that might typically carry a negative connotation ("cheap little phone," "small") but are used positively here highlight the model's difficulty in grasping sentiment within specific contexts. This pattern suggests that enhancing the model's ability to understand comparative positives and context-specific sentiment could improve performance. Again, the analysis didn't show a significant difference in performance based on the review source, indicating that the classifier's issues are uniformly present across different domains. The presence of comparative or nuanced positive statements in shorter sentences ("I got it because it was so small and adorable") did not significantly affect the model's performance, pointing again to contextual understanding as the primary area for improvement.

1E : Report Performance on Test Set via Leaderboard

Our chosen classifier achieved a score of 91.571% on the ultimate autograder test set, putting us second in the class for this section of the project. In our cross validation test, we observed very similar results, but slightly lower. From our figure in 1C, you can see that with our best 'C' value, we model performed with an AUROC score of 87%. With that being said, the standard deviation was around 3%, so it is not alarming that our model performed better in practice than on average in our cross validation testing. We are certainly lucky to be on the more fortunate side of this standard deviation with the autograder test. Also, this graph shows the average performance on train data over each fold. Since we had five folds, the training data was 4/5 the size that is trained on before it is submitted to the autograder. Once we found the optimal 'C' value, we then used that and then trained the classifier on the whole dataset, which would offer marginal improvements because the training set is slightly larger once we have found the optimal 'C' value. In other words, it makes sense that we saw improvement when we tested with the autograder

because the size of our training data for the final model was bigger than in the gridsearchCV when we were trying to optimize parameters across the sample data.

Problem 2

2A: Feature Representation Description

When cleaning/preprocessing the data, we again converted all the text to lowercase and got rid of special characters, much like part 1A. Also, we omitted "stop words" and chose to ignore words outside of the vocabulary. The main differences in how we're transforming text into fixed-length feature vectors are the following:

- "Ngram" range is now from 1-2. This means that both unigrams and bigrams are considered.
- TfidfVectorizer used instead of CountVectorizer

In part 1, we were restricted to **only using unigrams**. This made it difficult to correctly analyze reviews that had negation words. See part 1D for more reference. By handling **bigrams**, this expands the vocabulary of vectorizer, and allows for these negation words to be handled properly.

The main difference is the type of vectorizer used. Previously, we were using a CountVectorizer, which simply counts the number of times a word appears in a document. We moved onto the **TfidfVectorizer**, which stands for Term Frequency and Inverse Document Frequency vectorizer. The main idea of this vectorizer is that it measures how unique or common a word is across an entire set of documents. The more a word appears, the lower IDF score it has. The TF is similar to the CountVectorizer, keeping track of the term frequency. These two metrics (TF & IDF) are used in parallel to create a fixed length feature vector for each document, ready for classification.

2B: Cross Validation (or Equivalent) description:

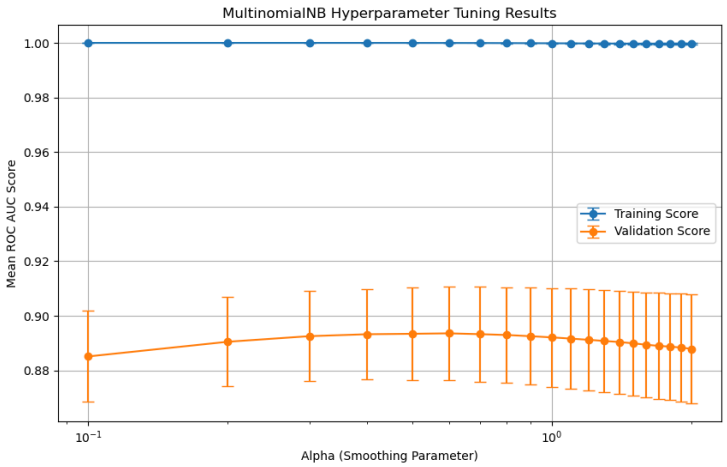
The cross validation method is similar to part 1B, where we are using 5-fold cross-validation within `GridSearchCV`. For the same reasons, we are using the AUROC metric for performance. The main difference is now that our classifier is a Multinomial Naive Bayes model, so we will need new parameters to go along with it. The main hyperparameter MultinomialNB expects is 'alpha', so we will provide a range of alpha values to tune this hyperparameter.

2C: Classifier description with Hyperparameter search:

The classifier we chose, **MultinomialNB**, is a simple classifier that relies on Bayes theorem. One advantage of using this classifier is the **alpha** hyperparameter, which can be tuned to dictate the model's sensitivity to low frequency words, something that is important to our use case. Thus, we built our value grid on these **alpha values**, searching values across **20 log spaced values ranging from 0.1 to 2**. For alpha values closer to zero, the smoothing value has less of an effect, which could lead to underfitting. Surprisingly, as alpha values get larger, there is also a risk of underfitting. See, the model could make the smoothing value too large, to where the model is more reliant on this alpha value than the actual data, and the model is less sensitive overall. There were **no convergence issues** with this training process, so no modifications to step size or early stoppage were necessary.

Multinomial Naive Bayes (MultinomialNB) is a probabilistic learning model that is particularly suited for classification tasks involving discrete features, such as word counts or frequencies in text documents. It

operates under the simplifying assumption of feature independence given the class, calculating the likelihood of a document belonging to a certain class (good or bad) based on the frequency of each feature (word) in the document. Due to its computational efficiency, natural handling of text data, and effectiveness with high-dimensional sparse datasets, MultinomialNB is a compelling choice for sentiment analysis, offering a straightforward and robust method for categorizing text based on sentiment.



The figure above has a peak near the center, and the tails to either side of the graph illustrate the underfitting across the range of alpha values. From the figure, the optimal value is **alpha = 0.6**, and this is reflected in the GridSearchCV object. The attribute "best_params_" within this object is also an alpha of 0.6, matching up with the results from the graph. Thus, it is certain that this alpha value of **0.6** is preferred for a MultinomialNB classifier.

2D: Error Analysis:

We used the same technique to find false negatives and false positives for part 2 as we did in part 1D.

False Positive Examples:	
Example 1:	" Plus, I seriously do not believe it is worth its steep price point."
Example 2:	"If you plan to use this in a car forget about it."
Example 3:	"If you are looking for a nice quality Motorola Headset keep looking, this isn't it."
Example 4:	"The internet access was fine, it the rare instance that it worked."

False Positives Analysis

Our Multinomial Naive Bayes classifier's false positives predominantly arise from its inability to accurately interpret negation and implicit negativity within sentences. Sentences like "Plus, I seriously do not believe it is worth its steep price point" and "If you plan to use this in a car forget about it" were incorrectly classified as positive, likely due to the model's focus on individual words such as "plus" and "nice quality" without understanding their context. This indicates a challenge in parsing nuanced language and conditional statements that invert the sentiment of seemingly positive terms. Unlike the classifier in Problem 1, which may have had similar issues, the sophistication of textual nuances in these examples

suggests an area where feature representation and contextual understanding could be further refined to reduce misclassifications.

False Negative Examples:	
Example 1:	“Plan on ordering from them again and again”
Example 2:	“It does everything the description said it would”
Example 3/4:	“Better than you'd expect”, “Better than expected”
Example 5/6:	“Battery charge-life is quite long”, “It has kept up very well.”

False Negatives Analysis

The false negatives identified in our analysis highlight the model's difficulty in recognizing positive sentiments when they are expressed in a comparative manner or through less direct praise. Phrases such as “Plan on ordering from them again and again” and “Better than you'd expect” were misinterpreted as negative, despite clearly expressing satisfaction and exceeding expectations. This pattern suggests that while our model is capable of identifying straightforward positive expressions, it struggles with more subtle forms of approval. This is consistent with challenges observed in the previous classifier from Problem 1, though the use of TfidfVectorizer in the current model was expected to improve performance by better capturing the significance of such phrases.

2E: Report Performance on Test Set via Leadership:

Our Multinomial Naive Bayes classifier, leveraging the TfidfVectorizer, ultimately achieved a 0.922 AUROC score on the autograder, placing us 9th on the leaderboard. This represents a slight improvement over our previous model, which secured a 91.571% score and a second-place position for that project segment. The enhancement in performance can largely be attributed to our shift in feature representation from CountVectorizer to TfidfVectorizer. Unlike our prior method that prioritized word frequency, TfidfVectorizer emphasizes word relevance, diminishing the weight of common but less informative words. This approach likely allowed for a more nuanced understanding of sentiment within the text, contributing to the improved model accuracy on unseen data.

Reflecting on the comparative analysis of the two models, it's evident that the choice of feature extraction technique plays a critical role in model performance, particularly in tasks involving natural language processing. The TfidfVectorizer's ability to capture the significance of words in context provided a more discriminative feature set for sentiment analysis. Additionally, the exploration of bi-gram features (pairs of consecutive words) in our latest model may have captured contextual nuances that single words (unigrams) could not, further enhancing our ability to discern sentiment accurately. This slight but meaningful improvement in our model's performance, as indicated by the leaderboard, reaffirms the value of iterative refinement and experimentation in model development, particularly through the adoption of more sophisticated machine learning methods.