Introduction

To predict whether customer reviews express positive or negative sentiment, a supervised learning model using binary classification was developed. The dataset, comprising approximately 3000 reviews (2400 training, 600 test), was processed through feature engineering to convert raw text into numerical features. We employed a Bag-of-Words (BoW) approach with TF-IDF vectorization, leveraging Logistic Regression classifiers. The model was trained and evaluated to maximize accuracy, precision, recall, and F1 score, with feature engineering tailored to capture sentiment-bearing terms and phrases effectively.

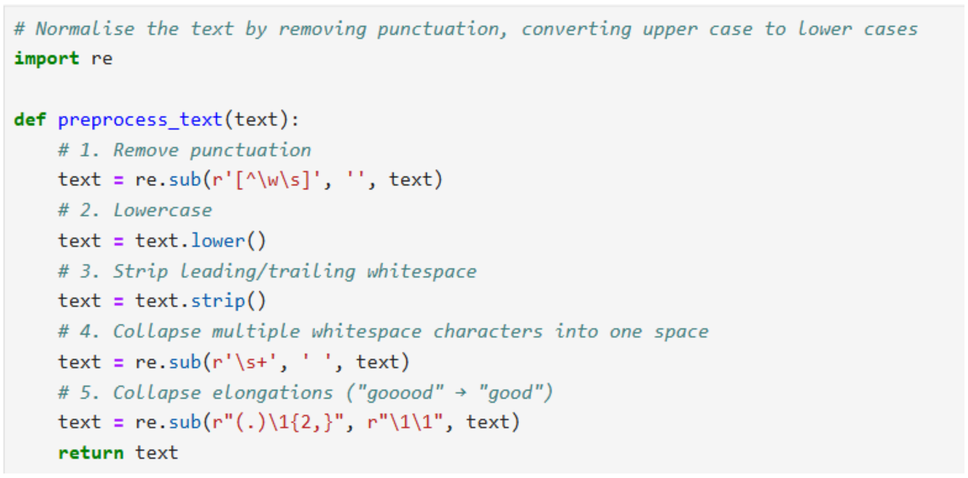
Analysis and Findings

4.1 Bag-of-Words Design Decision Description

The Bag-of-Words (BoW) model transforms customer review text into numerical features by representing each review as a collection of words or phrases, ignoring word order but capturing their frequency or importance. The BoW feature representation begins with a three-stage text-normalization pipeline, followed by a TF-IDF re-weighting step that produces 5 000 features (a mix of unigrams and bigrams). This is an overview of each component:

4.1.1 Text normalization

Function preprocess_text() was implemented to standardize casing, punctuation, and spacing:



```python
# Normalise the text by removing punctuation, converting upper case to lower cases
import re

def preprocess_text(text):
    # 1. Remove punctuation
    text = re.sub(r'[^\w\s]', '', text)
    # 2. Lowercase
    text = text.lower()
    # 3. Strip leading/trailing whitespace
    text = text.strip()
    # 4. Collapse multiple whitespace characters into one space
    text = re.sub(r'\s+', ' ', text)
    # 5. Collapse elongations ("gooood" → "good")
    text = re.sub(r"(.)\1{2,}", r"\1\1", text)
    return text
```

This ensures that the strings "Good!!!", "good", and "gooood" all map to the same canonical token "good". At the next step, common chat-style abbreviations, slangs or simple typos was fixed via expand_slang() function. I maintain a lookup dictionary for slang and then apply a SpellChecker to any token of length ≥ 3:

```python
# Handling slangs and misspelling
from spellchecker import SpellChecker

SLANG_DICT = {
    "u":   "you",
    "ur":  "your",
    "thx":"thanks",
    "pls":"please",
    "idk": "i do not know",
    "lol": "laughing out loud",
    "luv": "love",
    "omg": "oh my god",
    "gr8": "great",
    "l8": "late",
    "gtg": "got to go",
    "imo": "in my opinion",
}

spell = SpellChecker()

def expand_slang(text):
    words = []
    for w in text.split():
        w = SLANG_DICT.get(w, w)           # slang lookup
        if len(w) >= 3:
            w = spell.correction(w) or w   # spelling
        words.append(w)
    return " ".join(words)
```

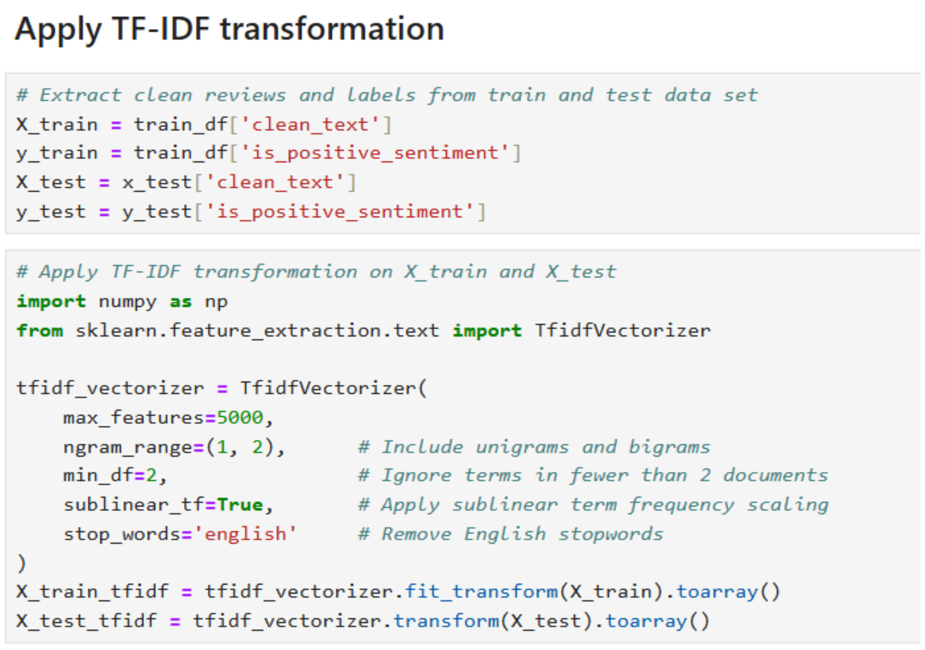
This handles cases like "omg" → "oh my god" and corrects misspellings such as "recieve" → "receive". After this step, some shortened form of a group of words such as “don’t”, “didn’t” still exist so the expand_contractions() function was applied to expand all those words to a standard contraction with a small regex dictionary:

```python
#Expand contraction
import unicodedata, string

CONTRACTIONS = {
    r"\bdon['’]?t\b": "do not",
    r"\bdidn['’]?t\b": "did not",
    r"\bdoesn['’]?t\b": "does not",
    r"\bcan['’]?t\b" : "can not",
    r"\bcouldn['’]?t\b": "could not",
    r"\bshouldn['’]?t\b": "should not",
    r"\bisn['’]?t\b": "is not",
    r"\bwasn['’]?t\b": "was not",
    r"\bweren['’]?t\b": "were not",
    r"\bhaven['’]?t\b": "have not",
    r"\bhasn['’]?t\b": "has not",
    r"\bhadn['’]?t\b": "had not",
    r"\bwon['’]?t\b": "will not",
    r"\bwouldn['’]?t\b": "would not",
    r"\bmustn['’]?t\b": "must not",
    r"\bthey['’]?re\b": "they are",
    r"\bwe['’]?re\b": "we are",
    r"\bwe['’]?ve\b": "we have",
    r"\byou['’]?d\b": "you would",
    r"\byou['’]?ll\b": "you will",
    r"\byou['’]?re\b": "you are",
    r"\bim\b": "i am",
    r"\bive\b": "i have",
    r"\bill\b": "i will",
    r"\bid\b": "i would",


}

def expand_contractions(text: str) -> str:
    # normalise case so all patterns match
    text = text.lower()
    for pattern, replacement in CONTRACTIONS.items():
        text = re.sub(pattern, replacement, text, flags=re.IGNORECASE)
    return text
```

After applying these three steps, each review sentence is clean, lowercased, free of punctuation, uniform in spacing, and normalized for slang, spelling, and contractions.

4.1.2 TF-IDF BoW vectorisation

With the cleaned corpus ready, TF-IDF Weighting was chosen because, for customer-review classification, it highlights the most discriminative terms while down-weighting common, uninformative words, produces higher-quality features that improve model accuracy, and has demonstrated strong effectiveness across numerous text-classification tasks.



Apply TF-IDF transformation

```python
# Extract clean reviews and labels from train and test data set
X_train = train_df['clean_text']
y_train = train_df['is_positive_sentiment']
X_test = x_test['clean_text']
y_test = y_test['is_positive_sentiment']
```

```python
# Apply TF-IDF transformation on X_train and X_test
import numpy as np
from sklearn.feature_extraction.text import TfidfVectorizer

tfidf_vectorizer = TfidfVectorizer(
    max_features=5000,
    ngram_range=(1, 2),      # Include unigrams and bigrams
    min_df=2,                # Ignore terms in fewer than 2 documents
    sublinear_tf=True,       # Apply sublinear term frequency scaling
    stop_words='english'     # Remove English stopwords
)
X_train_tfidf = tfidf_vectorizer.fit_transform(X_train).toarray()
X_test_tfidf = tfidf_vectorizer.transform(X_test).toarray()
```

Vocabulary: The TfidfVectorizer built a vocabulary of the 5000 most frequent terms (max_features=5000) to balance informativeness and computational efficiency. Stopwords (e.g., "the," "and") were removed using stop_words='english' to focus on sentiment terms, and min_df=2 excluded terms appearing in fewer than two reviews to reduce noise. The vocabulary, fixed at ~5000 terms, was consistent across folds, as TF-IDF was fitted before splitting the training set.

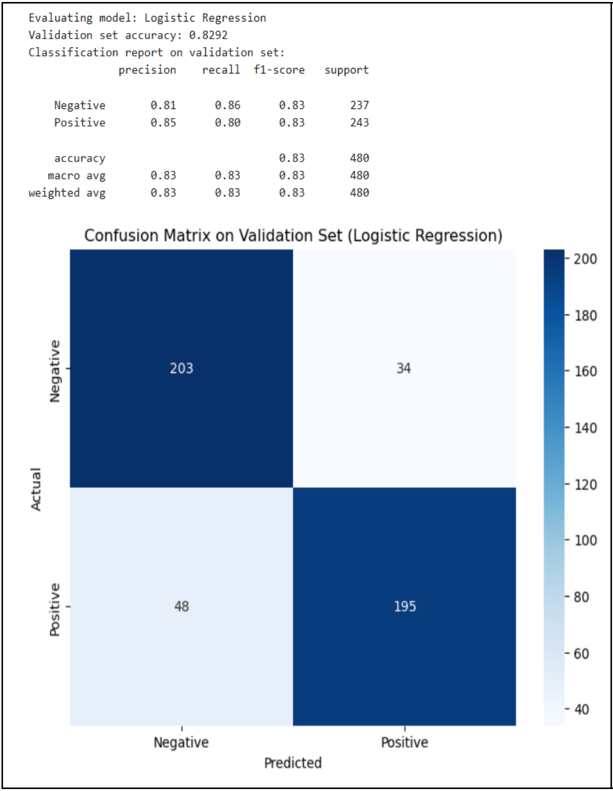
N-grams: I used ngram_range=(1, 2) to include unigrams (e.g., "great") and bigrams (e.g., "not good"), capturing single words and short phrases critical for sentiment (e.g., "customer service"). sublinear_tf=True mitigated the impact of overly frequent terms, enhancing robustness for varied review lengths.

Out-of-Vocabulary Words: The vectorizer was fitted on the training set only, and test set words absent from the training vocabulary were ignored, ensuring no data leakage and maintaining generalizability.

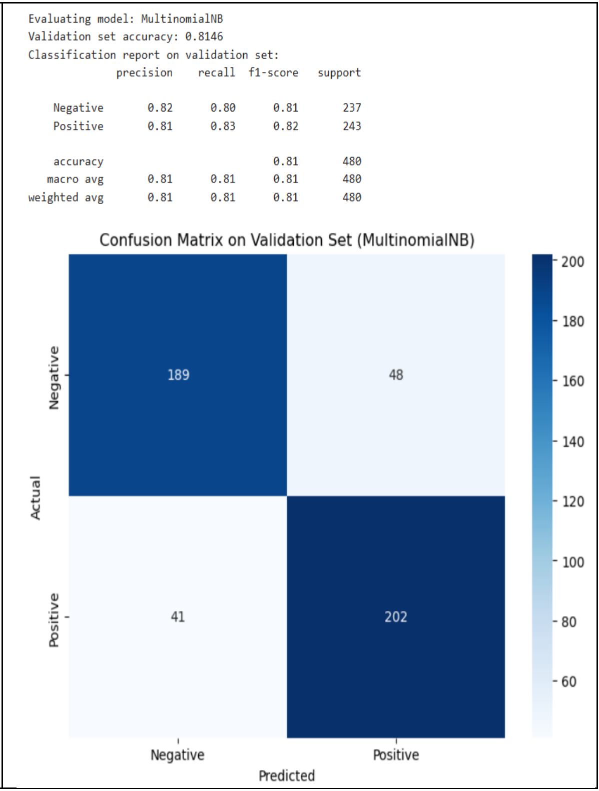
I believe this pipeline, with its balance between expressiveness and robustness, formed a reliable foundation for the classification model.

4.2 Cross Validation and Hyperparameter Selection Design Description

Before conducting the Cross validation and Hyperparameter Selection step, I was confused in choosing the classification model to build. In the problem of classifying customer reviews (positive/negative) based on feedback content, both Logistic Regression and Multinomial Naive Bayes (MultinomialNB) are popular and effective choices for text classification. To decide properly, I tested both models on the data and compared the performance (accuracy, F1-score, confusion matrix). These are the result:



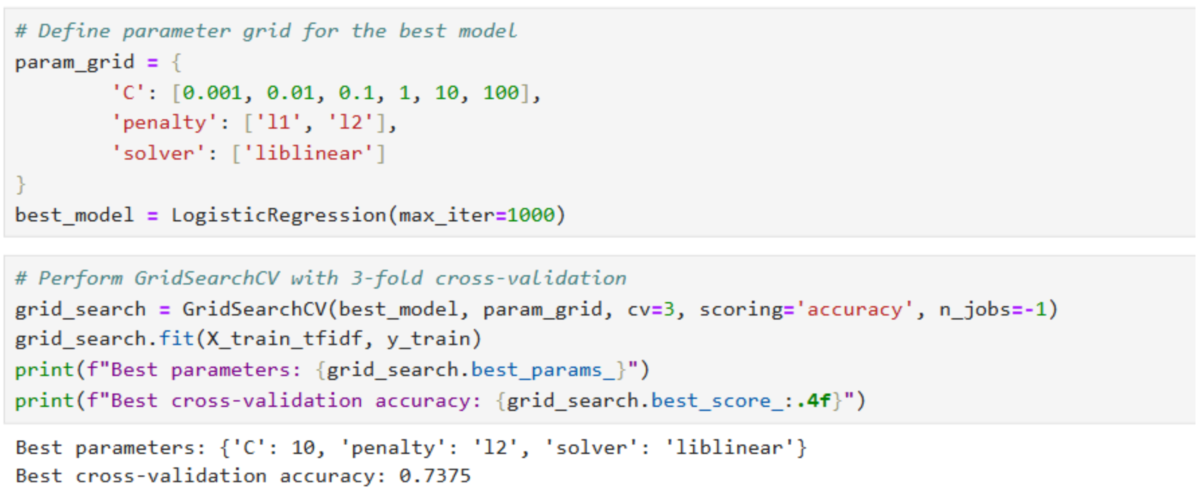
```
Evaluating model: Logistic Regression
Validation set accuracy: 0.8292
Classification report on validation set:
              precision    recall  f1-score   support

    Negative       0.81      0.86      0.83       237
    Positive       0.85      0.80      0.83       243

    accuracy                           0.83       480
   macro avg       0.83      0.83      0.83       480
weighted avg       0.83      0.83      0.83       480
```



```
Evaluating model: MultinomialNB
Validation set accuracy: 0.8146
Classification report on validation set:
              precision    recall  f1-score   support

    Negative       0.82      0.80      0.81       237
    Positive       0.81      0.83      0.82       243

    accuracy                           0.81       480
   macro avg       0.81      0.81      0.81       480
weighted avg       0.81      0.81      0.81       480
```

In general, Logistic Regression has a sightly better performance than the MultinomialNB model (LR correctly labels about 83% of the validation sentences versus 81.5% for NB). The F_1-score balances precision and recall. Looking at the macro-average F_1 (simple average across the two classes), LR's 0.83 beats NB's 0.81. That's the reason Logistic Regression was chosen to be my classifier.

To optimize the classifier pipeline, we used 3-fold cross-validation (CV) and grid search to select hyperparameters, ensuring robust performance on unseen data. The performance metric chosen was **accuracy**, as it balances correct predictions for both positive and negative sentiments, suitable for our balanced dataset (~1200 positive, ~1200 negative training samples)

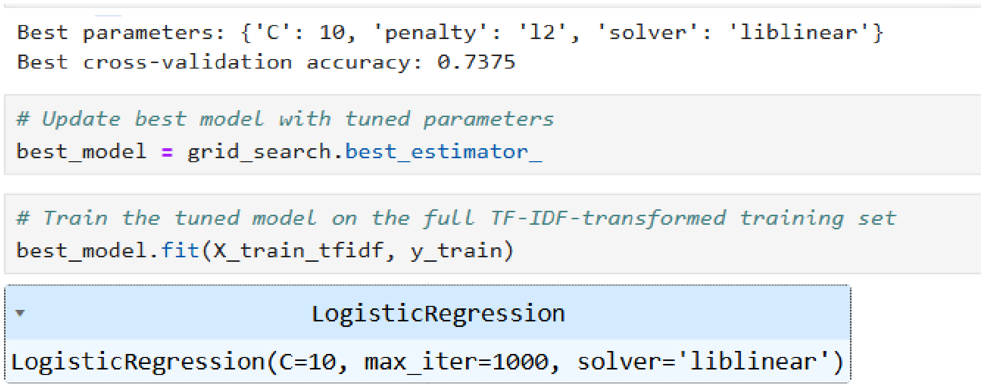


```python
# Define parameter grid for the best model
param_grid = {
        'C': [0.001, 0.01, 0.1, 1, 10, 100],
        'penalty': ['l1', 'l2'],
        'solver': ['liblinear']
}
best_model = LogisticRegression(max_iter=1000)
```

```python
# Perform GridSearchCV with 3-fold cross-validation
grid_search = GridSearchCV(best_model, param_grid, cv=3, scoring='accuracy', n_jobs=-1)
grid_search.fit(X_train_tfidf, y_train)
print(f"Best parameters: {grid_search.best_params_}")
print(f"Best cross-validation accuracy: {grid_search.best_score_:.4f}")
```

```
Best parameters: {'C': 10, 'penalty': 'l2', 'solver': 'liblinear'}
Best cross-validation accuracy: 0.7375
```

Grid search tested configurations for Logistic Regression with the parameter grid: C: [0.001, 0.01, 0.1, 1, 10, 100], penalty: ['l1', 'l2'], and solver: ['liblinear']. The heldout data came from the 2400-sample training set, split into three folds of ~800 samples each, using random splitting to ensure representative subsets. TF-IDF was applied before splitting to maintain a consistent vocabulary. After selecting the best hyperparameters via CV, the final model, with the best hyperparameters applied, was trained on the entire training set (2400 samples) using the optimal configuration, ensuring maximum data utilization for test set predictions.



```
Best parameters: {'C': 10, 'penalty': 'l2', 'solver': 'liblinear'}
Best cross-validation accuracy: 0.7375
```

```python
# Update best model with tuned parameters
best_model = grid_search.best_estimator_
```

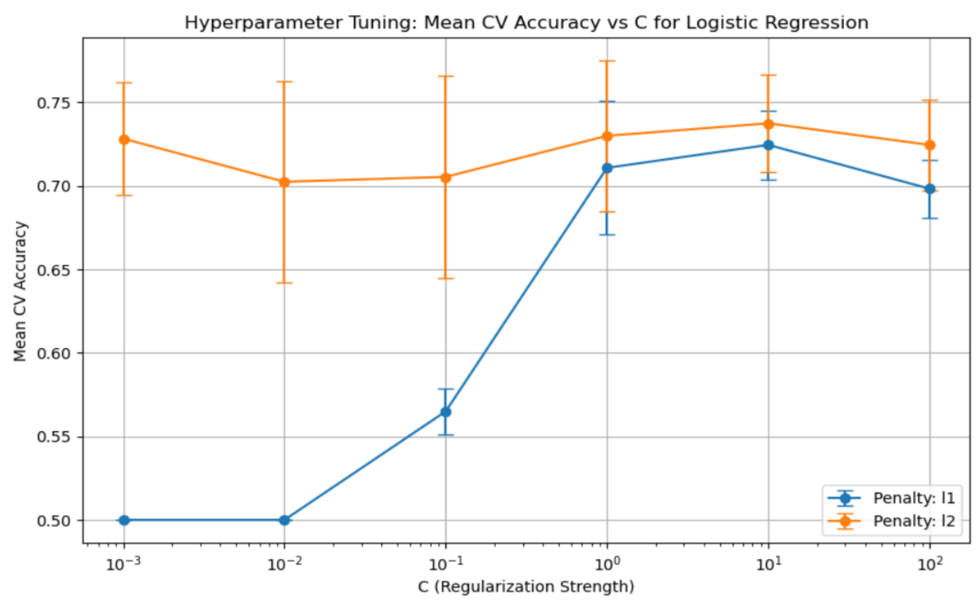
```python
# Train the tuned model on the full TF-IDF-transformed training set
best_model.fit(X_train_tfidf, y_train)
```

```
▾ LogisticRegression
LogisticRegression(C=10, max_iter=1000, solver='liblinear')
```

This approach balanced computational efficiency (3 folds) and generalization, with accuracy as a clear metric for binary classification.

4.3 Hyperparameter Selection Figure for Classifier

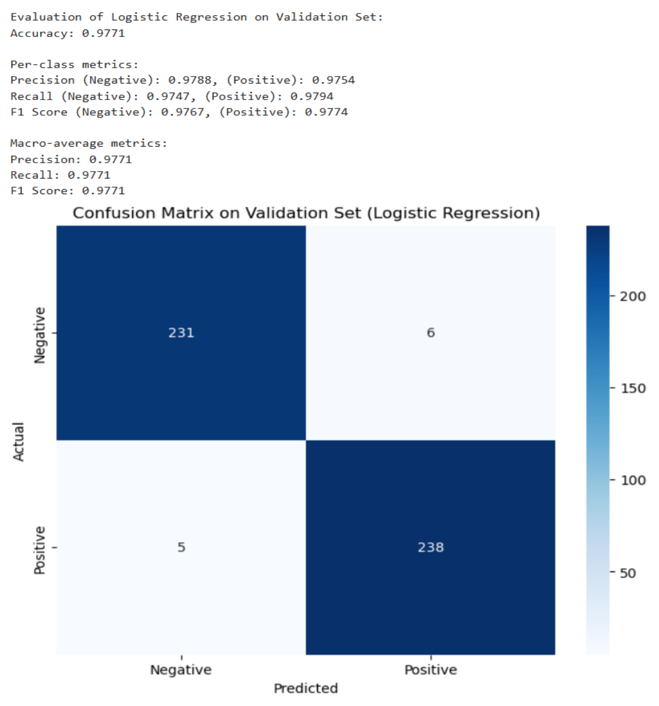
Using the BoW pipeline with Logistic Regression, we conducted a 3-fold cross-validation grid search to optimize hyperparameters (C: [0.001, 0.01, 0.1, 1, 10, 100], penalty: ['l1', 'l2'], solver: ['liblinear']) for maximum accuracy on heldout data. The search evaluated 12 configurations across three folds of the 2400-sample training set (~800 samples per fold). The best configuration was C=10, penalty='l2', solver='liblinear', achieving a cross-validation accuracy of 0.7375. Figure visualizes the mean CV accuracy for each C value, grouped by penalty type:



The Figure shows that l2 regularization with C=10 outperformed others, balancing model complexity and fit. Lower C values (e.g., 0.001) underfit, while l1 regularization scored lower due to its sparsity-inducing nature, less suited for TF-IDF features. The 'l2' line peaks at C=10 (0.7375), while 'l1' peaks lower (~0.72). Error bars indicate standard deviation across folds, with small variation (~0.02), confirming stable performance. The plot highlights C=10, penalty='l2' as the optimal configuration.

4.4 Analysis of Predictions for the Classifier

The Logistic Regression classifier (with C=10, penalty='l2') was evaluated on the validation set (480 samples, 20% of training data) using confusion matrix, accuracy, precision, recall, and F1 score.



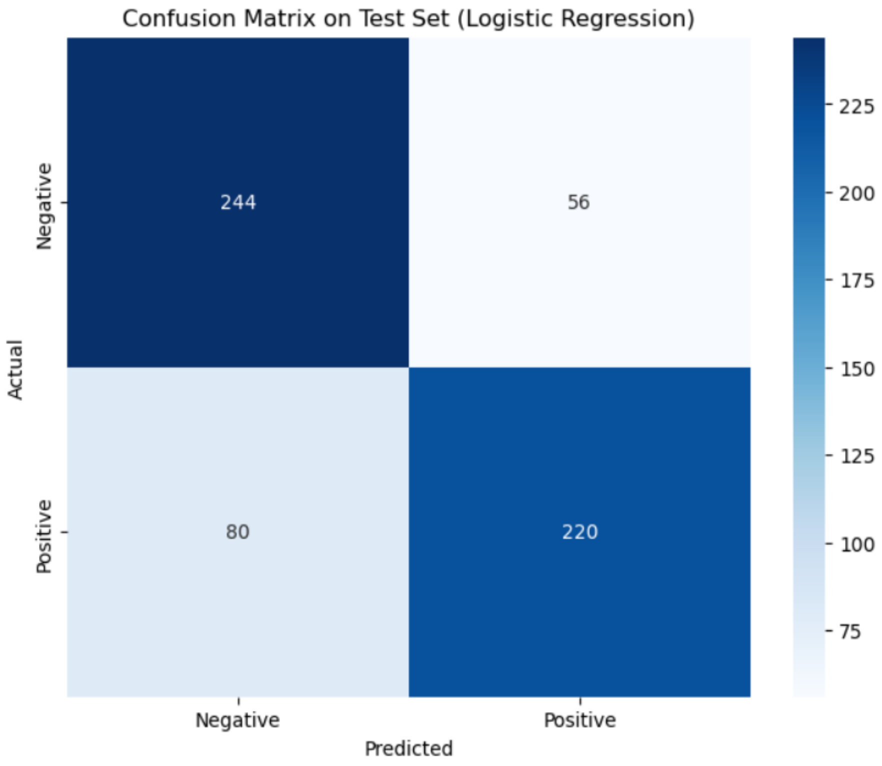
```
Evaluation of Logistic Regression on Validation Set:
Accuracy: 0.9771

Per-class metrics:
Precision (Negative): 0.9788, (Positive): 0.9754
Recall (Negative): 0.9747, (Positive): 0.9794
F1 Score (Negative): 0.9767, (Positive): 0.9774

Macro-average metrics:
Precision: 0.9771
Recall: 0.9771
F1 Score: 0.9771
```

The validation accuracy was 0.9771, with per-class metrics: precision (Negative: 0.9788, Positive: 0.9754), recall (Negative: 0.9747, Positive: 0.9794), and F1 score (Negative: 0.9767, Positive: 0.9774). Macro-average metrics were 0.9771 for precision, recall, and F1. The confusion matrix showed excellent performance, with 231 true negatives, 238 true positives, 6 false positives, and 5 false negatives. Table 1 below lists representative false positives and false negatives, analyzed for patterns.

The results from applying the classification model to the validation dataset indicate that the model is overfitting. This outcome is expected, as the model was trained on the entire training set, and the validation set was splitted from that same training data.

4.5 Performance on Test Set

```
Evaluation of Logistic Regression on Test Set:
Accuracy: 0.7733

Per-class metrics:
Precision (Negative): 0.7531, (Positive): 0.7971
Recall (Negative): 0.8133, (Positive): 0.7333
F1 Score (Negative): 0.7821, (Positive): 0.7639

Macro-average metrics:
Precision: 0.7751
Recall: 0.7733
F1 Score: 0.7730
```



The final Logistic Regression model (C=10, penalty='l2') was applied to the test set (600 samples), achieving a test accuracy of 0.7733, with a macro-average F1 of 0.773. It is somewhat stronger at identifying negatives (precision 0.753, recall 0.813, F1 0.782) than positives (precision 0.797, recall 0.733, F1 0.764), as reflected in its higher negative recall and slightly lower positive recall. This means the model misses positive reviews more often than it mislabels negatives, suggesting that targeted improvements—such as refining negation handling or adjusting class weights—could help boost sensitivity to positive sentiment. In comparison to the validation accuracy (0.9771) and cross-validation estimate (0.7375), the test accuracy was lower than validation but higher than CV. The significant drop from validation (~0.2038) suggests overfitting to the validation set. The test set's higher error rate (e.g., 56 false positives, 80 false negatives) indicates challenges with unseen reviews, likely involving negations or ambiguous sentiments. Future improvements could focus on better negation handling and increasing training data diversity.

References

TfidfVectorizer. scikit-learn Machine Learning in Python. https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html#sklearn.feature_extraction.text.TfidfVectorizer

From Slang to Standard: Text Normalization Techniques with Python. (2024, Sep 08). Medium. https://medium.com/@codeasarjun/from-slang-to-standard-text-normalization-techniques-with-python-9a5ca834168b