Comparing the Performance of Multinomial Naive Bayes against DistilBERT for Poem Sentiment Analysis

July 4, 2024

1 I. Introduction

1.1 Introduction to Problem Area

Sentiment analysis involves the automated classification of texts based on their emotional tone. Kim and Klinger identify one definition of sentiment analysis as determining whether an **opinion** expresses “positive or negative feeling” (2019). Others define SA as “a part of natural language processing (NLP), which aims to extract sentiments and opinions from text” (Ilmawan, Muladi, & Prasetya, 2024). However, detecting opinion polarity towards specific entities is merely one application of sentiment analysis, albeit an important one (Jurafsky & Martin, 2024). This project focuses on the classification of lines of poetry into different sentiment polarities or “semantic orientations” (Yusof, Mohamed, & Abdul-Rahman, 2015). Lines of English poetry from Project Gutenberg have been marked with one of four possible labels: 0 indicating “negative sentiment”, 1 indicating “positive sentiment,” 2 indicating “neutral” or “no impact,” and 3 indicating “mixed.” The main aim of this project is to assess and compare the performance of a traditional statistical algorithm (Naïve Bayes) on this classification task with that of a modern deep-learning model (DistilBERT). Unlike more common sentiment analysis applications which focus on **attitudes** towards entities, poetry frequently expresses general affective states such as moods and emotions (Jurafsky & Martin, 2024).

The vast availability of text data on the Web since the early 2000s has established sentiment analysis as a valuable research field. Liu (2012) delineates how the sentiment analysis of online customer reviews or social media comments is pivotal to much of the current decision-making processes in the business domain. Moreover, analysing online attitudes can assist in predicting election results and major sociopolitical trends. While the advantages of sentiment analysis for measuring customer satisfaction or political moods have been extensively documented, the rationale for sentiment analysis for poetic texts is not immediately apparent.

Poetic language constitutes a specific challenge for any text classification task due to the prevalence of figurative language and unconventional word usage (Kim & Klinger, 2019). Metaphorical and figurative expressions are also used in customer reviews and social media posts; consequently, exploring the challenges raised during the computational detection of sentiment polarity in poetic language may improve the robustness of more general sentiment analysis techniques.

Furthermore, Kim and Klinger argue that sentiment analysis techniques can enrich the growing discipline of the “digital humanities” (Kim & Klinger, 2019). They can shed important new perspectives on research topics in literary studies, such as the evolution of literary expressions of emotion over time or authorship attribution (Jurafsky & Martin, 2024). Additionally, sentiment analysis of literary texts can also be used to facilitate the detection of bias towards certain demographic groups over time. Sheng and Uthus, who compiled and annotated the dataset used for this project, did so to counteract the societal bias of an automated poetry collaboration tool (Sheng & Uthus, 2020). As such, the field of sentiment analysis of poetic language can be valuable for refining NLP models' capability to handle figurative language, assisting in the formulation and testing of theories in literary research, and diagnosing social biases in seminal cultural texts..

WordCount: 498

1.2 Objectives

1.2.1 Primary Objective

- The primary goal of this project is to compare the performance of a statistical text classification algorithm (Multinomial Naive-Bayes) on this poetry sentiment polarity detection dataset (Sheng & Uthus, 2020) with that of a modern transformer-based model (DistilBERT).

1.2.2 Methodology

- The current state-of-the-art performance scores on this dataset, as of June 2024, are 89% accuracy, 92% precision, 88% recall and 90% F1, achieved by AiManatee on HuggingFace using a fine-tuned version of RoBERTa (AiManatee, 2024). This score even outperforms the original authors' accuracy score of 84.6% on the test set (Sheng & Uthus, 2020).
- First, a baseline evaluation of performance will be calculated using a simple Multinomial Naïve Bayes Classifier without using complex feature-construction methods, before comparing this to the impact on performance made by various optimization techniques.
- The project will compare the effectiveness of both classifiers on the original training-validation-test split, in order to facilitate comparability with other researchers' scores.
- However, in order to be able to truly evaluate the classifiers' robustness, the original dataset will *also* be recombined into new training and test portions, each of which will contain a proportional representation of each class. The same experiments will be repeated using K-fold cross-validation. This aims to more completely assess the predictive power of the two classification algorithms on this dataset while addressing its limitations (its small size and unbalanced nature).
- Results for each of the experiments will be tabulated and visualized using charts and confusion matrices.

Potential Contributions

- Deep-learning transformer-based models constitute a kind of opaque "black box", where excellent results can come at the cost of the interpretability provided by statistical models (Tunstall., von Werra, & Wolf., 2022). As such, this project will explore whether the applying different feature-engineering experiments on texts used to train a classical statistical classifier can potentially yield comparable results to a powerful deep-learning model.
- Additionally, this study seeks to examine the impact of deep-learning models on the classification of emotion in texts containing extremely ambiguous and figurative language. Nan Da claims (2019) that there is no case for computational techniques in literary studies due to their over-reliance on "simple word counts". This project aims to either affirm or undermine

her claim by showing that transformer-based models have the ability to model the rich contextual relationships between words, and may thus dramatically improve text classification tasks even on literary language (Tunstall, von Werra, & Wolf, 2022).

Summary

- Overall, this study hopes to determine whether applying deep-learning to sentiment analysis can improve the predictive power of computational techniques for classifying figurative, metaphorical and indirect expressions of emotion. Meanwhile, it will also explore whether using certain feature-engineering techniques, such as negation handling or text normalization, can improve the performance of simpler, statistical algorithms such as Naïve Bayes on this task, due to the greater interpretability and lower computational cost associated with traditional classifiers.

W.C.: 489

1.3 Dataset Description

The dataset used here is the Google Research Datasets Poem Sentiment dataset. It was constructed from random verses from the Gutenberg Poem Dataset by Emily Sheng and David Uthus (a developer at Google). The purpose of creating the dataset was to develop techniques mitigating societal bias for a collaborative "poetry composition system" (Sheng & Uthus, 2020). The dataset is licensed under the Creative Commons Attribution 4.0 International License, which can be found here (Creative Commons, n.d.). It allows the user to copy, share, adapt and remix "the material for any purpose, even commercially" as long as attribution is given, and a link to the license is provided. The attribution is: *"Sheng, E., & Uthus, D. (2020). Investigating Societal Biases in a Poetry Composition System. arXiv. Retrieved June 15, 2024, from https://arxiv.org/abs/2011.02686"*. Additionally, the dataset was added to HuggingFace by Suraj Patil (link to GitHub page).

The dataset downloaded from Hugging Face has **already been split into three parts**: the training, validation, and test set. The train set contains 892 samples, the validation set 105 samples, and the test set 104. As can be seen in the code below, each sample in the dataset consists of an 'id' field (an integer, with the count starting from 0), a 'verse_text' field which is the string of poetry that is to be classified, and finally an integer representing the sentiment polarity of the verse_text, with 0 for negative, 1 for positive, 2 for "no impact" (neutral) and 3 for "mixed" (both negative and positive).

According to Sheng and Uthus, at the time of publication (2020), there was "no existing public poetry dataset with sentiment annotations". I was unable to successfully locate any alternative English language poetry dataset with sentiment scores either (in June 2024). The authors employed two expert annotators to label the extracts of poetic language. The "Cohen's kappa" inter-annotator agreement score was 0.53 when all possible labels (including "mixed" sentiment) were included, but increased to 0.58 when these ambiguous/mixed samples were removed. Cohen's kappa measures "how often the annotators may agree with each other" (Wang, Yang, & Xia, 2019, p. 164387). A score between 0.41–0.60 indicates "moderate agreement" between the annotators. Additionally, Spearman's correlation for the samples in the basic positive/neutral/negative categories was 0.67 - which Sheng and Uthus state shows substantial inter-annotator agreement. The authors state that they only kept the sample if there was agreement across both annotators. Before training the

BERT model, they filtered the samples to keep only those with a "negative", "no impact" (neutral) and "positive" labels - the "mixed" lines of poetry were removed. Thus, the accuracy score of 84.6% achieved here was based on excluding any "mixed" samples in either the training or test data. As shown in the code below, one can see that although the training set contains 49 instances of the "mixed" class, the validation and test sets do not contain any samples from this class. The evaluation protocol defined below will outline how this shortcoming will be handled.

W.C.: 496

---

[5]:
```
import os # for storing logs outputted by model training
import copy
# Import the "datasets" library that allows downloading datasets from Hugging␣
↪Face
import datasets
# Enable loading the remote copy of the dataset with this function
from datasets import load_dataset
# Enable loading the local copy of the dataset with this function
from datasets import load_from_disk
# Save important model hyperparameter configurations using pickle
import pickle
import pandas as pd
import numpy as np
# Download nltk NLP functionality
import nltk
from nltk.tag import pos_tag
from nltk.tokenize import word_tokenize
nltk.download('punkt')
from nltk.classify import accuracy
from nltk.corpus import stopwords
from nltk import bigrams
nltk.download('stopwords')
from nltk.stem import WordNetLemmatizer
## Import wordnet functionality for negation handling
from nltk.corpus import wordnet
nltk.download('wordnet')
# String processing and regex functionality
import string
# Used to flatten lists (e.g. token lists into one long vocabulary list)
from itertools import chain
import re
# Import plotting library for plotting confusion matrices
import matplotlib.pyplot as plt

# Scikit-Learn functionality
# TF-IDF functionality
from sklearn.feature_extraction.text import TfidfVectorizer
# Import evaluation metrics
```

```python
from sklearn.metrics import accuracy_score, f1_score, ␣
↪precision_recall_fscore_support, classification_report, confusion_matrix
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.naive_bayes import MultinomialNB
# Import train_test_split for stratified dataset splitting to maintain␣
↪proportions of each class in each split for cross-validation
from sklearn.model_selection import train_test_split
# Allow stratified k-fold cross-validation to address challenges of dealing␣
↪with an unbalanced dataset.
from sklearn.model_selection import StratifiedKFold
from sklearn.model_selection import ParameterGrid

# Import sentiment lexicons for feature engineering tasks
from afinn import Afinn
from nltk.corpus import sentiwordnet as swn
nltk.download('sentiwordnet')

# Import hyperparameter tuning functionality from ray
import ray
from ray import tune
# Functionality for Bayesian Optimization of hyperparameters
from ray.tune.schedulers import ASHAScheduler
from ray.tune.search.bayesopt import BayesOptSearch

# Import deep-learning model functionality
from transformers import DistilBertTokenizer,␣
↪DistilBertForSequenceClassification, Trainer, TrainingArguments
import torch
from torch.utils.data import Dataset, DataLoader
```

```
[nltk_data] Downloading package punkt to
[nltk_data]     C:\Users\ophel\AppData\Roaming\nltk_data…
[nltk_data]   Package punkt is already up-to-date!
[nltk_data] Downloading package stopwords to
[nltk_data]     C:\Users\ophel\AppData\Roaming\nltk_data…
[nltk_data]   Package stopwords is already up-to-date!
[nltk_data] Downloading package wordnet to
[nltk_data]     C:\Users\ophel\AppData\Roaming\nltk_data…
[nltk_data]   Package wordnet is already up-to-date!
[nltk_data] Downloading package sentiwordnet to
[nltk_data]     C:\Users\ophel\AppData\Roaming\nltk_data…
[nltk_data]   Package sentiwordnet is already up-to-date!
```

[3]:

```python
# Loads in the poem-sentiment dataset from Hugging Face --> link: https://
↪huggingface.co/datasets/google-research-datasets/poem_sentiment
dataset = load_dataset("google-research-datasets/poem_sentiment")
```

[4]:
```
# Print a summary of the different splits in the poem sentiment dataset
print(dataset)
```

```
DatasetDict({
    train: Dataset({
        features: ['id', 'verse_text', 'label'],
        num_rows: 892
    })
    validation: Dataset({
        features: ['id', 'verse_text', 'label'],
        num_rows: 105
    })
    test: Dataset({
        features: ['id', 'verse_text', 'label'],
        num_rows: 104
    })
})
```

[22]:
```
# Output the first twenty samples in the train part of the dataset
print(dataset['train'][0:20])

# Output the first five samples in the validation part of the dataset
print(dataset['validation'][0:5])

# Output the first five samples in the test part of the dataset
print(dataset['test'][0:5])
```

```
{'id': [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19],
'verse_text': ['with pale blue berries. in these peaceful shades--', 'it flows
so long as falls the rain,', 'and that is why, the lonesome day,', 'when i
peruse the conquered fame of heroes, and the victories of mighty generals, i do
not envy the generals,', 'of inward strife for truth and liberty.', 'the red
sword sealed their vows!', 'and very venus of a pipe.', 'who the man, who,
called a brother.', 'and so on. then a worthless gaud or two,', 'to hide the orb
of truth--and every throne', "the call's more urgent when he journeys slow.",
"with the _quart d'heure_ of rabelais!", 'and match, and bend, and thorough-
blend, in her colossal form and face.', 'have i played in different countries.',
'tells us that the day is ended."', 'and not alone by gold;', 'that has a
charmingly bourbon air.', "sounded o'er earth and sea its blast of war,", 'chief
poet on the tiber-side', 'as under a sunbeam a cloud ascends,'], 'label': [1, 2,
0, 3, 3, 3, 2, 2, 0, 2, 2, 2, 2, 2, 2, 1, 0, 2, 2]}
{'id': [0, 1, 2, 3, 4], 'verse_text': ['to water, cloudlike on the bush afar,',
'shall yet be glad for him, and he shall bless', 'on its windy site uplifting
gabled roof and palisade,', '(if haply the dark will of fate', 'jehovah, jove,
or lord!'], 'label': [2, 1, 2, 0, 2]}
{'id': [0, 1, 2, 3, 4], 'verse_text': ['my canoe to make more steady,', 'and be
glad in the summer morning when the kindred ride on their way;', 'and when they
reached the strait symplegades', 'she sought for flowers', 'if they are hungry,
```

```
paradise'], 'label': [2, 1, 2, 2, 2]}
```

[ ]:
```python
# # Saves this dataset locally inside the 'data' sub-directory using the
 ↪built-in dataset's 'save_to_disk' method
# original_dir = './datasets/original_poem_sentiment_dataset'
# dataset.save_to_disk(original_dir)
```

[6]:
```python
# Loads in the HuggingFace poem dataset from local storage and stores it in a
 ↪variable called "poem_dataset"
original_dir = './datasets/original_poem_sentiment_dataset'
poem_dataset = load_from_disk(original_dir)
print(f"Original dataset: {poem_dataset}")

# Convert the three splits into pandas dataframes for easier viewing and
 ↪analysis of the dataset using the inbuilt 'to_pandas' method
train_df = poem_dataset['train'].to_pandas()
val_df = poem_dataset['validation'].to_pandas()
test_df = poem_dataset['test'].to_pandas()

# Display the first 10 lines in each of the training, val and test data splits
print("TRAIN DATA")
print(train_df.head(10))
print('\n**********************************************************************\n')
print("VALIDATION DATA")
print(val_df.head(10))
print('\n**********************************************************************\n')
print("TEST DATA")
print(test_df.head(10))

## Save DataFrames as .csv files
train_df.to_csv('original_train_df.csv', index=False)
val_df.to_csv('original_val_df.csv', index=False)
test_df.to_csv('original_test_df.csv', index=False)
```

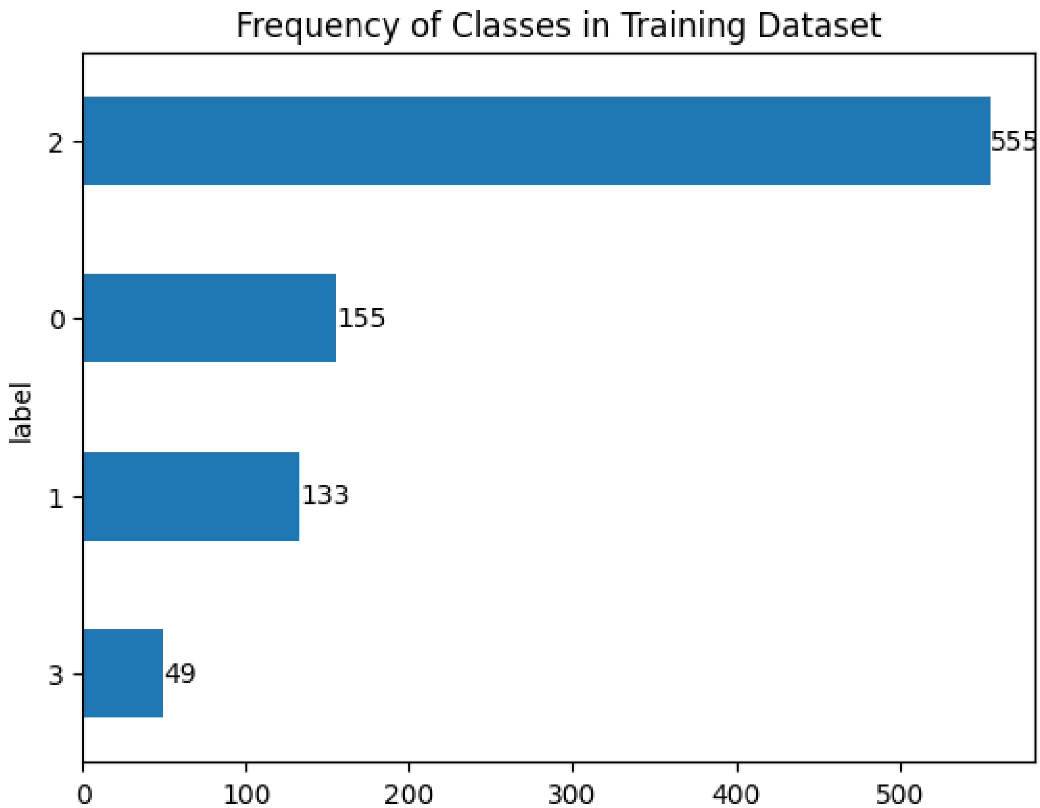
```
Original dataset: DatasetDict({
    train: Dataset({
        features: ['id', 'verse_text', 'label'],
        num_rows: 892
    })
    validation: Dataset({
        features: ['id', 'verse_text', 'label'],
        num_rows: 105
    })
    test: Dataset({
        features: ['id', 'verse_text', 'label'],
        num_rows: 104
    })
```

```
})
TRAIN DATA
   id                                         verse_text  label
0   0  with pale blue berries. in these peaceful shad...      1
1   1                it flows so long as falls the rain,      2
2   2                 and that is why, the lonesome day,      0
3   3  when i peruse the conquered fame of heroes, an...      3
4   4          of inward strife for truth and liberty.      3
5   5                     the red sword sealed their vows!      3
6   6                            and very venus of a pipe.      2
7   7                 who the man, who, called a brother.      2
8   8          and so on. then a worthless gaud or two,      0
9   9        to hide the orb of truth--and every throne      2

**********************************************************************

VALIDATION DATA
   id                                         verse_text  label
0   0              to water, cloudlike on the bush afar,      2
1   1      shall yet be glad for him, and he shall bless      1
2   2  on its windy site uplifting gabled roof and pa...      2
3   3                    (if haply the dark will of fate      0
4   4                             jehovah, jove, or lord!      2
5   5        when the brow is cold as the marble stone,      0
6   6        taking and giving radiance, and the slopes      1
7   7                       press hard the hostile towers!      0
8   8     his head is bowed. he thinks on men and kings.      2
9   9                     with england if the day go hard,      2

**********************************************************************

TEST DATA
   id                                         verse_text  label
0   0                       my canoe to make more steady,      2
1   1  and be glad in the summer morning when the kin...      1
2   2         and when they reached the strait symplegades      2
3   3                                 she sought for flowers      2
4   4                       if they are hungry, paradise      2
5   5                        indignantly i hurled the cry:      0
6   6                     with which his house is haunted;      0
7   7    and, laying snow-white flowers against my hair.      2
8   8         of long-uncoupled bed, and childless eld,      2
9   9  of the boulder-strewn mountain, and when they ...      2
```

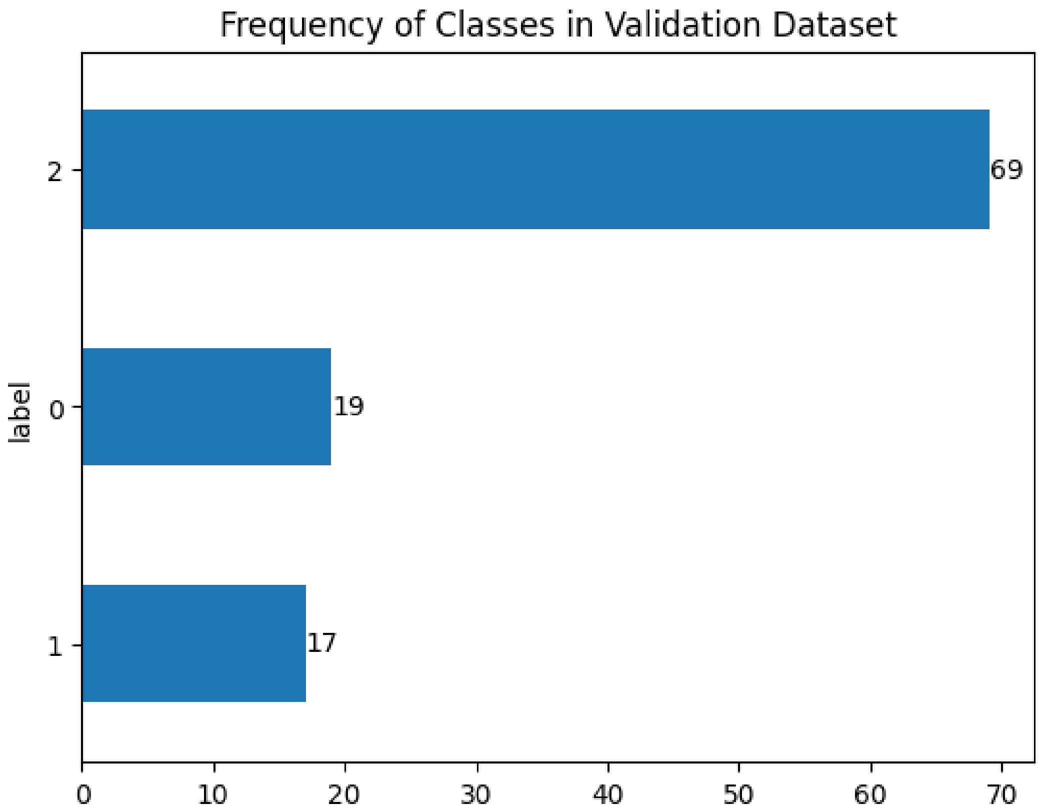
1.3.1 Visualizations of Class Distributions in Training, Validation and Test Sets

In this section, I will visually represent the class counts in each split of this dataset:

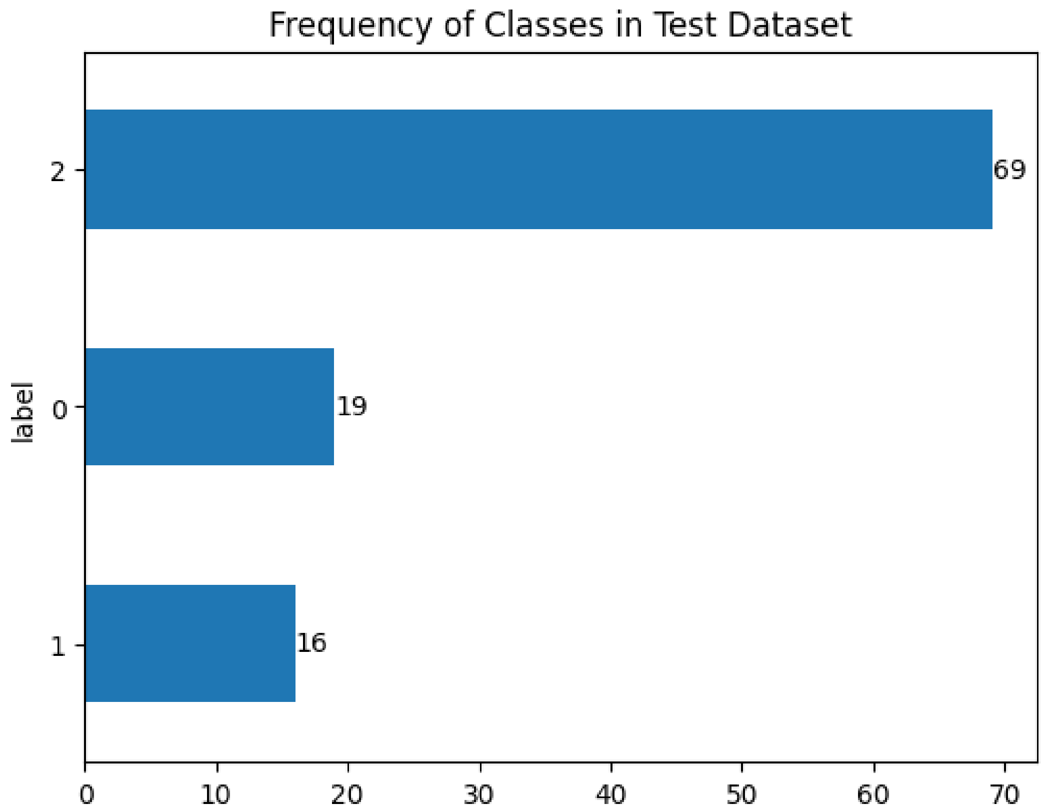
[13]:
```python
# Get value counts for each label type (0-neg, 1-pos, 2-neutral,␣
↪3-mixed_sentiment)
train_class_counts = train_df["label"].value_counts(ascending=True)
# Plot the train label counts on a simple horizontal bar chart
plot = train_class_counts.plot.barh()
plt.title("Frequency of Classes in Training Dataset")
# Annotate each bar with the counts for that class
for i, value in enumerate(train_class_counts):
    plot.text(value, i, str(value), va='center')
```



[16]:
```python
# Visualize class distribution in the validation set

# Get value counts for each label type (0-neg, 1-pos, 2-neutral,␣
↪3-mixedSentiment) in the validation dataset
val_class_counts = val_df["label"].value_counts(ascending=True)
# Plot the label counts on a simple horizontal bar chart
plot = val_class_counts.plot.barh()
plt.title("Frequency of Classes in Validation Dataset")
# Annotate each bar with the counts for that class
for i, value in enumerate(val_class_counts):
    plot.text(value, i, str(value), va='center')
```



[17]:
```
# Visualize class distributions in the test set

# Get value counts for each label type (0-neg, 1-pos, 2-neutral,␣
↪3-mixedSentiment) in the test dataset
test_class_counts = test_df["label"].value_counts(ascending=True)
# Plot the label counts on a simple horizontal bar chart
plot = test_class_counts.plot.barh()
plt.title("Frequency of Classes in Test Dataset")
# Annotate each bar with the counts for that class
for i, value in enumerate(test_class_counts):
    plot.text(value, i, str(value), va='center')
```



As can be seen here, only the training split contains (49) instances of class 3 (mixed sentiment)! The dataset is deeply unbalanced.

1.4 Evaluation Methodology

1.4.1 Calculating a Baseline

- First, the samples will be pre-processed using only the basic techniques mandatory for constructing inputs to NLTK's Multinomial Naïve Bayes classifier.
- The optimal number of word features to use for the baseline will be selected using a line graph showing performance for different word feature counts.
- This basic classifier will then be trained on the training set and evaluated on the validation set to create a benchmark.
- F1, recall and precision scores as well as accuracy will be used to measure performance. Relying on accuracy alone may lead to overly optimistic conclusions due to class imbalance – one can achieve a seemingly high accuracy score of 66% by simply predicting the majority class every time on the validation set, misrepresenting the classifier's predictive power.
- Macro-averages will be used to evaluate F1, precision and recall across classes. Micro-averaging can "overemphasize the performance on the majority class", thus resulting in "inflated" performance scores when the algorithm performs poorly on the minority classes.

Macro-averages are thus more useful when determining the performance on each class is "equally important" and when dealing with a heavily unbalanced dataset (EvidentlyAI, 2024).

1.4.2 Experiments on Original Dataset Split

- Several experiments will be run by training a Multinomial Naïve Bayes classifier on the original split's validation set to compare a broad range of text-processing and feature-extraction techniques.
- The same metrics mentioned above will be used to compare effectiveness of different features.
- The highest-performing feature extracting pipeline will be used to evaluate the performance of a Naïve Bayes classifier on the original test set.

1.4.3 Experiments on Recombined Dataset Split

- The samples will be merged into one dataset, before creating a stratified train-test split ensuring the proportional representation of each class in each split, to address the class imbalance problem visualized in the graphs above.
- Experiments will be repeated using **stratified** five-fold cross-validation. With ordinary cross-validation, the splits might not be "representative of the overall data distribution" (Nagaraj, 2023). If a fold lacks "mixed" samples, Naïve Bayes would struggle with the "zero probability problem" (Jayaswal, 2020). The dataset is relatively small and unbalanced. Consequently, cross-validation ensures that all of the training data is used. It also reduces the chance of overfitting to the validation set.
- Mean accuracy and macro-average F1 scores for each experiment will be collected to select the best-performing feature-extraction pipeline for evaluation on the new test set.

1.4.4 Deep-Learning Model Evaluation

- Both dataset splits will also be used to compare the performance of the transformer-based DistilBERT model.
- The validation set (for the original split) and stratified cross-validation (for the recombined dataset) will be used again for hyperparameter tuning.
- Models with the highest-performing configuration will be trained on the test sets.
- Tables and confusion matrices will be used throughout to compare the different models and techniques, before conducting a critical analysis of the differences in the statistical and deep-learning models' performances.

W.C.: 495

---

2 II. Implementation

2.1 Basic Text Pre-Processing and Baseline Calculation

A baseline is useful way of providing quick checks before exploring more powerful, deep-learning models: for example, if a large BERT model results in an accuracy score of 80%, you might be simply conclude that the model performed reasonably well. However, if a simple classifier like Naive Bayes or Logistic Regression obtains a 95% score, this might "prompt you to debug your model" and analyze the problems with the more complex model (Tunstall, von Werra, & Wolf, 2022).

Here, the baseline accuracy, F1, precision and recall scores will be calculated using a simple Multinomial Naive Bayes classifier trained on the original "train" split of the data and evaluated on the original "validation" split (for comparison purposes with other researchers' results).

For the baseline benchmark, each sample will undergo only the basic pre-processing using the following methods:

- Tokenizing using the *nltk* word_tokenize function
- Binary vectorization - converting each line of poetry into a vector of 0s and 1s. Each value representing the presence (1) or absence (0) of a type in a subset of the total vocabulary. In this particular case, to construct inputs appropriate for the NLTK Naive Bayes classifier, a dictionary storing True or False for each vocabulary term will be used as an equivalent way of expressing this concept.

The words in the lines of poetry have already all been converted into lowercase by the dataset creators, therefore this basic pre-processing step can be skipped.

As Jurafsky and Martin note (2024), "for sentiment classification and a number of other text classification tasks, whether a word occurs or not seems to matter more than its frequency. Thus it often improves performance to clip the word counts in each document at 1". This technique is called "binary Naive Bayes". As word occurence matters more than frequency in the context of this particular type of text classification (sentiment analysis), this encoding technique will be used for calculating the baseline.

After calculating the baseline scores, more advanced feature-engineering experiments will be conducted and compared to the baseline scores to see which feature construction technique best outperforms the baseline.

2.1.1 Why Naïve Bayes?

- A Multinomial Naive Bayes Classifier will be used for, first, calculating the baseline score (using only the most basic text pre-processing required), and, second, for running the different feature-engineering experiments to evaluate and compare the performance of a statistical algorithm to a deep-learning model.

Advantages of Naive Bayes

- The advantages of using the Naive Bayes classifier include its speed and simplicity, and its lack of reliance on tuneable hyperparameters. It can provide "a quick-and-dirty baseline for a classification problem". Additionally, Naive Bayes tends to perform well when working with high-dimensional data, such as text, as clusters of instances in high dimensions tend to be more separate (Vanderplas, 2016).

Comparison to Logistic Regression Classifiers

- Nevertheless, as Jurafsky and Martin explain (Jurafsky & Martin, 2024), more sophisticated models such as logistic regression have some advantages over Naive Bayes. When Naïve Bayes estimates the likelihood of a class having specific features (word occurences), it works on the assumption that every feature is independent and equally important. However, in real life, features are seldom independent of one another. Furthermore, certain words in a text usually have more weight than others. As such, logistic regression can be more robust when there are many correlated features (Jurafsky & Martin, 2024). The authors conclude that when working with very large datasets *or* long documents (i.e. text samples), logistic regression tends to perform better, but that Naive Bayes can work just as well on short pieces of text (like the samples in this dataset).

Comparison to Support Vector Machines (SVMs)

- Although support vector machines have been applied successfully in many text-classification scenarios(Sharma & Dey, 2012), and techniques exist for making these algorithms more robust, overall "the success of SVM is very limited when it is applied to the problem of learning from imbalanced datasets in which negative instances heavily outnumber the positive instances" (Akbani, Kwek & Japkowicz, 2004). Wang & Manning have also shown that "for short snippet sentiment tasks, NB actually does better than SVMs ("Support Vector Machines"). Moreover, Naive Bayes has been used successfully for decades for spam detection tasks, where datasets are usually extremely unbalanced (a spam email is a much rarer event than a legitimate email) (Jurafsky & Martin, 2024).

Comparison to Decision Trees and Random Forests

- A decision tree is another kind of classifier that can be used for sentiment prediction tasks. However, decision trees are very prone to overfitting (and thus performing poorly on unseen data) (Bramer, 2007). Although there are different strategies for mitigating this risk, this involves extensive experimentation with hyperparameters such as configuring maximum tree depth or tree pruning settings, which introduces another layer of complexity to the task. Random Forest Classifiers have also been applied successfully to sentiment classification tasks (see this article on applying one to the Twitter sentiment dataset by Bahrawi, 2019) - however, this kind of ensemble classifier (which uses a collection of simple decision trees) has the disadvantage of being less interpretable.

Comparison to Maximum-Entropy Classifiers

- Maximum-Entropy classifiers can frequently perform better than Naive Bayes classifiers on certain datasets (Nigam, Lafferty, & McCallum, 1999), as it does not rely on low correlation between features - however, they take much longer to train and can overly rely on having a large set of training data for optimal performance (Vryniotis, 2013).

As a result, considering the small size and unbalanced nature of this dataset, Naive Bayes seems like a decent starting point for this sentiment analysis task.

[7]:
```
## Convert the "verse_text" (contains samples of poetry) and labels columns␣
↪from pandas Series objects to Python lists
original_dataset_train_samples = train_df['verse_text'].to_list()
original_dataset_train_labels = train_df['label'].to_list()
original_dataset_validation_samples = val_df['verse_text'].to_list()
original_dataset_validation_labels= val_df['label'].to_list()
original_dataset_test_samples = test_df['verse_text'].to_list()
original_dataset_test_labels = test_df['label'].to_list()


# View the first 10 lines of poetry and labels in the original training set
```

```
print(f"FIRST 10 SAMPLES FROM TRAINING DATA: {original_dataset_train_samples[0:
↪10]}")
print(f"FIRST 10 LABELS FROM TRAINING DATA: {original_dataset_train_labels[0:
↪10]}\n")

"""
    As can be seen below, labels of the same kind are not all grouped together␣
↪but randomly spread out,
    thus avoiding the problem of unnatural ordering of the dataset.
"""

# Tokenize each verse_text sample into "words" using the NLTK word_tokenizer
original_dataset_train_tokens = [word_tokenize(sample) for sample in␣
↪original_dataset_train_samples]
original_dataset_validation_tokens = [word_tokenize(sample) for sample in␣
↪original_dataset_validation_samples]
original_dataset_test_tokens = [word_tokenize(sample) for sample in␣
↪original_dataset_test_samples]

# To obtain the entire vocabulary from the training samples, define a method␣
↪that flattens all the token-lists into one list of word tokens
# Reference: https://realpython.com/python-flatten-list/
↪#flattening-a-list-using-standard-library-and-built-in-tools
def flatten_list_of_lists(list_of_lists):
    """
        Flatten a list-of-list (i.e. lists of tokens) into one long list of␣
↪tokens.
        Input: a list of lists.
        Output: a flattened list containing all the elements in the sub-lists.
    """
    return list(chain.from_iterable(list_of_lists))

# Get the vocabulary as a list of tokens
vocabulary_list = flatten_list_of_lists(original_dataset_train_tokens)
# Remove duplicates by turning the vocabulary list into a set.
vocabulary_set = set(vocabulary_list)
# Print the total size of the vocabulary
print(f"\nTOTAL VOCABULARY SIZE: {len(vocabulary_set)}\n")

############################################### THIS CODE IS ADAPTED FROM THE␣
↪COURSE MATERIALS FROM WEEK 10 ###########################################
# Extract the 750 (out of 2304) most frequent words (about 1/3 of the most␣
↪common words) from the vocabulary list using a frequency distribution
N = 750
# Creat a frequency distribution of the words in the vocabulary list (all the␣
↪tokens in the training set)
```

```
all_words = nltk.FreqDist(w for w in vocabulary_list)
# Keep just the top N i.e. 750 words for the baseline calculation.
word_features = list(all_words)[:N]
print(f"TOP 40 TERMS: {list(all_words)[0:40]}") # Print the top 40 most common␣
↪words/terms

# Define an auxiliary function to extract the features-dict for N =␣
↪word_features for each token-set/sample
def doc_features(document, word_features):
    """
        Turns a document/sample (list of tokens) into a dict of word features␣
↪where each key is the word whose
        occurence is to be used as a feature, and each key is "True" or "False"␣
↪indicating whether the word occurs
        in the sample.
        Inputs:
            document = a list of tokens representing a single sample/line of␣
↪poetry.
            word_features = the subset of words from the entire training set to␣
↪use as features.
        Outputs:
            A dictionary for the inputted sample containing key-value pairs␣
↪indicating if each word in
            word_features (subset of training vocabulary) appears in the sample.
    """
    # Use 'set' to remove duplicate words from the document (line of poetry)
    document_words = set(document)
    # Create a features dict to represent the
    features = {}
    # Iterate over the top N vocabulary words (word_features) and create a␣
↪dict-key for that word, with the dict-value signalling whether the
    # word occurs in the document (line of poetry) or not.
    for word in word_features:
        features[f"contains({word})"] = (word in document_words)
    return features
##########################################################################################

# Create a list of tuples storing the token-lists for each doc as the first␣
↪element and the corresponding label as the second element.
# Do this for each dataset split:
original_train_data_tuples = list(zip(original_dataset_train_tokens,␣
↪original_dataset_train_labels))
original_validation_data_tuples = list(zip(original_dataset_validation_tokens,␣
↪original_dataset_validation_labels))
original_test_data_tuples = list(zip(original_dataset_test_tokens,␣
↪original_dataset_test_labels))
```

```python
# Create featuresets out of the train and test document-tuples by applying the doc_features function defined above
original_train_data_featuresets = [(doc_features(doc, word_features), label) for (doc, label) in original_train_data_tuples]
original_validation_data_featuresets = [(doc_features(doc, word_features), label) for (doc, label) in original_validation_data_tuples]

# Log the results for verification: print the first 10 features of the 1st doc:
# The first index [0] extracts the first featureset for the first training sample
# The second index [0] extracts the first element in the tuple (the dictionary representing
# presence and absence of words, excluding the label) --> Then the key-value pairs in the dict are extracted using 'items()' -->
# then the dict-items are converted into a list and the first 10 key-value pairs for this sample printed out.
print(f"\nFIRST FEATURESET (first 10 features): {list(original_train_data_featuresets[0][0].items())[0:10]}\n")

# Instantiate and train a basic multinomial distribution NB classifier using the NLTK library, as shown in the course lectures.
NBclassifier = nltk.NaiveBayesClassifier.train(original_train_data_featuresets)

# Evaluate on the original validation set and print accuracy score
print(f"BASELINE NB CLASSIFIER ACCURACY: {nltk.classify.accuracy(NBclassifier, original_validation_data_featuresets)}")

"""
    To calculate what the accuracy would be if the model just trivially selected the majority (neutral sentiment) class every time,
    calculate the ratio of the number of majority class samples to the total number of samples in the validation set.
"""
num_majority_class_labels = original_dataset_validation_labels.count(2) # 2= neutral class/majority class, count occurrences in validation set
valset_size = len(original_dataset_validation_labels) # total samples in validation set
print(f"A classifier selecting the majority class every time would achieve a result of {num_majority_class_labels / valset_size}\n")

# Store a list of the predicted labels for each of the samples in the validation split for calculating more advanced metrics
original_dataset_validation_predictions = [] # store predicted labels in here.
"""
```

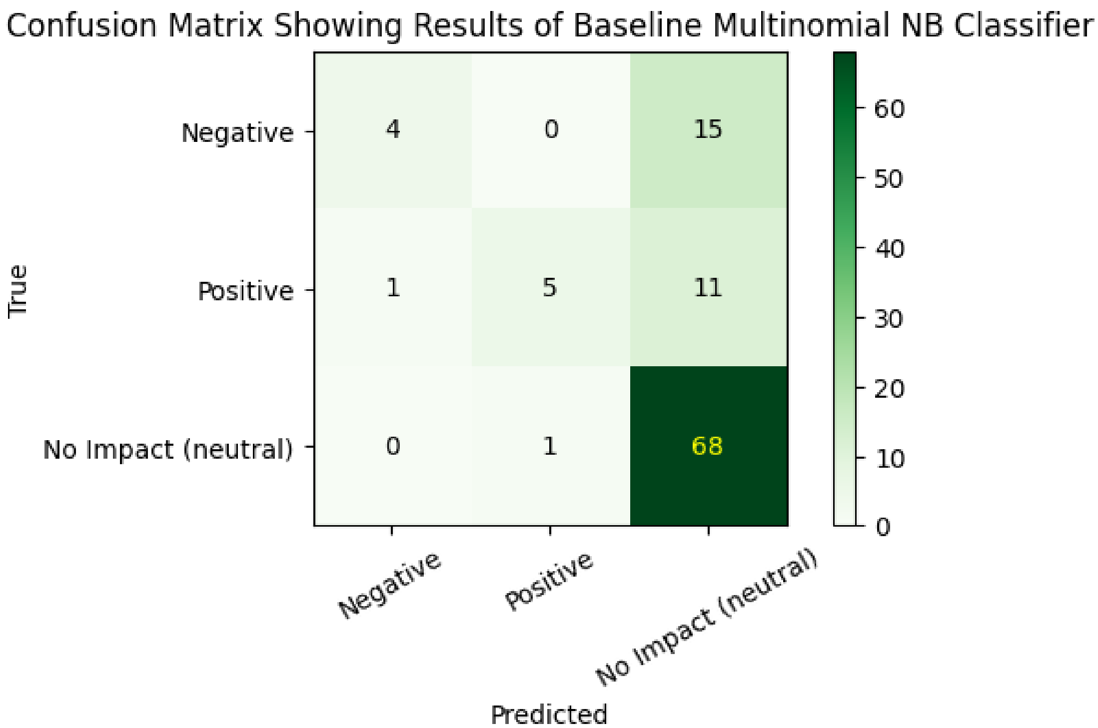
```
    Iterate over the tuples in the validation featureset (reminder: first␣
 ↪element is the
    dict recording the absence/presence of each word, second is the target␣
 ↪label)
"""
for features_dict, label in original_validation_data_featuresets:
    # Apply the NB classifier to get the predicted label for each sample in the␣
 ↪validation set
    predicted_label = NBclassifier.classify(features_dict)
    # Add the predicted label to the predictions list.
    original_dataset_validation_predictions.append(predicted_label)

# Store a set of all the target/label names in ascending order for easier␣
 ↪interpretation of the classification report and confusion matrix.
label_names = ['Negative', 'Positive', 'No Impact (neutral)']

# Print the classification report to view the precision, recall, f1 score for␣
 ↪each class and the macro-averages of each metric
print("CLASSIFICATION REPORT:\n")
print(classification_report(
    original_dataset_validation_labels,
    original_dataset_validation_predictions,
    target_names=label_names)
)

# Define a function that outputs and visualizes a confusion matrix: we will use␣
 ↪a lot of these, so this ensures re-usability of code!
# Code adapted from: https://medium.com/@eceisikpolat/
 ↪plot-and-customize-multiple-confusion-matrices-with-matplotlib-a19ed00ca16c
def generate_and_show_confusion_matrix(
        true_labels, predicted_labels,
        label_names, # informative class names go here
        classifier_description, matrix_color=plt.cm.Greens, # default: green␣
 ↪colour-coded confusion matrix
        above_threshold_text_color="yellow" # color in which to show the text␣
 ↪of counts above the threshold
    ):
    """
        Creates a confusion matrix to show errors made by the classifier on␣
 ↪each class.
        Inputs:
            true_labels = list of true labels
            predicted_labels = list of predicted labels
            label_names = list of label names as strings
            classifier_description = string with description of the classifier/
 ↪methods used
```

```
        matrix_color = color scheme of the matrix from matplotlib
        above_threshold_text_color = the string representing the color to␣
↪print scores above a certain threshold
    Outputs:
        none, just displays the confusion matrix
"""
# Visualize the errors made for each class using a confusion matrix
matrix = confusion_matrix(true_labels, predicted_labels)
# Set matrix size
plt.figure(figsize=(6, 4))
plt.imshow(matrix, interpolation='nearest', cmap=matrix_color) # Use␣
↪nearest-neighbour interpolation to preserve exact values
plt.title(classifier_description)

# Show the color-coding legend
plt.colorbar()
# Add labels for each class to the x- and y-axes: get a numpy array of␣
↪numbers from 0 to nr of classes - 1
ticks = np.arange(len(label_names))
print(ticks)
plt.xticks(ticks, label_names, rotation=30) # Rotate x-axis labels by 30%␣
↪for improved readability
plt.yticks(ticks, label_names)

# Add a threshold value (max value in the matrix divided by 2) after which␣
↪the text-color is inverted from black to yellow (for visibility)
threshold = matrix.max() / 2.
# Iterate of the number of rows in the confusion matrix
for i in range(matrix.shape[0]):
    # Iterate of the number of columns in the confusion matrix
    for j in range(matrix.shape[1]):
        # For each cell in the matrix, convert each value to an integer␣
↪('d') and place the number in the center of the cell
        plt.text(
            j, i, format(matrix[i, j], 'd'),
            ha="center", va="center",
            # If the value/score in this cell is above the threshold, print␣
↪it in the above_threshold_text_color, else use black
            color=above_threshold_text_color if matrix[i, j] > threshold␣
↪else "black"
        )

# Adjust the subplot parameters so that the matrix fits in to the figure␣
↪area
plt.tight_layout()
# Label the axes to show which are true and which are predicted values
```

```
    plt.xlabel("Predicted")
    plt.ylabel("True")
    plt.show()

# Apply the confusion matrix function to visualize the results of the baseline␣
 ↪classifier
generate_and_show_confusion_matrix(
    original_dataset_validation_labels, # true labels
    original_dataset_validation_predictions, # predicted labels
    label_names=label_names,
    classifier_description="Confusion Matrix Showing Results of Baseline␣
 ↪Multinomial NB Classifier"
)
```

```
FIRST 10 SAMPLES FROM TRAINING DATA: ['with pale blue berries. in these peaceful
shades--', 'it flows so long as falls the rain,', 'and that is why, the lonesome
day,', 'when i peruse the conquered fame of heroes, and the victories of mighty
generals, i do not envy the generals,', 'of inward strife for truth and
liberty.', 'the red sword sealed their vows!', 'and very venus of a pipe.', 'who
the man, who, called a brother.', 'and so on. then a worthless gaud or two,',
'to hide the orb of truth--and every throne']
FIRST 10 LABELS FROM TRAINING DATA: [1, 2, 0, 3, 3, 3, 2, 2, 0, 2]


TOTAL VOCABULARY SIZE: 2304

TOP 40 TERMS: [',', 'the', 'and', '.', 'of', 'to', 'a', ';', 'in', 'i', "'s",
'that', 'with', 'his', '!', '--', 'it', 'on', 'he', 'is', 'as', 'but', "'",
'from', 'you', 'all', '?', 'her', 'my', 'not', 'for', '``', 'their', 'so', ':',
'when', 'by', 'thy', 'they', 'we']

FIRST FEATURESET (first 10 features): [('contains(,)', False), ('contains(the)',
False), ('contains(and)', False), ('contains(.)', True), ('contains(of)',
False), ('contains(to)', False), ('contains(a)', False), ('contains(;)', False),
('contains(in)', True), ('contains(i)', False)]

BASELINE NB CLASSIFIER ACCURACY: 0.7333333333333333
A classifier selecting the majority class every time would achieve a result of
0.6571428571428571

CLASSIFICATION REPORT:

                     precision    recall  f1-score   support

           Negative       0.80      0.21      0.33        19
           Positive       0.83      0.29      0.43        17
No Impact (neutral)       0.72      0.99      0.83        69
```

```
    accuracy                           0.73       105
   macro avg       0.79      0.50      0.53       105
weighted avg       0.76      0.73      0.68       105

[0 1 2]
```



2.1.2 Evaluation of Baseline NB Classifier Performance

Accuracy

- The accuracy (ratio of total correct to total predictions) of the baseline Naive Bayes classifier was 73%. If the classifier had trivially predicted the major class every time, the accuracy would have been ((69/105)*100) 66%. This demonstrates that even the baseline classifier with no complex feature engineering for the inputs (yet) has some statistical power. However, as mentioned previously, accuracy can be very misleading in this scenario due to the highly unbalanced nature of the validation set (proportions of 19-17-69 for the three classes).

Precision and Recall

- While the precision for each class was quite high (80%, 83% and 72%), indicating a low rate of false positives, the recall scores for the negative and positive classes were extremely poor.
- This highlights the importance of taking these metrics into account as well as accuracy. While 73% accuracy seems to indicate decent performance, the recall for negative samples was only 21% and 29% for positive samples.

- This also leads to very low F1-scores (the harmonic mean of precision and recall) for both non-neutral classes.
- The confusion matrix clearly highlights that a large number of positive and negative samples are classified incorrectly as neutral, indicating a very low true positive rate for these classes. This might be to do with the fact that the dataset is seriously limited as it contains many more neutral samples than positive and negative samples, thus degrading the performance of the simple Naive Bayes classifier. The section below logs the samples which received the wrong prediction score, to get a more detailed glance at where the classifier is struggling.

[85]:
```
############################################ PERFORMING ERROR ANALYSIS ON␣
↪MISCLASSIFIED SAMPLES #####################################################

# Create some lists storing the misclassified poem lines from the validation␣
↪set and their predicted and true labels
misclassified_samples = []
misclassified_predicted_labels = []
true_labels = []

# Go through each feature set in the validation set, retrieve  the predicted␣
↪label, and if it is wrong, append the misclassified sample
# and the true/predicted labels to the above lists. Use the "enumerate" syntax␣
↪to get the index of the corresponding sample text.
for index, (features_dict, true_label) in␣
↪enumerate(original_validation_data_featuresets):
    # Use the Naive Bayes classifier to classify the validation sample
    predicted_label = NBclassifier.classify(features_dict)
    # If prediction is incorrect --> proceed to append the information about␣
↪this sample
    if predicted_label != true_label:
        # Find the corresponding original sample from the set of texts
        misclassified_samples.append(original_dataset_validation_samples[index])
        misclassified_predicted_labels.append(predicted_label)
        true_labels.append(true_label)

# Print the information about each incorrectly predicted sample
print(f"Total num of wrong predictions: {len(misclassified_samples)}\n")
for i in range(len(misclassified_samples)):
    # Get the true and predicted labels for each misclassified sample as a␣
↪number between 0 and 2
    true_label_as_integer = true_labels[i]
    predicted_label_as_integer = misclassified_predicted_labels[i]
    # Convert the integer label to the actual name of the class
    true_label = label_names[true_label_as_integer]
    predicted_label = label_names[predicted_label_as_integer]
    sample = misclassified_samples[i]
    print(f'Misclassified sample nr {i + 1}: "{sample}"')
```

```
    print(f"The true label was {true_label} but the predicted label was␣
↪{predicted_label}")
   ␣
↪print("********************************************************************************
```

```
Total num of wrong predictions: 28

Misclassified sample nr 1: "shall yet be glad for him, and he shall bless"
The true label was Positive but the predicted label was No Impact (neutral)
********************************************************************************
*********
Misclassified sample nr 2: "taking and giving radiance, and the slopes"
The true label was Positive but the predicted label was No Impact (neutral)
********************************************************************************
*********
Misclassified sample nr 3: "press hard the hostile towers!"
The true label was Negative but the predicted label was No Impact (neutral)
********************************************************************************
*********
Misclassified sample nr 4: "and ever the rocks' disdain;"
The true label was Negative but the predicted label was No Impact (neutral)
********************************************************************************
*********
Misclassified sample nr 5: "let fall on her a rose-leaf rain of dreams,"
The true label was Positive but the predicted label was No Impact (neutral)
********************************************************************************
*********
Misclassified sample nr 6: "alone went the fair-armed gudrun to her flowery
garden-close;"
The true label was Positive but the predicted label was No Impact (neutral)
********************************************************************************
*********
Misclassified sample nr 7: "all passionate-sweet, as are the loving beams"
The true label was Positive but the predicted label was No Impact (neutral)
********************************************************************************
*********
Misclassified sample nr 8: "which 'mongst the wanton gods a foul reproach was
held."
The true label was Negative but the predicted label was No Impact (neutral)
********************************************************************************
*********
Misclassified sample nr 9: "nor can express the love it knew,"
The true label was Negative but the predicted label was No Impact (neutral)
********************************************************************************
*********
Misclassified sample nr 10: "or dying wail!"
```

```
The true label was Negative but the predicted label was No Impact (neutral)
********************************************************************************
*********
Misclassified sample nr 11: "turn'd back the shafts, and mock'd the gates of
death,"
The true label was Negative but the predicted label was No Impact (neutral)
********************************************************************************
*********
Misclassified sample nr 12: "whose voices, hushed, have left our pathway
lonely,"
The true label was Negative but the predicted label was No Impact (neutral)
********************************************************************************
*********
Misclassified sample nr 13: "on us lift up the light"
The true label was Positive but the predicted label was No Impact (neutral)
********************************************************************************
*********
Misclassified sample nr 14: "fix’d on the walls with wonder and surprise,"
The true label was Positive but the predicted label was Negative
********************************************************************************
*********
Misclassified sample nr 15: "i kin eat in peace."
The true label was Positive but the predicted label was No Impact (neutral)
********************************************************************************
*********
Misclassified sample nr 16: "this hot, sick air! and how i covet here"
The true label was Negative but the predicted label was No Impact (neutral)
********************************************************************************
*********
Misclassified sample nr 17: "abominations; and with cursed things"
The true label was Negative but the predicted label was No Impact (neutral)
********************************************************************************
*********
Misclassified sample nr 18: "how the white mountain-tops distinctly shine,"
The true label was Positive but the predicted label was No Impact (neutral)
********************************************************************************
*********
Misclassified sample nr 19: "willis sneered:"
The true label was Negative but the predicted label was No Impact (neutral)
********************************************************************************
*********
Misclassified sample nr 20: "strong tarchon snatch’d and bore away his prize."
The true label was Positive but the predicted label was No Impact (neutral)
********************************************************************************
*********
Misclassified sample nr 21: "and at my door they cower and die."
The true label was Negative but the predicted label was No Impact (neutral)
********************************************************************************
```

```
*********
Misclassified sample nr 22: "weak, timid, homesick, slow to understand"
The true label was Negative but the predicted label was No Impact (neutral)
********************************************************************************
*********
Misclassified sample nr 23: "but no tidings thread the gloom,"
The true label was Negative but the predicted label was No Impact (neutral)
********************************************************************************
*********
Misclassified sample nr 24: "nor looks on that dread place"
The true label was Negative but the predicted label was No Impact (neutral)
********************************************************************************
*********
Misclassified sample nr 25: "from the hushed and silent tomb."
The true label was Negative but the predicted label was No Impact (neutral)
********************************************************************************
*********
Misclassified sample nr 26: "with the freedom of lakes and lands."
The true label was Positive but the predicted label was No Impact (neutral)
********************************************************************************
*********
Misclassified sample nr 27: "shall part, yet link, thy nature's tone and mine."
The true label was No Impact (neutral) but the predicted label was Positive
********************************************************************************
*********
Misclassified sample nr 28: "so generous to me. farewell, friend, since friend"
The true label was Positive but the predicted label was No Impact (neutral)
********************************************************************************
*********
```

This basic classifier fails to correctly detect the polarity of poetry fragments even where the sentiment is clearly strongly positive or negative. Examples containing words with strong positive connotations such as "shall yet be **glad** for him, and he shall **bless**" are predicted to be "neutral", while fragments filled with words indicating strong negative emotion ("abominations; and with cursed things") have also been incorrectly categorized as neutral. As a result, the next section of this project will focus on trying to improve these disappointing scores with more sophisticated feature engineering techniques.

2.1.3 Why Use 750 Top Words? A Visualization Comparing the Impact of Different Vocabulary Sizes on Performance of Baseline NB Classifier

In this section, it will be shown that for the simple baseline classifier used above, using the top 750 words from the vocabulary (which is of size of approximately 2300 words) results in the best accuracy and average f1-scores. The performance of the classifier can change drastically based on the number of word features used, hence the code below and the rationale for using 750 for the basic classifier trained above.

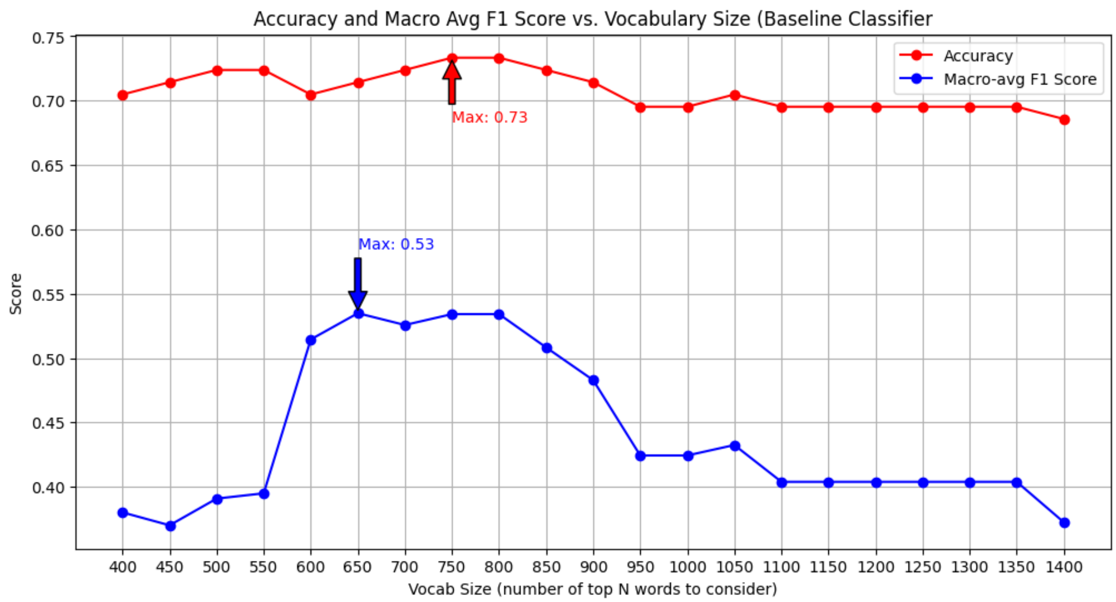
[8]:
```python
def calculate_metrics_for_different_vocab_size_features(
    lowest_num_words_limit, # lower end of range for how many word features to␣
↪use
    highest_num_words_limit, # higher end of range for how many word features␣
↪to use
    all_words, # the frequency distribution of words ordered by most common to␣
↪least common
    train_tuples, # training data tuples of form (sample, label)
    val_tuples,
    step_size=50, # interval size between numbers of words to test
):
    """
        A function that takes in a range of values for the different nr of most␣
↪common words to use as features
        and then calculates the accuracies and average f1-scores for each nr of␣
↪most common words.
        Inputs:
            lowest_num_words_limit: the minimum number of word features to try␣
↪out
            highest_num_words_limit: the maximum number of word features to try␣
↪out (word experiments go up to this number of words - 1)
            all_words: a FreqDist object containing the counts of words in all␣
↪the training tokens
            train_tuples: the set of training tuples containing (features-dict,␣
↪label) for each sample
            val_tuples: the set of validation tuples containing (features-dict,␣
↪label) for each sample
            step_size: the number of steps to take between each number of word␣
↪features to experiment with
        Outputs:
            top_word_counts = a NumPy array of the different word feature␣
↪counts to try out
            accuracies = a list of the accuracy scores achieved on the␣
↪validation set for each trial
            avg_f1_scores = a list of the macro-average F1-scores achieved on␣
↪the validation set for each trial
    """

    # Create NumPy array containing the numbers of word features to experiment␣
↪with
    top_word_counts = np.arange(lowest_num_words_limit,␣
↪highest_num_words_limit, step_size)
    # Create lists storing performance scores for each word features count
    accuracies = [] # store accuracies for each nr of top words used in here
    avg_f1_scores = [] # store macro f1 scores for each nr of top words used in␣
↪here
```

```
    # Iterate over the array of word feature counts to use (i.e. the size of␣
↪the vocabulary subset to use for creating sample features)
    for vocab_size in top_word_counts:
        # Store a list of top N  words to use as features
        word_features = list(all_words)[:vocab_size]
        # Get the training and validation feature sets based on the top N word␣
↪features for this iteration
        train_data_featuresets = [(doc_features(doc, word_features), label) for␣
↪(doc, label) in train_tuples]
        validation_data_featuresets = [(doc_features(doc, word_features),␣
↪label) for (doc, label) in val_tuples]
        # Train the NB classifier and append the validation set accuracy score␣
↪to the above-defined list
        NBclassifier = nltk.NaiveBayesClassifier.train(train_data_featuresets)
        accuracy = nltk.classify.accuracy(NBclassifier,␣
↪validation_data_featuresets)
        accuracies.append(accuracy)
        # Store predicted labels for calculating F1-scores
        validation_predictions= []

        # Iterate over each validation featureset and get the predicted label
        for features_dict, label in validation_data_featuresets:
            predicted_label = NBclassifier.classify(features_dict)
            validation_predictions.append(predicted_label)

        # Retrieve the macro-average F1 score from the scikit-learn␣
↪classification report and store it in avg_f1_scores
        class_report = classification_report(
            original_dataset_validation_labels,
            validation_predictions,
            output_dict=True,  # Return report as a dictionary to easily␣
↪extract F1-score
            # Set the score to 0 if "UndefinedMetricWarning" appears because␣
↪either precision or recall for a class are 0.0
            zero_division=0
        )

        # Access the macro-average F1-score from the confusion matrix
        macro_avg_f1 = class_report['macro avg']['f1-score']
        avg_f1_scores.append(macro_avg_f1)

    return top_word_counts, accuracies, avg_f1_scores
```

```
def plot_word_feature_counts_against_scores(top_word_counts, accuracies,␣
↪avg_f1_scores, title):
    """
        This function plots the accuracies and macro-average F1-scores achieved␣
↪against number of word features used.
        It facilitates determining which number of word features is associated␣
↪with the best performance.
        Inputs:
            top_word_counts = list of integers representing numbers of word␣
↪features to use
            accuracies = list of floats representing accuracy achieved for each␣
↪number of word features
            avg_f1_scores = list of macro-average F1 scores achieved for each␣
↪number of word features
            title = string describing the classifier being evaluated for␣
↪optimal number of features
    """
    # Plot the accuracies and f1-scores for each word features size
    plt.figure(figsize=(12, 6))
    # Plot accuracy in red
    plt.plot(top_word_counts, accuracies, color='red', marker='o', ␣
↪label='Accuracy')
    # Plot average F1 scores (blue)
    plt.plot(top_word_counts, avg_f1_scores, color='blue', marker='o',␣
↪label='Macro-avg F1 Score')

    # Add the labels on each axis and title
    plt.xlabel('Vocab Size (number of top N words to consider)')
    plt.ylabel('Score')
    plt.title(title)
    # Add legend to explain the colours for accuracy and F1-scores
    plt.legend()
    plt.xticks(top_word_counts)

    # Find the index of the first maximum accuracy and F1 score
    max_accuracy_index = np.argmax(accuracies)
    max_f1_score_index = np.argmax(avg_f1_scores)

    # Get the corresponding values for max accuracy and F1 score
    max_accuracy = accuracies[max_accuracy_index]
    max_f1_score = avg_f1_scores[max_f1_score_index]

    # Annotate the point where the first max accuracy occurs to make it easy to␣
↪find the optimal number of word features to use
    plt.annotate(
```

```
                    f"Max: {max_accuracy:.2f}", # the annotation, round␣
↪accuracy to 2 significant figures
                    xy=(top_word_counts[max_accuracy_index], max_accuracy), #␣
↪coordinates of the point to annotate
                    xytext=(top_word_counts[max_accuracy_index], max_accuracy -␣
↪0.05), # coordinates of where to put the text
                    arrowprops=dict(facecolor='red', shrink=0.05), fontsize=10,␣
↪color='red' # configure the appearance of the arrow pointing to score
                )

    # Annotate the point where the first max F1 score occurs (same as above but␣
↪in blue)
    plt.annotate(f"Max: {max_f1_score:.2f}",␣
↪xy=(top_word_counts[max_f1_score_index], max_f1_score),
                 xytext=(top_word_counts[max_f1_score_index], max_f1_score + 0.
↪05),
                 arrowprops=dict(facecolor='blue', shrink=0.05), fontsize=10,␣
↪color='blue')

    # Use a grid background to make it easier to trace which number of word␣
↪features leads to the best scores
    plt.grid(True)
    plt.show()


# Calculate the lists of accuracies and macro-average F1-scores for each number␣
↪of word features used from 400 words to 1401 words
top_word_counts, accuracies, avg_f1_scores = ␣
↪calculate_metrics_for_different_vocab_size_features(
                                                                                                                                        ␣
↪                   400,
                                                                                                                                        ␣
↪                   1401,
                                                                                                                                        ␣
↪                   all_words,
                                                                                                                                        ␣
↪                   original_train_data_tuples,
                                                                                                                                        ␣
↪                   original_validation_data_tuples
                                                                                                                                        ␣
↪                 )

# Use the lists of scores to plot the chart showing the best number of word␣
↪features to use
```

```
plot_word_feature_counts_against_scores(top_word_counts, accuracies,␣
 ↪avg_f1_scores, 'Accuracy and Macro Avg F1 Score vs. Vocabulary Size␣
 ↪(Baseline Classifier')
```



[9]:
```
# Find max accuracy and max F1 scores
max_accuracy = np.array(accuracies).max()
max_f1_score = np.array(avg_f1_scores).max()

# Print the results
print(f"Max accuracy: {max_accuracy}, max f1-average: {max_f1_score}")
```

```
Max accuracy: 0.7333333333333333, max f1-average: 0.534882366643373
```

The chart above demonstrates that for this dataset, the highest accuracy (73%) and highest F1-scores (53% - obtained when word features count is both 650, 750 and 800) are obtained when the top 750 words are taken as features. Hence the selection of this value for calculating the baseline values.

2.2 Text Pre-Processing and Feature-Engineering Experiments using the Original Dataset Training-Validation Split

2.2.1 Feature Engineering Experiment #1 for Statistical Classifier with Original Dataset Split: Negation Handling using _NOT Prefixes Following a Negation Cue

Negation Handling for Sentiment Analysis Negation handling is a valuable technique for improving the performance of sentiment classifiers. As Ilmawan, Muladi and Prasetia show (2024), there are myriad techniques that one can experiment with to signal the presence of negation within text, including both symbolic approaches and data-driven methods. Negation handling or "negation scope detection" is complicated by the fact that negation can be explicit, as signalled by cues such as the word "not", as well as implicit: "it was the first and last time he ever did a good job".

In this first feature-engineering experiment, a simple algorithm will be used to mark all words following a negation cue (e.g. "not" or "never") up until the next punctuation mark as negative using a "_NOT" prefix, as detailed in Jurafsky and Martin (2024). However, one should bear in mind that this simple approach might not work in all cases (i.e. the word "not" does not always indicate inversion of sentiment, for instance in the phrase "not only does he cook but he also cleans"). Another technique that will be experimented with in this project is outlined in this algorithm written by Utkarsh Lal (2022). As the context is left unchanged if a word does not have an antonym in WordNet, this slightly more advanced approach can hopefully reduce the risk of negating the following words when an apparent negation cue does not actually signal negation.

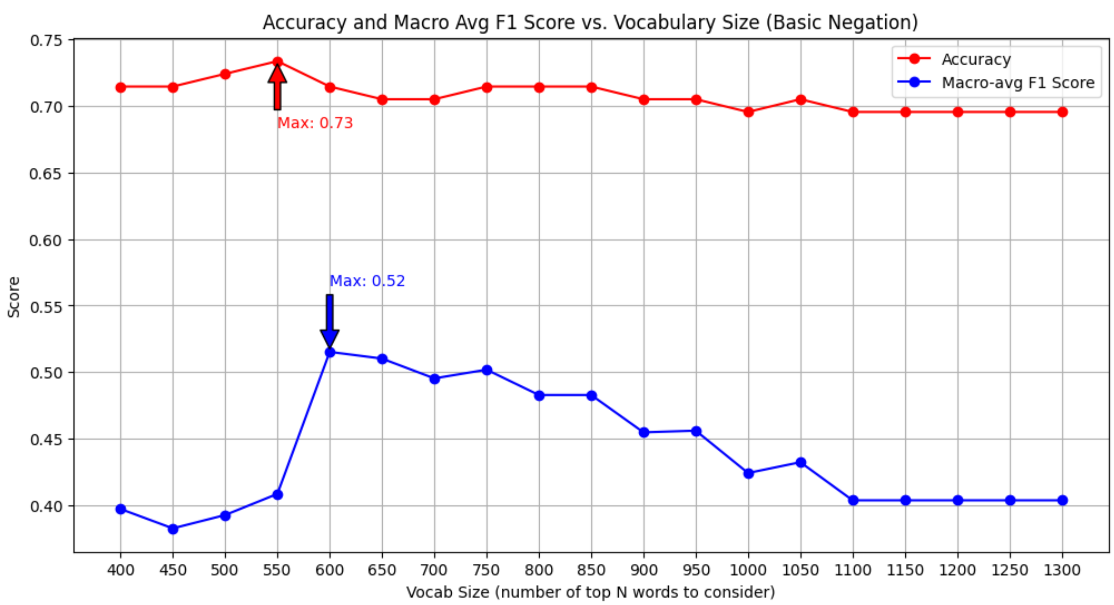
In this section, features will be added indicating that polarity of a fragment of text is negative following a "negation word" (e.g. not, nor, never) using the method detailed in Jurafsky and Martin (2024). For each term following a negation word, a NOT_ will be used to prefix each following word.

[10]:
```
# Define the possible negation patterns using regular expressions
negation_patterns = [
    r"\bnot\b",      # Matches the whole word "not"
    r"not$",         # Matches "not" at the end of the word, e.g. "cannot"
    r"\bno\b",       # Matches the whole word "no"
    r"n't$",         # Matches "n't" at the end of the word, e.g. "shouldn't"
    # Matches the whole words "never", "nor", "none", "nothing", "nowhere" and␣
  ↪"neither"
    r"\bnever\b",
    r"\bnor\b",
    r"\bnone\b",
    r"\bnothing\b",
    r"\bnowhere\b",
    r"\bneither\b"
]

# A list of punctuation markers now including "..." to indicate punctuation␣
  ↪after which negation prefix should be added
punctuation_markers = list(string.punctuation)
punctuation_markers.append("...")

def simple_negation_features(tokens):
    """
        Adds "_NOT" to indicate negation after a token indicating negation such␣
  ↪as "not" or "never" is found.
        Inputs: a list of tokens representing a text sample.
        Outputs: a list of tokens representing the text sample but with "_NOT"␣
  ↪prepended to tokens following a negation.
```

```
    """
    add_negation = False  # Flag to indicate whether to add "_NOT"
    negation_prefix = "NOT_"  # Prefix to prepend to tokens indicating negation
    negated_tokens = []  # List to store tokens with any new added negation␣
↪markers
    # Iterate over each token
    for token in tokens:
        # Check if the token matches any of the negation patterns, thus is a␣
↪negation cue
        if any(re.match(cue, token) for cue in negation_patterns):
            add_negation = True  # Set the flag to add "_NOT" to subsequent␣
↪tokens
            negated_tokens.append(token)  # Add current token as-is to␣
↪negated_tokens
            continue # proceed to the next token

        # If punctuation found, reset flag to stop adding "_NOT" as a prefix
        if token in punctuation_markers:
            add_negation = False

        # Add the "_NOT" prefix to token when add_negation flag is on
        if add_negation:
            negated_tokens.append(negation_prefix + token)  # Add "_NOT" prefix␣
↪to token
        else:
            negated_tokens.append(token)  # Add the original token if flag is␣
↪off
    return negated_tokens
```

[11]:
```
# Apply negation handling to each sample in the train and val sets
basic_negated_train_tokens = [simple_negation_features(tokens) for tokens in␣
↪original_dataset_train_tokens]
basic_negated_validation_tokens = [simple_negation_features(tokens) for tokens␣
↪in original_dataset_validation_tokens]

# Turn the token lists into tuples pairing each token-list with its label
basic_negated_train_data_tuples = list(zip(basic_negated_train_tokens,␣
↪original_dataset_train_labels))
basic_negated_validation_data_tuples =␣
↪list(zip(basic_negated_validation_tokens,␣
↪original_dataset_validation_labels))

# make new vocab list containing _NOT flags
vocabulary_list = flatten_list_of_lists(basic_negated_train_tokens)
# create a Frequency Dist of the words to find the top N words that lead to the␣
↪highest scores
```

```python
all_words = nltk.FreqDist(w for w in vocabulary_list)

# Get the accuracies and f1 scores for different numbers of word features to use to choose the best nr of word features for the negated sets
top_word_counts, accuracies, avg_f1_scores = calculate_metrics_for_different_vocab_size_features(
                        400,
                        1301,
                        all_words,
                        basic_negated_train_data_tuples,
                        basic_negated_validation_data_tuples
                      )
# Plot the results
plot_word_feature_counts_against_scores(top_word_counts, accuracies, avg_f1_scores, 'Accuracy and Macro Avg F1 Score vs. Vocabulary Size (Basic Negation)')
```

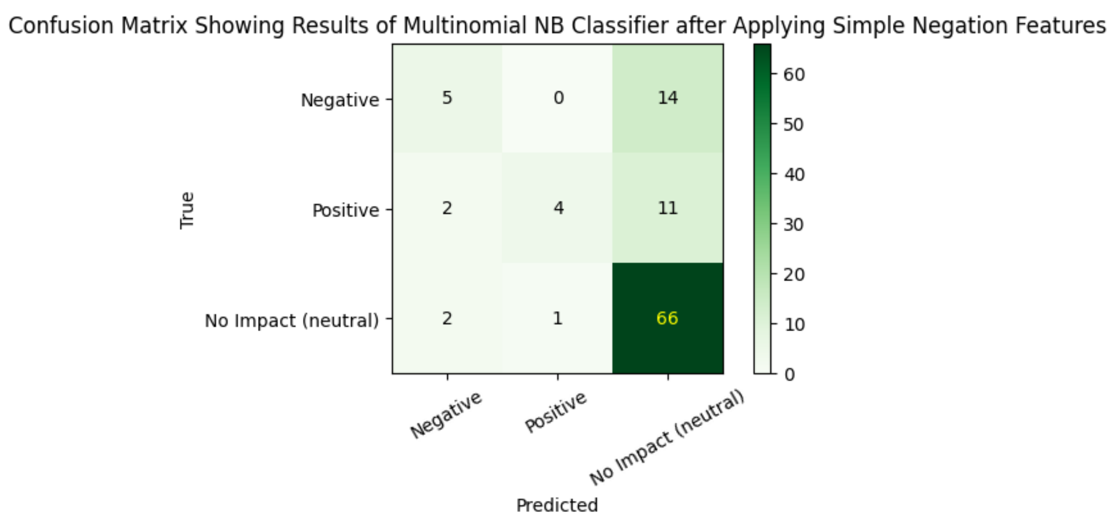


This chart shows that despite adding negation features, the accuracy has not improve. The average f1 score has fallen from 54% to 52%, indicating this is not a success. Morevoer, the optimal number of word features to use has changed from 750 to 550 for accuracy and from 750 to 600 for f1-score. Let's train the classifier again using 600 word features (highest F1-score) now get a closer look at its performance:

[12]:
```
# Train a NB classifier on the negated word sets using the best nr of word␣
↪features (600) based on the findings above
N = 600

# Create list of top N (600) words
word_features = list(all_words)[:N]

# Create feature sets out of the train and validation document-tuples by␣
↪applying doc_features to the negated samples
basic_negated_train_data_featuresets = [(doc_features(doc, word_features),␣
↪label) for (doc, label) in basic_negated_train_data_tuples]
basic_negated_validation_data_featuresets = [(doc_features(doc, word_features),␣
↪label) for (doc, label) in basic_negated_validation_data_tuples]

# Initialize the Multinomial Naive Bayes classifier
NBclassifier = nltk.NaiveBayesClassifier.
↪train(basic_negated_train_data_featuresets)
# Print the accuracy score
print(f"Accuracy Score: {nltk.classify.accuracy(NBclassifier,␣
↪basic_negated_validation_data_featuresets)}")

basic_negated_dataset_validation_predictions = [] # store predicted labels in␣
↪here.

# Iterate over the tuples in the validation featureset to get predicted label␣
↪for each using the classifier
for features_dict, label in basic_negated_validation_data_featuresets:
    predicted_label = NBclassifier.classify(features_dict)
    basic_negated_dataset_validation_predictions.append(predicted_label)

# Print classification report to view the precision, recall, f1 score for each␣
↪class and macro-average
print("CLASSIFICATION REPORT:\n")
print(classification_report(
    original_dataset_validation_labels, # labels are the same for the negated␣
↪featureset as for the original featureset
    basic_negated_dataset_validation_predictions,
    target_names=label_names)
)

# apply confusion matrix function to the new results
generate_and_show_confusion_matrix(
    original_dataset_validation_labels,
    basic_negated_dataset_validation_predictions,
    label_names=label_names,
```

```
    classifier_description="Confusion Matrix Showing Results of Multinomial NB␣
↪Classifier after Applying Simple Negation Features"
)
```

```
Accuracy Score: 0.7142857142857143
CLASSIFICATION REPORT:

                      precision    recall  f1-score   support

            Negative       0.56      0.26      0.36        19
            Positive       0.80      0.24      0.36        17
No Impact (neutral)        0.73      0.96      0.83        69

            accuracy                           0.71       105
           macro avg       0.69      0.48      0.52       105
        weighted avg       0.71      0.71      0.67       105

[0 1 2]
```



Evaluation of Results using NB Classifier with Simple Negation Features The results of this experiment were not very successful. Most of the negative and positive samples are still being classified as "neutral" (as clearly seen in the confusion matrix), and the average F1 score has decreased from 54% to 52%. We will now try a novel rule-based approach detailed by Utkarsh Lal in *Towards Data Science* that involves replacing each word that follows a negation-word with its antonym using WordNet synsets.

2.2.2 Feature Engineering Experiment #2 for Statistical Classifier with Original Dataset Split: Negation Handling using WordNet

The alternate method for negation handling detailed here will be adapted in the following cell. This technique uses WordNet to generate synsets (sets of "synonymous words that express the same concept") for the word imemdiately following a negation cue such as "not" or "never" (Lal, 2022).

After generating the synsets for the word, this algorithm checks whether WordNet contains its antonym (word with the opposite meaning). The word is left as it is if there is no available antonym. For instance, verbs do not have antonyms, so expressions such as "he did not envy" will not be negated, allowing for a more nuanced strategy for negation handling, that leaves certain words following a negation cue as they are if they do not have a clear inverted meaning

This new algorithm selects the antonym with the highest dissimilarity score (calculated as *dissimilarity = (1 — word1.wup_similarity(word2))*) to ensure that the maximum polarity reversal takes place. The negation-cue word preceding the antonym is then removed.

[47]:
```
## CODE TAKEN FROM https://towardsdatascience.com/
↪increasing-accuracy-of-sentiment-classification-using-negation-handling-9ed6dca91f53␣
↪###########

def handle_negation_with_wordnet(tokens, negation_patterns):
    """
        Reverses the meaning of tokens following a negation cue to its antonym␣
↪(if one exists) using WordNet synsets.
        Originally, the algorithm only replaced words following negation cues␣
↪with their antonym if they immediately followed
        a negation cue (e.g. "she was not beautiful" --> "she was ugly").␣
↪However, it was unable to reverse the meaning of sentiment-carrying words
        that appeared in a fragment of text where there was a determiner (e.g.␣
↪"the", "a") between the "negation cue" word and the
        word that should be inverted (e.g. "it was not A nice day" --> "it was␣
↪an ugly day").
        Input: Tokenized sentence (list of words)
        Output: Tokenized sentence with negation handled (list of words)
        Adaptations from original algorithm: I added more complex rules to␣
↪ensure that words following a negation pattern + determiner
        and following a negation pattern + adverb (e.g. "very", "extremely")␣
↪were also negated.
    """
    # Use a temp variable to store dissimilarity score to find the antonym with␣
↪the maximum polarity difference to the current word
    temp = int(0)

    # Get POS tags for the tokens to handle cases where a negation cue is␣
↪followed by a determiner or adverb.
    pos_tags = pos_tag(tokens)
```

```
    # Examine each token
    for i in range(1, len(tokens)):  # Start from 1 to avoid errors when␣
↪looking at prev token
        # Adapted the original code published by Lal (2022) to look if the␣
↪token matches any of the pre-defined regular expressions indicating negation
        if any(re.search(pattern, tokens[i-1]) for pattern in␣
↪negation_patterns): # check if prev token is a negation cue
            ######################################## MY ADAPTATION HERE␣
↪###############################################################################
            # If the current token is a determiner and next token is an adverb␣
↪(e.g. very, most, quite) then target the word following the adverb.
            if pos_tags[i][1] in ['DT'] and i + 2 < len(tokens) and pos_tags[i␣
↪+ 1][1] in ['RB']: # match pattern "NEG - DT - RB - TARGET WORD"
                index_of_word_to_replace = i + 2 # index of target word
            # If the current token is a determiner and next word is not an␣
↪adverb then target the word immediately after the determiner
            elif pos_tags[i][1] in ['DT'] and i + 1 < len(tokens): # match␣
↪pattern "NEG - DET - TARGET WORD"
                index_of_word_to_replace = i + 1
            # If the current token is not a determiner, target the current␣
↪token immediately following the negation pattern
            else:
                index_of_word_to_replace = i


          ␣
↪############################################################################################

            # Store antonyms for the target word.
            antonyms = []

            # Iterate over the synsets for the current token/word.
            for syn in wordnet.synsets(tokens[index_of_word_to_replace]):
                # Iterate over the lemmas/WordNet names for word senses in this␣
↪synset
                for l in syn.lemmas():
                    # If lemma/word sense has an antonym, append its name/
↪antonym word to the list of antonyms
                    if l.antonyms():
                        antonyms.append(l.antonyms()[0].name())

            # If antonyms exist for the target word, find the one with the␣
↪maximum dissimilarity to the current token/word
            max_dissimilarity = 0 # keep track of the max dissimilarity here
            antonym_max = tokens[index_of_word_to_replace]  # use current word␣
↪if no antonyms
```

```python
            # Only find max polarity antonym if antonyms exist
            if antonyms:
                w1 = wordnet.synsets(tokens[index_of_word_to_replace])[0]  # First synset of the target word
                # Iterate over possible antonyms and find their synsets
                for ant in antonyms:
                        # Find synsets/word meaning sets for current antonym
                        syns_ant = wordnet.synsets(ant)
                        if syns_ant:
                            # Get first meaning of antonym
                            w2 = syns_ant[0]
                            # Calculate WordNet similarity score between the target token and this antonym.
                            similarity = w1.wup_similarity(w2)
                            if similarity is not None:
                                # Get dissimilarity by subtracting similarity score from 1.
                                dissimilarity = 1 - similarity
                                # If exceeds current max, update the max_dissimilarity variable and the maximum antonym.
                                if dissimilarity > max_dissimilarity:
                                    max_dissimilarity = dissimilarity
                                    antonym_max = ant

            # Replace current token with maximum dissimilarity antonym if found.
            tokens[index_of_word_to_replace] = antonym_max
            # Remove the negation cue
            tokens[i-1] = ''

            # Adjust the indefinite article determiner "a" or "an" based on the first letter of the antonym.
            if tokens[index_of_word_to_replace-1] == "a" and tokens[index_of_word_to_replace][0] in 'aeiou':
                # If next word begins with vowel, change determiner to "an".
                tokens[index_of_word_to_replace-1] = "an"
            elif tokens[index_of_word_to_replace-1] == "an" and tokens[index_of_word_to_replace][0] not in 'aeiou':
                # If next word does not begin with a vowel, change determiner to "a".
                tokens[index_of_word_to_replace-1] = "a"

    # Remove any '' markers used in the algorithm for deleting negation words if an antonym was found.
    while '' in tokens:
        tokens.remove('')
```

```
    return tokens

########################################################################################
```

[48]:

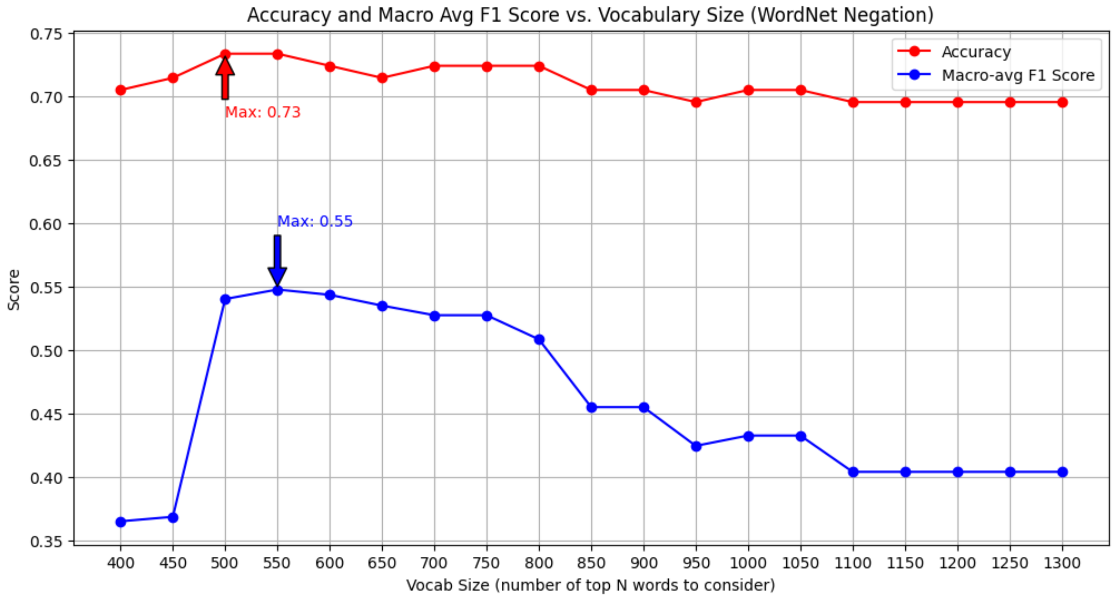
```
## Check this works using basic examples
sentence1 = word_tokenize("She was not nice") # check simple negation from␣
↪positive word to negative word
print(handle_negation_with_wordnet(sentence1, negation_patterns))

sentence2 = word_tokenize("It was not a pleasant day") # check inverting␣
↪meaning of a word following a determiner
print(handle_negation_with_wordnet(sentence2, negation_patterns))

sentence3 = word_tokenize("It was not boring") # check inverting a negative␣
↪word to a positive word following "not"
print(handle_negation_with_wordnet(sentence3, negation_patterns))

sentence4 = word_tokenize("She was never beautiful") # check inverting a␣
↪postive word after "never"
print(handle_negation_with_wordnet(sentence4, negation_patterns))

sentence5 = word_tokenize("It was not a very pleasant day") # check inverting a␣
↪word that follows a determiner and an adverb
print(handle_negation_with_wordnet(sentence5, negation_patterns))

sentence6 = word_tokenize("She cannot swim") # check inverting a verb (swim -->␣
↪sink)
print(handle_negation_with_wordnet(sentence6, negation_patterns))

sentence7 = word_tokenize("She didn't hope") # check inverting a word following␣
↪n't contraction
print(handle_negation_with_wordnet(sentence7, negation_patterns))
```

```
['She', 'was', 'nasty']
['It', 'was', 'an', 'unpleasant', 'day']
['It', 'was', 'interest']
['She', 'was', 'ugly']
['It', 'was', 'a', 'very', 'unpleasant', 'day']
['She', 'can', 'sink']
['She', 'did', 'despair']
```

[51]:

```
## RUN THE EXPERIMENT AGAIN USING THE NEW NEGATION FUNCTION

# Apply new WordNet negation function to all the tokens
wordnet_negated_train_tokens = [handle_negation_with_wordnet(tokens,␣
↪negation_patterns) for tokens in original_dataset_train_tokens]
```

```python
wordnet_negated_validation_tokens = [handle_negation_with_wordnet(tokens,␣
↪negation_patterns) for tokens in original_dataset_validation_tokens]

# Conver the tokens to tuples of (features, label)
wordnet_negated_train_data_tuples = list(zip(wordnet_negated_train_tokens,␣
↪original_dataset_train_labels))
wordnet_negated_validation_data_tuples =␣
↪list(zip(wordnet_negated_validation_tokens,␣
↪original_dataset_validation_labels))

# Create a new vocab list containing the WordNet antonyms
vocabulary_list = flatten_list_of_lists(wordnet_negated_train_tokens)

# Create a frequency distribution based on the new vocbulary
all_words = nltk.FreqDist(w for w in vocabulary_list)

# Get the accuracies and macro-average F1 scores for different numbers of word␣
↪features to choose the best number for the WordNet negated toekens
top_word_counts, accuracies, avg_f1_scores = ␣
↪calculate_metrics_for_different_vocab_size_features(
                                                                               ␣
↪                     400, # lowest number of word features
                                                                               ␣
↪                     1301, # highest number of word features
                                                                               ␣
↪                     all_words, # Frequency distribution for top words
                                                                               ␣
↪                     wordnet_negated_train_data_tuples,
                                                                               ␣
↪                     wordnet_negated_validation_data_tuples
                                                                               ␣
↪                   )
# Plot the results to find optimal number of word features
plot_word_feature_counts_against_scores(
    top_word_counts, accuracies,
    avg_f1_scores,
    'Accuracy and Macro Avg F1 Score vs. Vocabulary Size (WordNet Negation)'
)
```

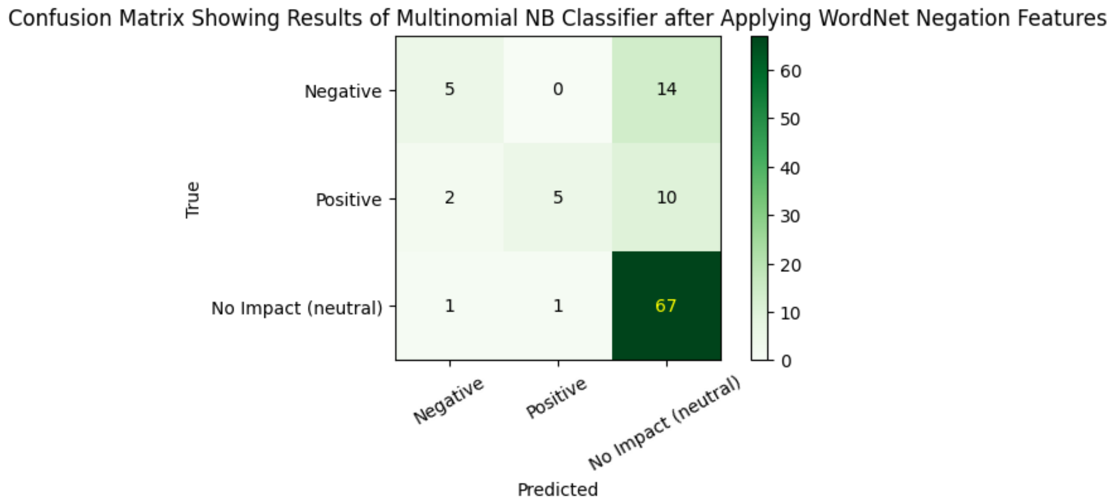


[52]:

```
# Train a new NB classifier on the negated word sets using the best number of␣
↪word features (550 --> best F1-score) using WordNet negation handling
N = 550
word_features = list(all_words)[:N]

# Create feature sets out of the train and validation document-tuples by␣
↪applying the function defined above
wordnet_negated_train_data_featuresets = [(doc_features(doc, word_features),␣
↪label) for (doc, label) in wordnet_negated_train_data_tuples]
wordnet_negated_validation_data_featuresets = [(doc_features(doc,␣
↪word_features), label) for (doc, label) in␣
↪wordnet_negated_validation_data_tuples]

# Instantiate classifier
NBclassifier = nltk.NaiveBayesClassifier.
↪train(wordnet_negated_train_data_featuresets)

# Print the accuracy score
print(f"Accuracy Score: {nltk.classify.accuracy(NBclassifier,␣
↪wordnet_negated_validation_data_featuresets)}")

# Store predicted labels
wordnet_negated_dataset_validation_predictions = []

for features_dict, label in wordnet_negated_validation_data_featuresets:
    # Get predicted label using NB classifier
    predicted_label = NBclassifier.classify(features_dict)
```

```python
    wordnet_negated_dataset_validation_predictions.append(predicted_label)

# View detailed metric scores
print("CLASSIFICATION REPORT:\n")
print(classification_report(
    original_dataset_validation_labels, # labels are the same for the negated␣
 ↪featureset as for the original featureset
    wordnet_negated_dataset_validation_predictions,
    target_names=label_names)
)

# Apply the confusion matrix function to the new results
generate_and_show_confusion_matrix(
    original_dataset_validation_labels,
    wordnet_negated_dataset_validation_predictions,
    label_names=label_names,
    classifier_description="Confusion Matrix Showing Results of Multinomial NB␣
 ↪Classifier after Applying WordNet Negation Features"
)
```

```
Accuracy Score: 0.7333333333333333
CLASSIFICATION REPORT:

                     precision    recall  f1-score   support

           Negative       0.62      0.26      0.37        19
           Positive       0.83      0.29      0.43        17
No Impact (neutral)       0.74      0.97      0.84        69

           accuracy                           0.73       105
          macro avg       0.73      0.51      0.55       105
       weighted avg       0.73      0.73      0.69       105

[0 1 2]
```



Unfortunately, using a much more complex negation handler with WordNet, containing rule-based mechanisms and regular expressions, did not substantially improve the performance of the classifier compared to the baseline. Accuracy was still 73% with macro-average F1-score increasing only by 1% (from 54% to 55%) from the baseline.

Therefore, this indicates that it might not be possible to mitigate the limitations of this dataset (few instances of certain classes, the small size) by using more complex negation handling. Nonetheless, this classifier still performed better than the one using the basic negation handling to generate features, as that basic negation handler reduced the accuracy from the baseline of 73% to 71%. As such, this negation handling technique will be the the one used to conduct the next experiments to try to improve upon these low scores.

2.2.3 Feature Engineering Experiment #3 for Statistical Classifier with Original Dataset Split: WordNet Negation Handling with TF-IDF instead of Binary Vectorization Dicts

In the next experiment, the WordNet negation handling technique will be combined with converting each sample to TF-IDF vectors. This measures the frequency of each term/token in the sample but gives more weight to words that appear more rarely in the corpus (total set of samples).

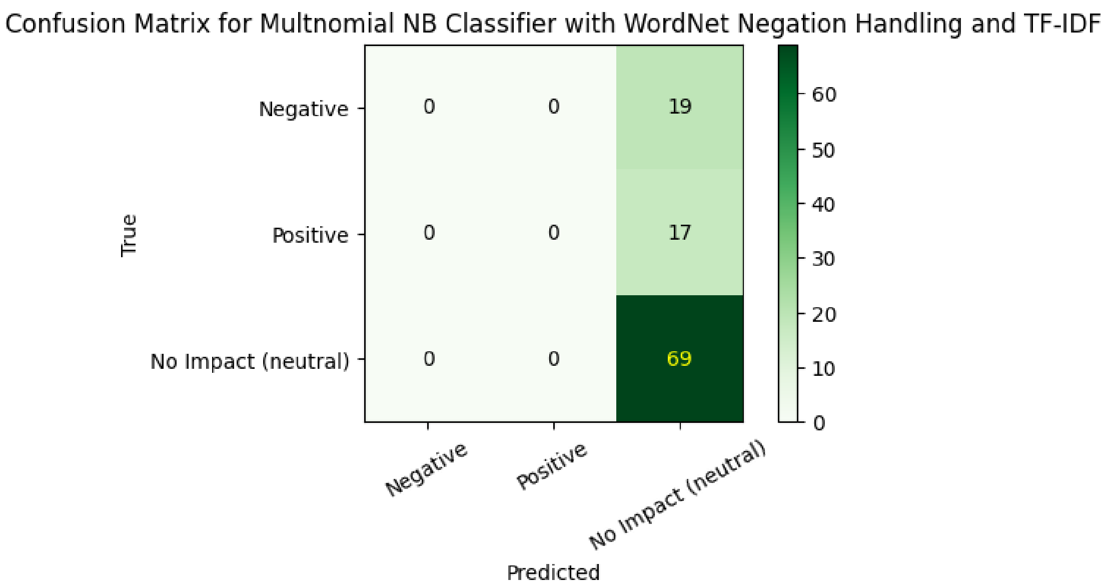
Although, according the Jurafsky and Martin (2024), using feature sets containing the presence or absence of each word type is frequently preferable to using frequency counts in the context of sentiment analysis, this next trial will verify whether this is indeed the case for this particular dataset. The main aim here is to experiment with different featuresets to try to improve the baseline score of 53% F1-measure.

[334]:
```
# Convert token-lists for each sample back into strings to as this is what the
  TF-IDF vectorizer requires as inputs
wordnet_negated_train_texts = [' '.join(tokens) for tokens in
 wordnet_negated_train_tokens]
```

```python
wordnet_negated_validation_texts = [' '.join(tokens) for tokens in␣
↪wordnet_negated_validation_tokens]

# Instantiate and train the TF-IDF Vectorizer on the WordNet negated training␣
↪texts
tfidf_vectorizer = TfidfVectorizer()
tfidf_vectorizer.fit(wordnet_negated_train_texts)

# Transform the training and validation texts/strings into TF-IDF featuresets
wordnet_negated_train_data_tf_idf_featuresets = tfidf_vectorizer.
↪transform(wordnet_negated_train_texts)
wordnet_negated_validation_data_tf_idf_featuresets = tfidf_vectorizer.
↪transform(wordnet_negated_validation_texts)

# Instantiate scikit-learn Multinomial Naive-Bayes classifier to handle TF-IDF␣
↪feature vectors
NBclassifier = MultinomialNB()
# Train the classifier on the training set
NBclassifier.fit(wordnet_negated_train_data_tf_idf_featuresets,␣
↪original_dataset_train_labels)

# Store predicted labels for validation set
validation_predictions = NBclassifier.
↪predict(wordnet_negated_validation_data_tf_idf_featuresets)

# Print accuracy score
accuracy = accuracy_score(original_dataset_validation_labels,␣
↪validation_predictions)
print(f"Accuracy: {accuracy}")

# Print classification report
print("Classification Report:")
print(classification_report(original_dataset_validation_labels,␣
↪validation_predictions, target_names=label_names, zero_division=0))

# Generate and display the confusion matrix using true and predicted labels
generate_and_show_confusion_matrix(
    original_dataset_validation_labels,
    validation_predictions,
    label_names,
    "Confusion Matrix for Multnomial NB Classifier with WordNet Negation␣
↪Handling and TF-IDF"
)
```

```
Accuracy: 0.6571428571428571
Classification Report:
                      precision    recall  f1-score   support
```

```
          Negative       0.00      0.00      0.00        19
          Positive       0.00      0.00      0.00        17
No Impact (neutral)      0.66      1.00      0.79        69

          accuracy                           0.66       105
         macro avg       0.22      0.33      0.26       105
      weighted avg       0.43      0.66      0.52       105

[0 1 2]
```



These results show that, indeed, TF-IDF leads to much worse results than using a simple dictionary with "True" and "False" values representing the presence or absence of each word feature. The accuracy has decreased from the baseline score of 73% to 65% - lower than simply predicting the "neutral" class every time (66%). None of the negative or positive samples have been correctly classified looking at the confusion matrix above.

2.2.4 Feature Engineering Experiment #4 for Statistical Classifier with Original Dataset Split: Negation Handling using WordNet Negation Handling and Removing Stopwords

In this section, we will continue with the attempts to improve the accuracy, recall, precision and F1 scores on the validation dataset by removing stopwords after negation has been handled. However, as Da notes, stopwords are often what provide the most informative indicators of difference in literary and poetic texts. We will see if this is indeed the case for this particular context.

[59]:
```
# Get list of NLTK English stopwords
english_stopwords = stopwords.words('english')
```

```
print(f"Length of stopwords set before filtering: {len(english_stopwords)}")

def filter_stopwords(stopwords, patterns):
    """
        Filters a set of stopwords by removing words which match certain␣
↪regular expressions.
        Used to filter out stopwords indicating negation (e.g. "can't").
        Inputs:
            stopwords = list of English stopwords
            patterns = negation patterns that shouldn't be replaced
        Output:
            non_negated_stopwords = set of English stopwords with negation␣
↪words removed
    """
    non_negated_stopwords = set(
        [stopword for stopword in stopwords
        if not any(re.search(pattern, stopword, re.IGNORECASE) for pattern in␣
↪patterns)
        ])
    return non_negated_stopwords

non_negated_english_stopwords = filter_stopwords(english_stopwords,␣
↪negation_patterns)
print(f"Length of stopwords set after filtering:␣
↪{len(non_negated_english_stopwords)}")

def remove_stopwords_from_tokens(tokens, stopwords):
    """
        Filters stopwords from a list of tokens
        Input:
            tokens = list of tokens representing a sample
            stopwords = list of stopwords
    """
    new_tokens = [token for token in tokens if token not in stopwords]
    return new_tokens
```

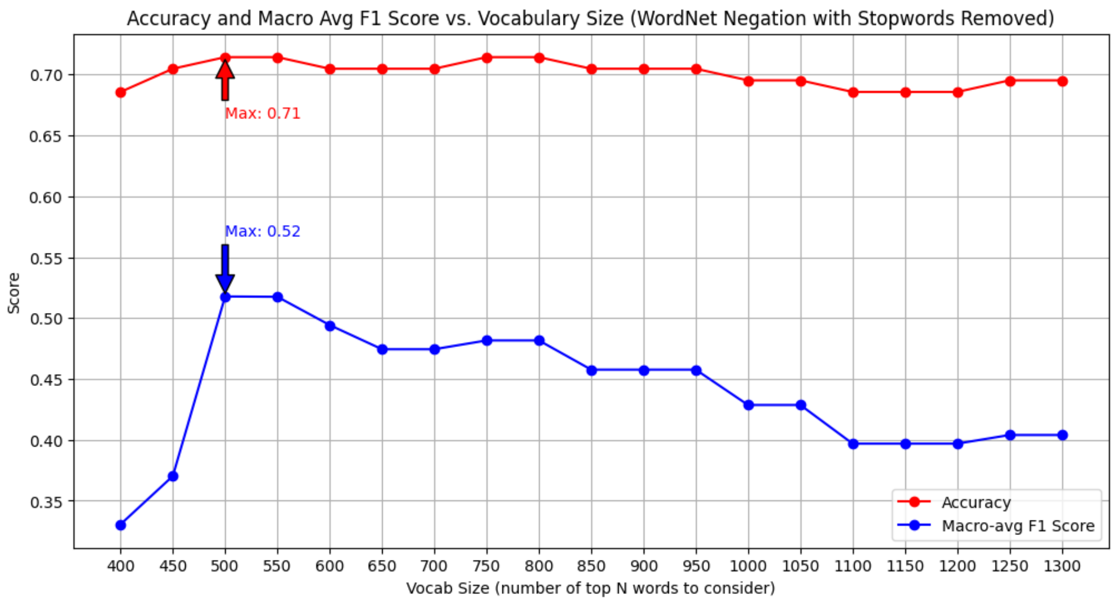
```
Length of stopwords set before filtering: 179
Length of stopwords set after filtering: 158
```

[409]:
```
# Run a trial using WordNet negated samples but with removal of stopwords

# Remove stopwords from WordNet samples
wordnet_negated_train_tokens_no_stopwords = [
    remove_stopwords_from_tokens(tokens, non_negated_english_stopwords)
    for tokens in wordnet_negated_train_tokens
]
```

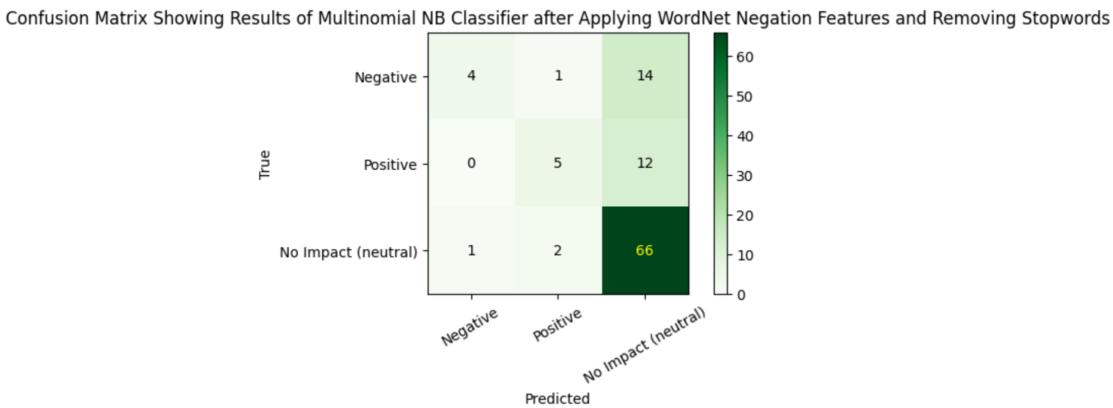
```
wordnet_negated_validation_tokens_no_stopwords = [
    remove_stopwords_from_tokens(tokens, non_negated_english_stopwords)
    for tokens in wordnet_negated_validation_tokens
]

wordnet_negated_train_data_tuples_no_stopwords =␣
 ↪list(zip(wordnet_negated_train_tokens_no_stopwords,␣
 ↪original_dataset_train_labels))
wordnet_negated_validation_data_tuples_no_stopwords =␣
 ↪list(zip(wordnet_negated_validation_tokens_no_stopwords,␣
 ↪original_dataset_validation_labels))

# New vocabulary list without stopwords
vocabulary_list =␣
 ↪flatten_list_of_lists(wordnet_negated_train_tokens_no_stopwords)
# FreqDist without all the stopwords
all_words = nltk.FreqDist(w for w in vocabulary_list)

# Get the accuracies and macro-average F1-scores for different numbers of word␣
 ↪features
top_word_counts, accuracies, avg_f1_scores = ␣
 ↪calculate_metrics_for_different_vocab_size_features(
                                                                              ␣
 ↪                    400,
                                                                              ␣
 ↪                    1301,
                                                                              ␣
 ↪                    all_words,
                                                                              ␣
 ↪                    wordnet_negated_train_data_tuples_no_stopwords,
                                                                              ␣
 ↪                    wordnet_negated_validation_data_tuples_no_stopwords
                                                                              ␣
 ↪                  )
# Plot the results
plot_word_feature_counts_against_scores(
                                         top_word_counts,
                                         accuracies,
                                         avg_f1_scores,
                                        'Accuracy and Macro Avg F1 Score vs.␣
 ↪Vocabulary Size (WordNet Negation with Stopwords Removed)'
                                      )
```

```
# Train a new NB classifier on the samples without stpowords using the best␣
↪number of word features (500)
N = 500
word_features = list(all_words)[:N]

# Create feature sets out of the train and validation document-tuples by␣
↪applying the doc_features func
wordnet_negated_train_data_no_stopwords_featuresets = [
                                            (doc_features(doc, word_features),␣
↪label)
                                            for (doc, label) in␣
↪wordnet_negated_train_data_tuples_no_stopwords
                                         ]

wordnet_negated_validation_data_no_stopwords_featuresets = [
                                                (doc_features(doc,␣
↪word_features), label)
                                                for (doc, label) in␣
↪wordnet_negated_validation_data_tuples_no_stopwords
                                              ]

# Train classifier
NBclassifier = nltk.NaiveBayesClassifier.
↪train(wordnet_negated_train_data_no_stopwords_featuresets)

# Print the accuracy score
print(f"Accuracy Score: {nltk.classify.accuracy(NBclassifier,␣
↪wordnet_negated_validation_data_no_stopwords_featuresets)}")

# Store predicted labels
wordnet_negated_dataset_no_stopwords_validation_predictions = [] # store␣
↪predicted labels in here.
# Iterate over samples
for features_dict, label in␣
↪wordnet_negated_validation_data_no_stopwords_featuresets:
    predicted_label = NBclassifier.classify(features_dict)
    wordnet_negated_dataset_no_stopwords_validation_predictions.
↪append(predicted_label)

# Print classification report to view the metrics
print("CLASSIFICATION REPORT:\n")
print(classification_report(
    original_dataset_validation_labels, # labels are the same for the negated␣
↪featureset as for the original featureset
    wordnet_negated_dataset_no_stopwords_validation_predictions,
    target_names=label_names))
```

```python
# Apply confusion matrix function to the new results
generate_and_show_confusion_matrix(
    original_dataset_validation_labels,
    wordnet_negated_dataset_no_stopwords_validation_predictions,
    label_names=label_names,
    classifier_description = "Confusion Matrix Showing Results of Multinomial␣
 ↪NB Classifier after Applying WordNet Negation Features and Removing␣
 ↪Stopwords"
)
```




```
Accuracy Score: 0.7142857142857143
CLASSIFICATION REPORT:

                      precision    recall  f1-score   support

            Negative       0.80      0.21      0.33        19
            Positive       0.62      0.29      0.40        17
No Impact (neutral)       0.72      0.96      0.82        69

            accuracy                           0.71       105
           macro avg       0.71      0.49      0.52       105
        weighted avg       0.72      0.71      0.66       105

[0 1 2]
```



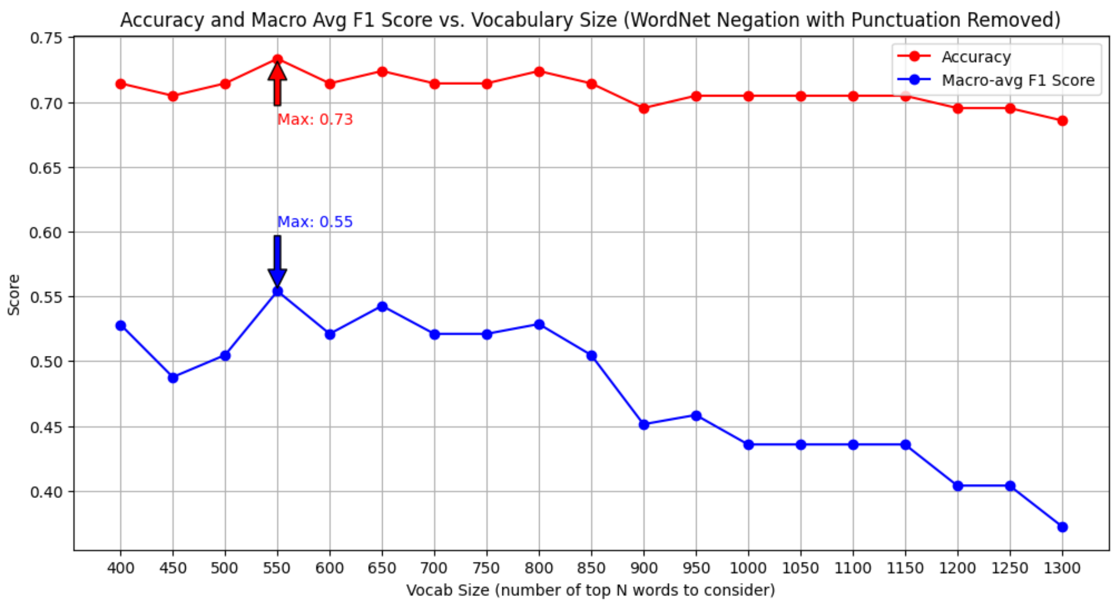
As predicted, removing stopwords has reduced the accuracy from 73% (baseline) to 71%, as well as F1-score from 53% to 52%. Therefore, removing the stopwords did not lead to an improvement in the classifier's performance.

2.2.5 Feature Engineering Experiment #5 for Statistical Classifier with Original Dataset Split: Negation Handling using WordNet and Removing Punctuation

[60]:
```
def remove_punctuation(tokens):
    """
        Removes punctuation from a list of tokens.
        Inputs:
            tokens = a list of tokens
        Outputs:
            new_tokens = the same tokens with punctuation removed
    """
    # Create a translation table mapping every punctuation mark from string.
 ↪punctuation to an empty string for each token
    new_tokens = [token.translate(str.maketrans('', '', string.punctuation))␣
 ↪for token in tokens]
    # Remove empty string tokens from token list
    new_tokens = [token for token in new_tokens if token != '']
    return new_tokens
```

[30]:
```
# Run a new trial with WordNet-negated features but remove punctuation.

# Apply the punctuation remover function to the WordNet negation handled tokens
wordnet_negated_train_tokens_no_punctuation = [
    remove_punctuation(tokens)
    for tokens in wordnet_negated_train_tokens
]
wordnet_negated_validation_tokens_no_punctuation = [
    remove_punctuation(tokens)
```

```
    for tokens in wordnet_negated_validation_tokens
]

# Create feature, label tuples
wordnet_negated_train_data_tuples_no_punctuation =␣
 ↪list(zip(wordnet_negated_train_tokens_no_punctuation,␣
 ↪original_dataset_train_labels))
wordnet_negated_validation_data_tuples_no_punctuation =␣
 ↪list(zip(wordnet_negated_validation_tokens_no_punctuation,␣
 ↪original_dataset_validation_labels))

# Create new vocabulary without punctuation
vocabulary_list =␣
 ↪flatten_list_of_lists(wordnet_negated_train_tokens_no_punctuation)
# Create new Frequency Distribution without punctuation
all_words = nltk.FreqDist(w for w in vocabulary_list)

# Get metrics for each number of word features to select best number
top_word_counts, accuracies, avg_f1_scores = ␣
 ↪calculate_metrics_for_different_vocab_size_features(
                                                                             ␣
 ↪                    400,
                                                                             ␣
 ↪                    1301,
                                                                             ␣
 ↪                    all_words,
                                                                             ␣
 ↪                    wordnet_negated_train_data_tuples_no_punctuation,
                                                                             ␣
 ↪                    wordnet_negated_validation_data_tuples_no_punctuation
                                                                             ␣
 ↪                  )
# Plot the results
plot_word_feature_counts_against_scores(
                                            top_word_counts,
                                            accuracies,
                                            avg_f1_scores,
                                            'Accuracy and Macro Avg F1 Score vs.
 ↪ Vocabulary Size (WordNet Negation with Punctuation Removed)'
                                       )
```



Removing the punctuation leads to the same accuracy (73%) and macro-average F1 (55%) scores as achieved just using the WordNet negation function without other feature extraction techniques. Therefore, removing punctuation did not increase the performance on the validation set.

2.2.6 Feature Engineering Experiment #6 for Statistical Classifier with Original Dataset Split: Negation Handling using WordNet and Lemmatization

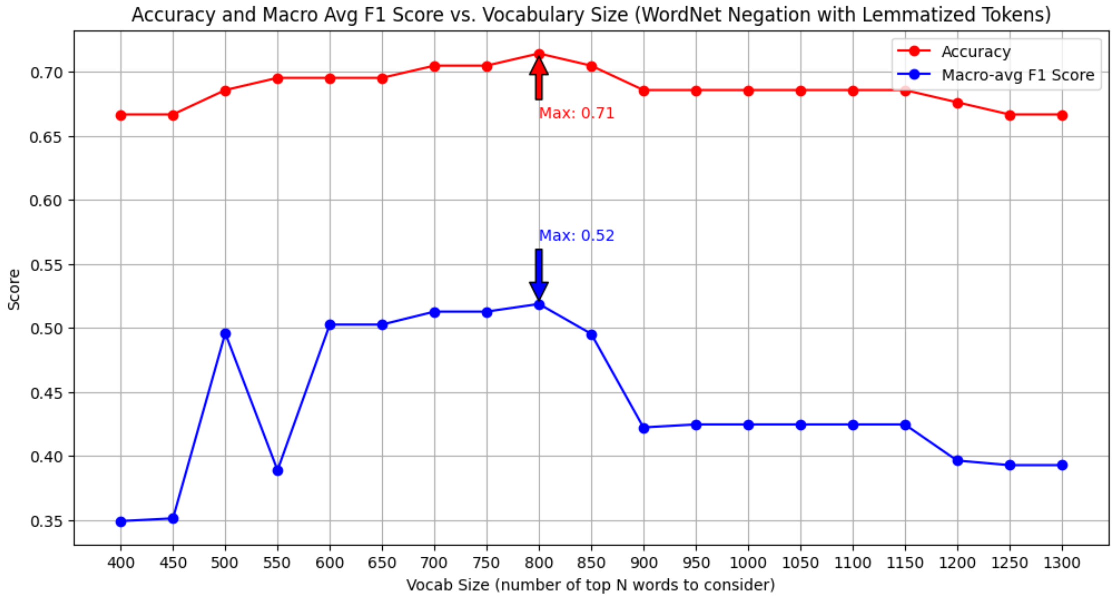
Lemmatization involves converting tokens into their dictionary form, standardizing the vocabulary by representing inflected words with the same base root with the same token. For example, "runs" will be contracted to "run". Sometimes, this can reduce token variability in the number of tokens and can thus assist the classifier in learning new patterns by making it clearer which words are similar. Therefore, we will attempt this experiment in the below code cell. Lemmatization retains the meaning of a word compared to simple stemming, as it outputs genuine words rather than truncated stems, and uses part-of-speech tags to take more context into account (Saumyab271, 2024). Reducing similar words to one root form might help the classifier learn more patterns related to the essential meaning of words rather than the details of there inflections, hence the trial below.

[53]:
```
############################### Attribution:  Code adapted from https://medium.
↪com/@yashj302/lemmatization-f134b3089429 ###############################

def get_wordnet_pos(token):
    """
        Maps a token's part-of-speech tag to the format of a tag that the␣
↪WordNet lemmatizer accepts.
        Input: a string/token
        Output: the WordNet part-of-speech tag for the token
    """
```

```
    # Get the nltk pos_tag for the token
    tag = nltk.pos_tag([token])[0][1]
    # Map the NLTK pos_tag to equivalent WordNet tags
    if tag.startswith('J'):
        return wordnet.ADJ
    elif tag.startswith('N'):
        return wordnet.NOUN
    elif tag.startswith('V'):
        return wordnet.VERB
    elif tag.startswith('R'):
        return wordnet.ADV
    else:
        # Return noun tag as default if no match is found
        return wordnet.NOUN

def lemmatize_tokens(lemmatizer, tokens):
    """
        Lemmatizes a list of tokens.
        Inputs:
            instance of lemmatizer class
            tokens = a list of tokens
        Outputs:
            lemmatized_tokens = a list of lemmatized tokens
    """
    lemmatized_tokens = [lemmatizer.lemmatize(token, get_wordnet_pos(token))␣
 ↪for token in tokens]
    return lemmatized_tokens

#############################################################################################
 ↪
```

[54]:
```
## Run the trial again on WordNet handled tokens but lemmatize them

# Instantiate lemmatizer
lemmatizer = WordNetLemmatizer()

# Lemmatize each token list
lemmatized_wordnet_negated_train_tokens = [
    lemmatize_tokens(lemmatizer, tokens)
    for tokens in wordnet_negated_train_tokens
]
lemmatized_wordnet_negated_tokens = [
    lemmatize_tokens(lemmatizer, tokens)
    for tokens in wordnet_negated_validation_tokens
]

# Convert into tuples
```

```
lemmatized_wordnet_negated_train_data_tuples =␣
 ↪list(zip(lemmatized_wordnet_negated_train_tokens,␣
 ↪original_dataset_train_labels))
lemmatized_wordnet_negated_validation_data_tuples =␣
 ↪list(zip(lemmatized_wordnet_negated_tokens,␣
 ↪original_dataset_validation_labels))

# Create new vocabulary list based on lemmas
vocabulary_list = flatten_list_of_lists(lemmatized_wordnet_negated_train_tokens)
# Create a Freq Distribution of the lemmatized tokens
all_words = nltk.FreqDist(w for w in vocabulary_list)

# Get the accuracies and macro-average F1 scores for different numbers of word␣
 ↪features
top_word_counts, accuracies, avg_f1_scores = ␣
 ↪calculate_metrics_for_different_vocab_size_features(
                                                                               ␣
 ↪                    400,
                                                                               ␣
 ↪                    1301,
                                                                               ␣
 ↪                    all_words,
                                                                               ␣
 ↪                    lemmatized_wordnet_negated_train_data_tuples,
                                                                               ␣
 ↪                    lemmatized_wordnet_negated_validation_data_tuples
                                                                               ␣
 ↪                  )
# Plot the results
plot_word_feature_counts_against_scores(top_word_counts, accuracies,␣
 ↪avg_f1_scores,
                                        'Accuracy and Macro Avg F1 Score vs.␣
 ↪Vocabulary Size (WordNet Negation with Lemmatized Tokens)')
```



These results indicate that lemmatization has been unsuccessful in improving the performance of the classifier beyond the baseline either, as the accuracy and macro-average F1-score are both lower than the original values of 73% and 53%.

2.2.7 Feature Engineering Experiment #7 for Statistical Classifier with Original Dataset Split: Negation Handling using WordNet and AFINN Sentiment Lexicon

The experiments conducted up to this point using standard text-processing techniques have failed to increase the macro-average F1-score beyond 55%. A new approach will now be explored involving generating sentiment polarity scores for the word features using a sentiment lexicon: a manually-curated list of negative and positive words. Jurafsky and Martin list a wide range of possible sentiment lexicons, such as the General Inquirer, MPQA Subjectivity Lexicon (for detecting objective/subjective conntations of words), and Hu and Liu's polarity lexicon (Jurafsky & Martin, 2024).

For this trial, I will use the AFINN polarity lexicon because of its simplicity (rakshita_iyer, 2023) and easy integrability with the Python environment.

After downloading the lexicon, I will add features representing the AFINN sentiment score for each word to the dictionary containing word features used previously in this project. In the AFINN lexicon, a score of 0.0 indicates neutral sentiment, a negative score indicates negative sentiment, and a positive score indicates positive sentiment. However, some limitations of using this lexicon for feature engineering should be outlined: as the lexicon outputs scores for individual words, the word "terribly" in the context of "they were terribly happy" would lead to a negative score being assigned to "terribly", even though the meaning here is to strengthen the positive affect expressed by "happy". Instead, the negative score for "terribly" will cancel out the positive score for "happy", leading to a neutral score.

This sort of combination of typically negative modifiers to augment the positive affect of other words can appear frequently in literary language. As a result, sentiment lexicons might not be sufficient to achieve higher performance when classifying the sentiment of poetic texts. As such, this highlights a possible limitation of using word-counting and feature-engineering with a basic statistical classifier for poetic texts. When applied to product reviews or political opinions (which typically use more direct, typical and straightforward language), these approaches might be more useful: however, it may be that the dependence of words on the *context* of the other words' meanings and positions in a document is even more important when handling literary texts.

[62]:
```python
# Instantiate the lexicon
afinn = Afinn()

# Adapt the doc_features function to include new features containing AFINN␣
↪sentiment scores
def doc_features_with_afinn_sentiment_scores(document, word_features):
    """
        Creates a dictionary of document features for inputs to the NLTK␣
↪Multinomial Naive Bayes classifier,
        with features marking the absence/presence of a subset of the␣
↪vocabulary in the inputted document,
        as well as features containing the sentiment scores of the document's␣
↪words.
        Inputs:
            document = list of tokens representing a sample
            word_features = subset of the vocabulary to use as word features
        Outputs:
            a feature set containing features indicating the presence or␣
↪absence of words in the document
            as well as the sentiment scores for the words in the document.
    """
    # Remove duplicate words from the document (line of poetry)
    document_words = set(document)
    # Create a features dict to represent the word features
    features = {}
    # Iterate over the top N vocabulary words (word_features) and create a␣
↪dict-key for that word, with the dict-value signalling whether the
    # word occurs in the document (line of poetry) or not.
    # Also add a new key-value pair to the features dictionary indicating the␣
↪sentiment score of the word in question
    for word in word_features:
        features[f"contains({word})"] = (word in document_words)
        features[f"sentiment_score({word})"] = afinn.score(word)
    return features


def afinn_calculate_metrics_for_different_vocab_size_features(
    lowest_num_words_limit, # lower end of range for how many words to use
```

```
    highest_num_words_limit, # higher end of range for how many words to use
    all_words, # the freq dist of words ordered by most common to least common
    train_tuples, # training data tuples of form (sample, label)
    val_tuples,
    step_size=50, # interval size between numbers of words to test
):
    """
        A function that takes in a range of values for the different nr of most␣
↪common words to use as features
        and then calculates the accuracies and macro-average f1-scores for each␣
↪nr of most common words.
        Similar to the `calculate_metrics_for_different_vocab_size_features`, ␣
↪but this function
        uses `doc_features_with_afinn_sentiment_scores` instead of the␣
↪`doc_features` function which does
        NOT contain sentiment scores for each word feature.
    """
    top_word_counts = np.arange(lowest_num_words_limit,␣
↪highest_num_words_limit, step_size)
    accuracies = [] # store accuracies for each nr of top words used in here
    avg_f1_scores = [] # store macro f1 scores for each nr of top words used in␣
↪here

    # Iterate over the array of word feature numbers to use (i.e. vocab subset␣
↪to use in features)
    for vocab_size in top_word_counts:
        print(vocab_size)
        # store the list of top "vocab_size" words to use
        word_features = list(all_words)[:vocab_size]
        # Create the feature sets based on the top vocab_size word features for␣
↪training and validation splits
        train_data_featuresets =␣
↪[(doc_features_with_afinn_sentiment_scores(doc, word_features), label) for␣
↪(doc, label) in train_tuples]
        validation_data_featuresets =␣
↪[(doc_features_with_afinn_sentiment_scores(doc, word_features), label) for␣
↪(doc, label) in val_tuples]
        # Train a NB classifier
        NBclassifier = nltk.NaiveBayesClassifier.train(train_data_featuresets)
        accuracy = nltk.classify.accuracy(NBclassifier,␣
↪validation_data_featuresets)
        accuracies.append(accuracy)
        # Now obtain the macro-avg F1 scores by storing predicted labels
        validation_predictions= []

        # Iterate over each validation feature set and get the predicted label
```

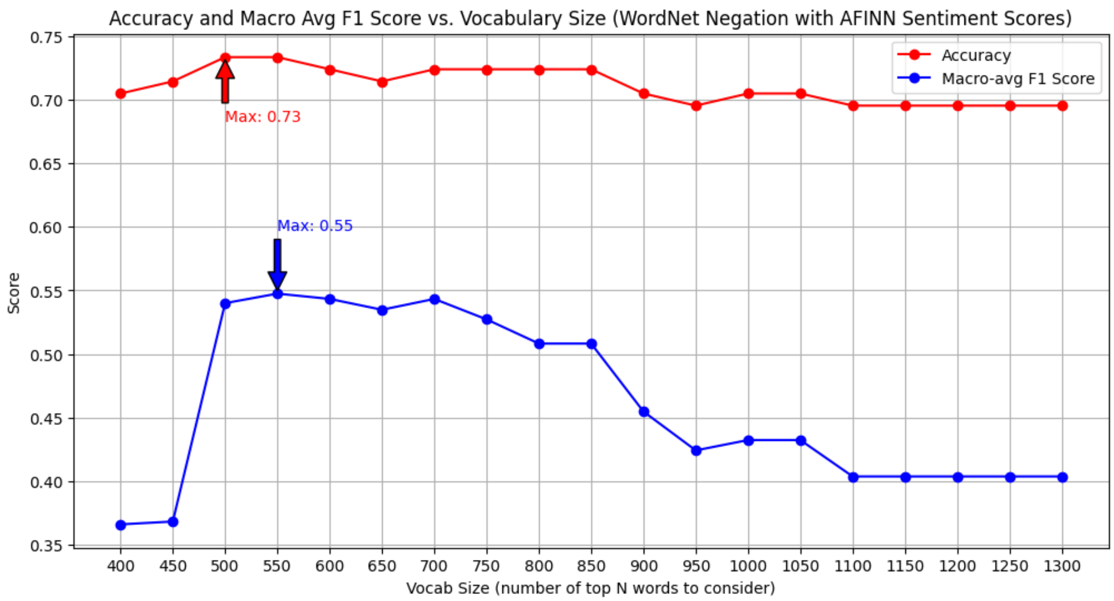
```
        for features_dict, label in validation_data_featuresets:
            predicted_label = NBclassifier.classify(features_dict)
            validation_predictions.append(predicted_label)

        # Retrieve macro-average F1 score from classification report and store␣
↪it in avg_f1_scores
        class_report = classification_report(
            original_dataset_validation_labels,
            validation_predictions,
            output_dict=True,  # Return report as a dictionary (easier to␣
↪access metrics)
            # Set the score to 0 if "UndefinedMetricWarning" appears because␣
↪either recall or precision for a class are 0.0
            zero_division=0
        )

        macro_avg_f1 = class_report['macro avg']['f1-score']
        avg_f1_scores.append(macro_avg_f1)

    return top_word_counts, accuracies, avg_f1_scores
```

[445]:

```
## RUN THE EXPERIMENT AGAIN USING THE WORDNET NEGATION FUNCTION BUT WITH ADDDED␣
↪AFINN LEXICON SENTIMENT SCORES

# Use the wordnet_negated tokens (train and validation) for the vocabulary set
vocabulary_list = flatten_list_of_lists(wordnet_negated_train_tokens)
vocabulary_set = set(vocabulary_list)
# create a freq dist of the words, convert to lower case
all_words = nltk.FreqDist(w for w in vocabulary_list)

# Get the accuracies and f1 scores for different numbers of word features to␣
↪use to choose the best nr of word features for the negated sets
top_word_counts, accuracies, avg_f1_scores = ␣
↪afinn_calculate_metrics_for_different_vocab_size_features(
                                                                           ␣
↪                      400,
                                                                           ␣
↪                      1301,
                                                                           ␣
↪                      all_words,
                                                                           ␣
↪                      wordnet_negated_train_data_tuples,
                                                                           ␣
↪                      wordnet_negated_validation_data_tuples
                                                                           ␣
↪                    )
```

```python
# Plot the results
plot_word_feature_counts_against_scores(top_word_counts, accuracies,␣
↪avg_f1_scores,
                                         "Accuracy and Macro Avg F1 Score vs.␣
↪Vocabulary Size (WordNet Negation with AFINN Sentiment Scores)")
```

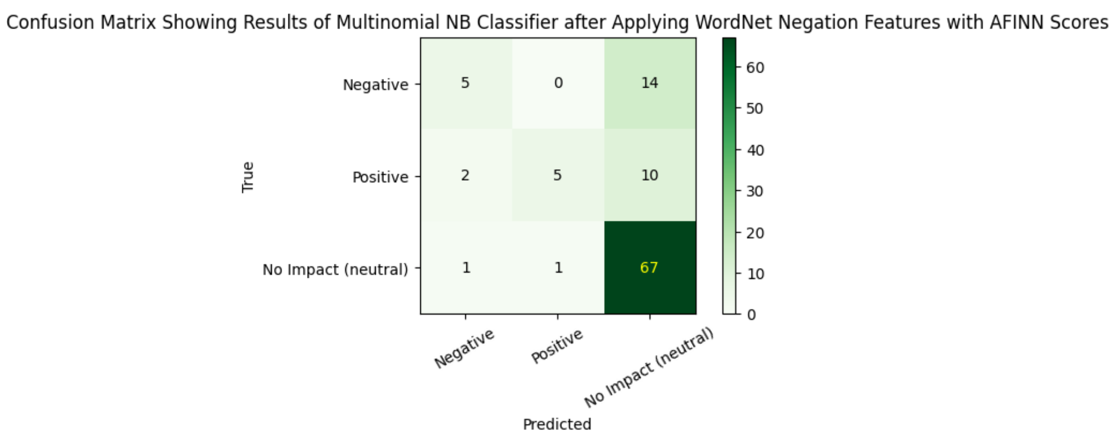
```
400
450
500
550
600
650
700
750
800
850
900
950
1000
1050
1100
1150
1200
1250
1300
```



```
[446]: # Train a new NB classifier on the negated word sets using the best nr of word␣
        ↪features (550) using WordNet negation and sentiment scores
N = 550
word_features = list(all_words)[:N]

# Create feature sets out of the train and validation document-tuples by␣
 ↪applying the doc_features_with_afinn_sentiment_scores function
afinn_wordnet_negated_train_data_featuresets =␣
 ↪[(doc_features_with_afinn_sentiment_scores(doc, word_features), label)
                                                    for (doc, label) in␣
 ↪wordnet_negated_train_data_tuples
                                                ]
afinn_wordnet_negated_validation_data_featuresets =␣
 ↪[(doc_features_with_afinn_sentiment_scores(doc, word_features), label)
                                                         for (doc, label) in␣
 ↪wordnet_negated_validation_data_tuples
                                                    ]

# Train the classifier
NBclassifier = nltk.NaiveBayesClassifier.
 ↪train(afinn_wordnet_negated_train_data_featuresets)
# Print the accuracy score
print(f"Accuracy Score: {nltk.classify.accuracy(NBclassifier,␣
 ↪afinn_wordnet_negated_validation_data_featuresets)}")

# Get predictions
afinn_wordnet_negated_dataset_validation_predictions = [] # store predicted␣
 ↪labels in here.
for features_dict, label in afinn_wordnet_negated_validation_data_featuresets:
    # Apply the NB model to output the predicted label for each sample in the␣
 ↪validation set
    predicted_label = NBclassifier.classify(features_dict)
    # Store the predicted label
    afinn_wordnet_negated_dataset_validation_predictions.append(predicted_label)

# Print classification report to view the precision, recall, f1 score for each␣
 ↪class and the macro-averages
print("CLASSIFICATION REPORT:\n")
print(classification_report(
    original_dataset_validation_labels, # labels are the same for the negated␣
 ↪featureset as for the original featureset
    afinn_wordnet_negated_dataset_validation_predictions,
    target_names=label_names)
)

# Apply confusion matrix function to the new results
```

```
generate_and_show_confusion_matrix(
    original_dataset_validation_labels,
    afinn_wordnet_negated_dataset_validation_predictions,
    label_names=label_names,
    classifier_description="Confusion Matrix Showing Results of Multinomial NB␣
 ↪Classifier after Applying WordNet Negation Features with AFINN Scores"
)
```

```
Accuracy Score: 0.7333333333333333
CLASSIFICATION REPORT:

                     precision    recall  f1-score   support

           Negative       0.62      0.26      0.37        19
           Positive       0.83      0.29      0.43        17
No Impact (neutral)       0.74      0.97      0.84        69

           accuracy                           0.73       105
          macro avg       0.73      0.51      0.55       105
       weighted avg       0.73      0.73      0.69       105

[0 1 2]
```



The accuracy score is still 73% and the average F1-score is still 55% (the same as using the WordNet negation handling alone). As a result, the AFINN sentiment scores have not fulfilled the aim of improving the performance of the Naive Bayes classifier.

2.2.8 Feature Engineering Experiment #8 for Statistical Classifier with Original Dataset Split: Negation Handling using WordNet and SentiWordNet Sentiment Lexicon

I will now try to add the positive and negative scores for the string of tokens as features, using a different sentiment lexicon called SentiWordNet (based on WordNet). "Every synset *s* is associated with a Pos(s): a positivity score Neg(s): a negativity score Obj(s): an objectivity (neutrality) score" (Sharma, 2021). SentiWordNet takes a word's POS-tag into account when calculating the scores, which may hopefully lead to improved scores, as more information is provided about the grammatical structure when trying to extract a word's core meaning.

[55]:
```
##################### CODE ADAPTED FROM: https://medium.com/@nidhikayadav/
↪sentiment-analysis-with-python-sentiwordnet-fd07ffc557 ##################
def get_sentiwordnet_scores_from_tokens(tokens):
    """
        Returns the positive and negative sentiment score for a list of tokens␣
↪using SentiWordNet sentiment lexicon.
        Input:
            tokens = a list of tokens
        Output:
            positive_sentiment = a number representing the total positive␣
↪sentiment for the set of tokens
            negative_sentiment = a number representing the total negative␣
↪sentiment for the set of tokens
    """
    # Remove any bigrams (will use them below in this project!) to calculate␣
↪the sentiment score for the sample
    tokens_without_bigrams = [token for token in tokens if not␣
↪isinstance(token, tuple)]

    pos_tagged_tokens = nltk.pos_tag(tokens_without_bigrams)
    positive_sentiment = 0.0 # set counter for sentiment scores
    negative_sentiment = 0.0
    # Iterate over the token-pos_tag tuples
    for word_tag_pair in pos_tagged_tokens:
        # Store the token/word
        word = word_tag_pair[0]
        # Store the corresponding pos_tag
        pos_tag = word_tag_pair[1]
        # Convert the NLTK pos_tags to WordNet format for the most relevant pos␣
↪tags
        if pos_tag.startswith('J'):
            pos_tag =  wordnet.ADJ
        elif pos_tag.startswith('R'):
            pos_tag =  wordnet.ADV
        elif pos_tag.startswith('N'):
            pos_tag =  wordnet.NOUN
        else:
            continue # Continue if none of the above POS-tags apply
        # Get synsets for the word using its POS tag
        word_synsets = wordnet.synsets(word, pos=pos_tag)
        # If cannot find synset, skip it
        if not word_synsets:
```

```
            continue
        # Get first synset
        top_synset = word_synsets[0]
        # Fetch the senti_synset which contains sentiment scores for this synset
        senti_word_net = swn.senti_synset(top_synset.name())
        # Fetch positive and negative scores for the synset and add the score␣
 ↪to the running totals
        positive_sentiment += senti_word_net.pos_score()
        negative_sentiment +=  senti_word_net.neg_score()
    return positive_sentiment, negative_sentiment

##########################################################################################
```

```
[56]: # Test
tokens = word_tokenize("The meadow was terribly beautiful.")
p,n = get_sentiwordnet_scores_from_tokens(tokens)
print(f"Positive score: {p}, Negative score: {n}")
```

```
Positive score: 1.0, Negative score: 0.0
```

We can see that the SentiWordNet approach is slightly more nuanced (probably due to the inclusion of part-of-speech tags) than the AFINN method, as the sentence used as an example above has been given a positive sentiment score despite the use of the word "terribly" to modify the positive word "beautiful". This highlights the ability to capture more subtle affective connotations using POS-tags. We will now apply the total positive score and total negative scores for a sample as features to the document features dictionary by defining the following function.

```
[64]: # Adapt the doc_features function to include features containing SentiWordNet␣
 ↪sentiment scores
def doc_features_with_swn_sentiment_scores(document, word_features):
    """
        Constructs a features dictionary representing the presence or absence␣
 ↪of each of the word features in the document,
        as well as the sum of positive and negative SentiWordNet sentiment␣
 ↪scores for each token.
        Inputs:
            document = a list of tokens
            word_features = a list containing a subset of the total vocabulary␣
 ↪in the training corpus
        Outputs:
            features = a dictionary containing key-value pairs representing the␣
 ↪absence or presence of each of the word_features
            in word_features in the document, with the word feature as the key␣
 ↪and True/False as the value,
            and positive and negative sentiment scores for the document derived␣
 ↪using the SentiWordNet sentiment lexicon
```

```python
    """

    # Get the swn pos and neg summed scores for the document (a list of tokens)
    positive_score, negative_score =␣
↪get_sentiwordnet_scores_from_tokens(document)

    # Remove duplicate words from the document (line of poetry)
    document_words = set(document)

    # Create a features dict to represent the word features
    features = {}
    # Iterate over the top N vocabulary words (word_features) and create a␣
↪dict-key for that word, with the dict-value signalling whether the
    # word occurs in the document (line of poetry) or not.
    # Also add a new key-value pair to the features dictionary indicating the␣
↪sentiment score of the word in question
    for word in word_features:
        features[f"contains({word})"] = (word in document_words)

    # Add features representing swn positive and negative scores
    features["positive_sentiment"] = positive_score
    features["negative_sentiment"] = negative_score

    return features

"""
    A function that takes in a range of values for the different nr of most␣
↪common words to use as features
    and then calculates the accuracies and average f1-scores for each nr of␣
↪most common words.
    In contrast to the␣
↪`afinn_calculate_metrics_for_different_vocab_size_features`, this function
    uses `doc_features_with_swn_sentiment_scores`  instead of the␣
↪`doc_features_with_afinn_sentiment_scores` function
    to use SentiWordNet summed positive and negative scores instead of AFINN␣
↪sentiment scores.
"""
def swn_calculate_metrics_for_different_vocab_size_features(
    lowest_num_words_limit, # lower end of range for how many words to use
    highest_num_words_limit, # higher end of range for how many words to use
    all_words, # the freq dist of words ordered by most common to least common
    train_tuples, # training data tuples of form (sample, label)
    val_tuples,
    step_size=50, # interval size between numbers of words to test
):
```

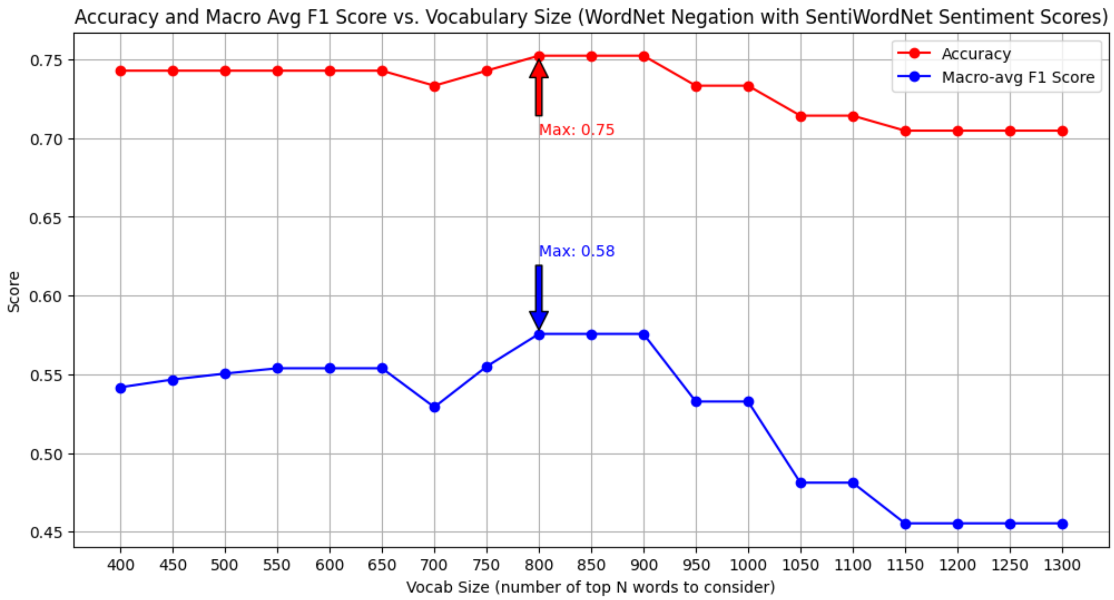
```
    top_word_counts = np.arange(lowest_num_words_limit,␣
↪highest_num_words_limit, step_size)
    accuracies = [] # store accuracies for each nr of top words used in here
    avg_f1_scores = [] # store macro f1 scores for each nr of top words used in␣
↪here

    # iterate over the array of top-word counts to use (i.e. vocab subset to␣
↪use in features)
    for vocab_size in top_word_counts:
        print(vocab_size)
        # store the list of top "vocab_size" words to use
        word_features = list(all_words)[:vocab_size]
        # get the featuresets based on the top N word features for training and␣
↪validation splits
        train_data_featuresets = [(doc_features_with_swn_sentiment_scores(doc,␣
↪word_features), label) for (doc, label) in train_tuples]
        validation_data_featuresets =␣
↪[(doc_features_with_swn_sentiment_scores(doc, word_features), label) for␣
↪(doc, label) in val_tuples]
        # train a NB classifier and append accuracy score to the above-defined␣
↪list
        NBclassifier = nltk.NaiveBayesClassifier.train(train_data_featuresets)
        accuracy = nltk.classify.accuracy(NBclassifier,␣
↪validation_data_featuresets)
        accuracies.append(accuracy)
        # # now get the macro avg f1 scores (more complicated)
        # # store all the predicted labels here
        validation_predictions= []

        # iterate over each validation featurset and get the predicted label
        for features_dict, label in validation_data_featuresets:
            predicted_label = NBclassifier.classify(features_dict)
            validation_predictions.append(predicted_label)

        # Retrieve macro-average F1 score from classification report and store␣
↪it in avg_f1_scores
        class_report = classification_report(
            original_dataset_validation_labels,
            validation_predictions,
            output_dict=True,  # Return report as a dictionary (easier to␣
↪access metrics)
            # Set the score to 0 if "UndefinedMetricWarning" appears because␣
↪either recall or precision for a class are 0.0
            zero_division=0
        )
```

```
        macro_avg_f1 = class_report['macro avg']['f1-score']
        avg_f1_scores.append(macro_avg_f1)
    return top_word_counts, accuracies, avg_f1_scores
```

[469]:
```
# Run the experiment again to find optimal number of word features for feature␣
↪sets with SentiWordNet scores

# Use the WordNet_negated tokens (train and validation) for the vocabulary
vocabulary_list = flatten_list_of_lists(wordnet_negated_train_tokens)
# Create a Freq Distribution of the words
all_words = nltk.FreqDist(w for w in vocabulary_list)

# Fetch the accuracies and f1 scores for different numbers of word features to␣
↪use to choose the best nr of word features for the negated sets
top_word_counts, accuracies, avg_f1_scores = ␣
↪swn_calculate_metrics_for_different_vocab_size_features(
                                                                              ␣
↪                      400,
                                                                              ␣
↪                      1301,
                                                                              ␣
↪                      all_words,
                                                                              ␣
↪                      wordnet_negated_train_data_tuples,
                                                                              ␣
↪                      wordnet_negated_validation_data_tuples
                                                                              ␣
↪                    )
# Plot the results
plot_word_feature_counts_against_scores(top_word_counts, accuracies,␣
↪avg_f1_scores,
                                        "Accuracy and Macro Avg F1 Score vs.␣
↪Vocabulary Size (WordNet Negation with SentiWordNet Sentiment Scores)")
```

```
400
450
500
550
600
650
700
750
800
850
900
950
1000
```

1050
1100
1150
1200
1250
1300

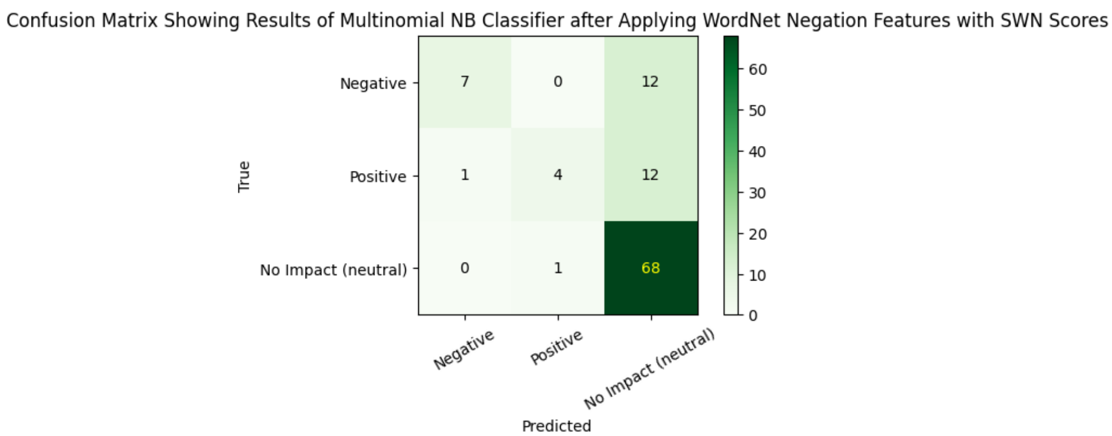


[470]:
```
# Train a new NB classifier on the negated word sets using the best number of␣
↪word features (800) using WordNet negation and SentiWordNet scores
N = 800
word_features = list(all_words)[:N]

# Create feature sets out of the train and validation document-tuples by␣
↪applying the doc_features_with_swn_sentiment_scores function
swn_wordnet_negated_train_data_featuresets =␣
↪[(doc_features_with_swn_sentiment_scores(doc, word_features), label)
                                                    for (doc, label) in␣
↪wordnet_negated_train_data_tuples
                                               ]
swn_wordnet_negated_validation_data_featuresets =␣
↪[(doc_features_with_swn_sentiment_scores(doc, word_features), label)
                                                         for (doc, label) in␣
↪wordnet_negated_validation_data_tuples
                                                    ]

# Train classifier on new featuresets
```

```python
NBclassifier = nltk.NaiveBayesClassifier.
 ↪train(swn_wordnet_negated_train_data_featuresets)
# Print the accuracy score
print(f"Accuracy Score: {nltk.classify.accuracy(NBclassifier,␣
 ↪swn_wordnet_negated_validation_data_featuresets)}")

# Get predictions
swn_wordnet_negated_dataset_validation_predictions = [] # store predicted␣
 ↪labels in here.
for features_dict, label in swn_wordnet_negated_validation_data_featuresets:
    # apply the NB model to output the predicted label for each sample in the␣
 ↪validation set
    predicted_label = NBclassifier.classify(features_dict)
    # store the predicted label
    swn_wordnet_negated_dataset_validation_predictions.append(predicted_label)

# Print classification report to view the precision, recall, f1 score for each␣
 ↪class and macro-averages
print("CLASSIFICATION REPORT:\n")
print(classification_report(
    original_dataset_validation_labels, # labels are the same for the negated␣
 ↪featureset as for the original featureset
    swn_wordnet_negated_dataset_validation_predictions,
    target_names=label_names)
)

# Apply the confusion matrix function to the new results
generate_and_show_confusion_matrix(
    original_dataset_validation_labels,
    swn_wordnet_negated_dataset_validation_predictions,
    label_names=label_names,
    classifier_description="Confusion Matrix Showing Results of Multinomial NB␣
 ↪Classifier after Applying WordNet Negation Features with SWN Scores"
)
```

```
Accuracy Score: 0.7523809523809524
CLASSIFICATION REPORT:

                     precision    recall  f1-score   support

           Negative       0.88      0.37      0.52        19
           Positive       0.80      0.24      0.36        17
No Impact (neutral)       0.74      0.99      0.84        69

           accuracy                           0.75       105
          macro avg       0.80      0.53      0.58       105
       weighted avg       0.77      0.75      0.71       105
```

[0 1 2]



While the performance is still not very good, with many misclassifications for the positive and negative classes, it is an improvement in comparison to the previous experiments.

The number of correctly classified negative samples has gone up from 5 to 7 compared with the previous results, raising accuracy from 73% to 75%, and macro average F1-score from 55% to 58% when compared to using the classifier with WordNet handling but no SentiWordNet scores.

This improvement might be due to the use of POS-tagging to calculate the SentiWordNet sentiment scores. As such this finding might highlight the importance of context and grammatical structure when trying to detect sentiment in verse texts.

2.2.9 Feature Engineering Experiment #9 for Statistical Classifier with Original Dataset Split: Negation Handling using WordNet and SentiWordNet Sentiment Lexicon and Bigrams

We can experiment further with the results achieved by adding SentiWordNet scores to the features dictionary by also creating word features out of the top 'N' bigrams (pairs of words). Tan, Wang, and Lee (2002) have demonstrated that "bigrams can substantially raise the quality of feature sets, showing increases in the break-even points and F1 measures". Including bigrams as features might allow the classifier to learn new patterns by drawing attention to words that frequently go together, and to represent frequently-occuring contextual relationships.

[65]:
```
## Adapt document features function to include bigrams combined with␣
↪SentiWordNet sentiment scores for individual words

def doc_features_with_swn_sentiment_scores_bigrams(document, word_features):
        """
        Constructs a features dictionary representing the presence or absence␣
↪of each of the word features in the document,
        as well as the sum of positive and negative SentiWordNet sentiment␣
↪scores for each token.
```

```
        Inputs:
            document = a list of tokens, containing bigram tuples as well as␣
↪unigram strings
            word_features = a list containing a subset of the total vocabulary␣
↪in the training corpus, including bigrams
        Outputs:
            features = a dictionary containing key-value pairs representing the␣
↪absence or presence of each of the word_features
            in word_features in the document, with the word feature as the key␣
↪and True/False as the value,
            and positive and negative sentiment scores for the document derived␣
↪using the SentiWordNet sentiment lexicon

    """
    # Get the SentiWordNet pos and neg summed scores for the document (a list␣
↪of tokens)
    positive_score, negative_score =␣
↪get_sentiwordnet_scores_from_tokens(document)

    # Remove duplicate words from the document (a tokenized line of poetry)
    document_words = set(document)
    # Extract the set of bigrams from the document
    document_bigrams = set(bigrams(document))

    # Create a features dictionary to represent the word features
    features = {}
    # Iterate over the top N vocabulary words (word_features) and create a␣
↪dict-key for that word, with the dict-value signalling whether the
    # word or bigram occurs in the document (line of poetry) or not.
    for word in word_features:
        features[f"contains({word})"] = (word in document_words or word in␣
↪document_bigrams)

    # Add features representing swn positive and negative scores
    features["positive_sentiment"] = positive_score
    features["negative_sentiment"] = negative_score

    return features


def swn_calculate_metrics_for_different_vocab_size_features_with_bigrams(
    lowest_num_words_limit, # lower end of range for how many words to use
    highest_num_words_limit, # higher end of range for how many words to use
    all_grams, # the freq dist of unigrams and bigrams ordered by most common␣
↪to least common
    train_tuples, # training data tuples of form (sample, label)
```

```
    val_tuples,
    step_size=50, # interval size between numbers of words to test
):
    """
        A function that takes in a range of values for the most common words to␣
↪use as features
        and then calculates the accuracies and macro-average F1-scores for each␣
↪nr of most common words.
        Similar to the␣
↪`afinn_calculate_metrics_for_different_vocab_size_features`. However, this␣
↪function uses
        `doc_features_with_swn_sentiment_scores`  instead of the␣
↪`doc_features_with_afinn_sentiment_scores` function
        to use SentiWordNet summed positive and negative scores instead of␣
↪AFINN sentiment scores.
        This version of the function also includes the possibility of using␣
↪bigrams as features.
        Same set of input arguments as␣
↪calculate_metrics_for_different_vocab_size_features.
    """
    top_word_counts = np.arange(lowest_num_words_limit,␣
↪highest_num_words_limit, step_size)
    accuracies = [] # store accuracies for each nr of top words used in here
    avg_f1_scores = [] # store macro f1 scores for each nr of top words used in␣
↪here

    # Iterate over the array of word features to use (the vocab subset to use␣
↪in features)
    for vocab_size in top_word_counts:
        print(vocab_size)
        # Store the list of top "vocab_size" words to use
        word_features = list(all_grams)[:vocab_size]
        # Fetch the feature sets based on the top N word features for both the␣
↪training and validation splits
        train_data_featuresets =␣
↪[(doc_features_with_swn_sentiment_scores_bigrams(doc, word_features), label)␣
↪for (doc, label) in train_tuples]
        validation_data_featuresets =␣
↪[(doc_features_with_swn_sentiment_scores_bigrams(doc, word_features), label)␣
↪for (doc, label) in val_tuples]
        # Train a NB classifier and append accuracy score to the above-defined␣
↪list
        NBclassifier = nltk.NaiveBayesClassifier.train(train_data_featuresets)
        accuracy = nltk.classify.accuracy(NBclassifier,␣
↪validation_data_featuresets)
        accuracies.append(accuracy)
```

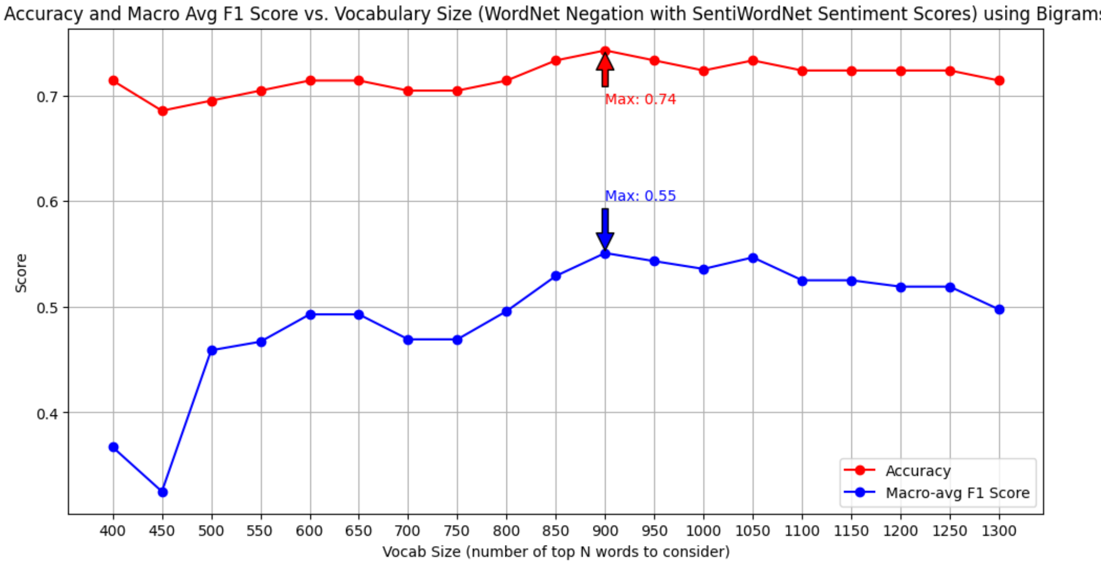
```python
        # Get predictions
        validation_predictions= []
        # Iterate over each validation featurset and get the predicted label
        for features_dict, label in validation_data_featuresets:
            predicted_label = NBclassifier.classify(features_dict)
            validation_predictions.append(predicted_label)
        # Retrieve the macro-average F1 score from classification report and␣
↪store it in avg_f1_scores
        class_report = classification_report(
            original_dataset_validation_labels,
            validation_predictions,
            output_dict=True,  # Return report as a dictionary (easier to␣
↪access metrics)
            # Set the score to 0 if "UndefinedMetricWarning" appears because␣
↪either recall or precision for a class are 0.0
            zero_division=0
        )

        macro_avg_f1 = class_report['macro avg']['f1-score']
        avg_f1_scores.append(macro_avg_f1)

    return top_word_counts, accuracies, avg_f1_scores
```

[507]:

```python
# Get a list of tokens and bigrams for each sample handled using the WordNet␣
↪negation strategies
wordnet_negated_train_tokens_with_bigrams = [list(bigrams(sample)) + sample for␣
↪sample in wordnet_negated_train_tokens]
wordnet_negated_validation_tokens_with_bigrams = [list(bigrams(sample)) +␣
↪sample for sample in wordnet_negated_validation_tokens]

# Convert samples-labels into tuples
wordnet_negated_train_data_tuples_with_bigrams =␣
↪list(zip(wordnet_negated_train_tokens_with_bigrams,␣
↪original_dataset_train_labels))
wordnet_negated_validation_data_tuples_with_bigrams =␣
↪list(zip(wordnet_negated_validation_tokens_with_bigrams,␣
↪original_dataset_validation_labels))

# Flatten the bigrams and tokens into a vocabulary list
vocabulary_list = [grams for sample in␣
↪wordnet_negated_train_tokens_with_bigrams for grams in sample]
# Create a Frequency Distribution for the words and bigrams
all_grams = nltk.FreqDist(grams for grams in vocabulary_list)
```

```
# Calculate the accuracies and f1 scores for different numbers of word features␣
↪to use to choose the best number of word features
top_word_counts, accuracies, avg_f1_scores = ␣
↪swn_calculate_metrics_for_different_vocab_size_features_with_bigrams(
                                                                                ␣
↪                    400,
                                                                                ␣
↪                    1301,
                                                                                ␣
↪                    all_grams, # Include unigrams and bigrams here
                                                                                ␣
↪                    wordnet_negated_train_data_tuples_with_bigrams,
                                                                                ␣
↪                    wordnet_negated_validation_data_tuples_with_bigrams
                                                                                ␣
↪                  )
# Plot the results
plot_word_feature_counts_against_scores(
    top_word_counts, accuracies, avg_f1_scores,
    "Accuracy and Macro Avg F1 Score vs. Vocabulary Size (WordNet Negation with␣
↪SentiWordNet Sentiment Scores) using Bigrams"
)
```

400
450
500
550
600
650
700
750
800
850
900
950
1000
1050
1100
1150
1200
1250
1300



Adding bigrams to the feature set has not outperformed the effectiveness of the classifier that uses SentiWordNet scores, as the accuracy has dropped by 1% and the macro-average F1 score has decreased by 3% compared to the previous experiment.

2.3 Evaluating the Best-Performing (WordNet Negation Handling and SentiWordNet Scores) Feature Sets with a Naive Bayes Classifier on the Original Test Set

So far we have been running the experiments on the (original) validation set from the HuggingFace dataset.

The experiments on this validation set performed above show that the best performance was achieved using the WordNet negation features and SentiWordNet scores as features, with 75% accuracy (compared to the 73% baseline) and 58% average F1-score (compared to 53% for the baseline).

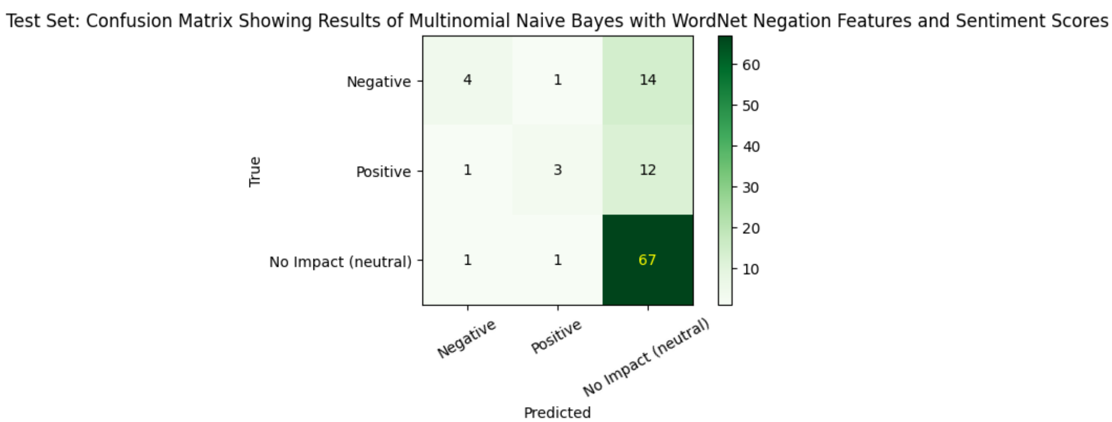
These best-performing pre-processing methods will now be applied to the original test set, to see how well the Multinomial Naive Bayes classifier generalizes to unseen data when paired with these feature extraction techniques.

[512]:
```
# Apply negation handling to original test samples
wordnet_negated_test_tokens = [handle_negation_with_wordnet(tokens,␣
↪negation_patterns) for tokens in original_dataset_test_tokens]
# Convert test samples into tuples (tokens, label)
wordnet_negated_test_data_tuples = list(zip(wordnet_negated_test_tokens,␣
↪original_dataset_test_labels))

# Get vocabulary list from WordNet negated tokens
vocabulary_list = flatten_list_of_lists(wordnet_negated_train_tokens)
# create a freq dist of the words, convert to lower case
```

```
all_words = nltk.FreqDist(w for w in vocabulary_list)

# Use the best number of word features (800) found for the validation set when␣
↪using WordNet negation and SentiWordNet sentiment scores
N = 800
word_features = list(all_words)[:N]

# Create feature sets out of the train and validation document-tuples by␣
↪applying the best-performing feature-extracting function
swn_wordnet_negated_train_data_featuresets =␣
↪[(doc_features_with_swn_sentiment_scores(doc, word_features), label)
                                                    for (doc, label) in␣
↪wordnet_negated_train_data_tuples
                                               ]
swn_wordnet_negated_test_data_featuresets =␣
↪[(doc_features_with_swn_sentiment_scores(doc, word_features), label)
                                                         for (doc, label) in␣
↪wordnet_negated_test_data_tuples
                                                    ]

NBclassifier = nltk.NaiveBayesClassifier.
↪train(swn_wordnet_negated_train_data_featuresets)

# Print the accuracy score
print(f"Accuracy Score: {nltk.classify.accuracy(NBclassifier,␣
↪swn_wordnet_negated_test_data_featuresets)}")

# Generate predictions
swn_wordnet_negated_dataset_test_predictions = []
for features_dict, label in swn_wordnet_negated_test_data_featuresets:
    predicted_label = NBclassifier.classify(features_dict)
    swn_wordnet_negated_dataset_test_predictions.append(predicted_label)

# Print thes classification report to view the precision, recall, f1 score for␣
↪each class and macro-averages for the test set
print("CLASSIFICATION REPORT:\n")
print(classification_report(
    original_dataset_test_labels,
    swn_wordnet_negated_dataset_test_predictions,
    target_names=label_names)
)

# Generate confusion matrix for performance on test set
generate_and_show_confusion_matrix(
    original_dataset_test_labels,
    swn_wordnet_negated_dataset_test_predictions,
```

```
    label_names=label_names,
    classifier_description="Test Set: Confusion Matrix Showing Results of␣
↪Multinomial Naive Bayes with WordNet Negation Features and Sentiment Scores"
)
```

```
Accuracy Score: 0.7115384615384616
CLASSIFICATION REPORT:

                      precision    recall  f1-score   support

           Negative       0.67      0.21      0.32        19
           Positive       0.60      0.19      0.29        16
No Impact (neutral)       0.72      0.97      0.83        69

           accuracy                           0.71       104
          macro avg       0.66      0.46      0.48       104
       weighted avg       0.69      0.71      0.65       104

[0 1 2]
```



The performance of this classifier on the test set was very poor, as the macro-average F1-score was only 48% (10% than on the validation set), and the confusion matrix shows that barely positive or negative samples were correctly classified.

To a large extent, this is probably due to the small size of the validation and test sets (only about 100 samples) and the disproportionally small number of positive and negative samples compared to the dominant "neutral" class. As such, the classifier may have struggled to extract significant patterns due to the lack of data. Additionally, conducting all the different pre-processing experiments using the same validation data every time might have led to overfitting to the idiosyncracies and patterns of this particular dataset. Furthermore, the traditional approach to NLP text classification does not take into account the context of the words, and is therefore limited in its ability to infer patterns from small sets of samples, compared to a neural network that can capture more complex relationships and dependencies in sequential data.

As a result, I will now repeat the above experiments using five-fold stratified cross-validation that ensures that each class (including the "mixed" class) is proportionally represented in each fold, as well as in the training and test sets.

Cross-validation makes use all of the training data to evaluate the best settings and features, which is vital when the dataset is small and there are limited samples of the minority classes. It also enables the evaluation of the classifier's performance on the "mixed" sentiment class, which was missing from the validation and test sets in the original dataset splits.

Furthermore, using cross-validation and aggregating the performance scores achieved on different folds/splits of the data increases the final classifier's ability to generalize to new data, rather than overfitting to one tiny validation set.

The primary reason for using the original dataset splits as well as stratified cross-validation was to allow for comparability to already published results on the same splits. However, experimenting with the different feature sets on the same validation set every time can degrade the robustness of the classifier due to the reasons outlined above. In order to develop a more reliable classifier while considering the importance of comparability and repeatability of results, I will recombine the dataset and split it using a random_state=2 argument and saving the new dataset as a local copy, so that others can access the same dataset split.

2.4 Repeating Text Pre-Processing Experiments using the Recombined Dataset Split with Stratified Five-Fold Cross Validation

In this section, the original dataset will be merged before splitting it into a new training and test set, each of which will contain proportional representations of each label/class. Then, the training set will be used for stratified five-fold cross-validation, with the validation fold also containing proportional samples of each class. The same text pre-processing experiments as above will be repeated using this cross-validation technique, and the mean accuracies and macro-average F1-scores calculated by collecting the score for each "fold".

[3]:
```
# Load the original HuggingFace poem dataset"
original_dir = './datasets/original_poem_sentiment_dataset'
poem_dataset = load_from_disk(original_dir)

# Convert the three splits into pandas dataframes for easier viewing and␣
↪analysis of the dataset using the inbuilt 'to_pandas' method
train_df = poem_dataset['train'].to_pandas()
val_df = poem_dataset['validation'].to_pandas()
test_df = poem_dataset['test'].to_pandas()

# Combine the three sets into one whole dataset
complete_df = pd.concat([train_df, val_df, test_df], ignore_index=True) # the␣
↪ignore_index=True creates a new index to avoid overlapping indices

# Now create a new train-test split (80:20 ratio), where each class (0, 1, 2,␣
↪3) is proportionally represented in both splits
```

```
# Validation splits will be created later using five-fold stratified␣
↪cross-validation.
# Remove the 'id' column as it's not required for retrieving the samples and␣
↪labels.
complete_df = complete_df.drop(columns=['id'])

# Extract the samples into
samples = complete_df["verse_text"]
labels = complete_df["label"]

# Perform the stratified train-test split using sk-learn's train_test_split␣
↪function, inputting stratify=labels to indicate labels should be
# evenly distributed. Use random_state for reproducibility of results.
# Reference: https://scikit-learn.org/stable/modules/generated/sklearn.
↪model_selection.train_test_split.html random_state = use for reproducibility
train_samples, test_samples, train_labels, test_labels =␣
↪train_test_split(samples, labels, test_size=0.2, stratify=labels,␣
↪random_state=2)

# Concatenate the samples and labels back into pandas DataFrames (for local csv␣
↪data storage)
new_train_df = pd.concat([train_samples, train_labels], axis=1) # concatenate␣
↪columns, not rows, using axis=1 argument
new_test_df = pd.concat([test_samples, test_labels], axis=1)

# Log the results to ensure everything has worked properly.
print(f"New train data: {new_train_df[0:10]}\n")
print(f"New test data: {new_test_df[0:10]}")

# Save the new train-test split dataframes into locally-stored csv files.
new_train_df.to_csv('new_poem_train_set.csv', index=False)
new_test_df.to_csv('new_poem_test_set.csv', index=False)
```

```
New train data:                                                  verse_text  label
1053                   o that i were where helen lies      2
264                   and keep my senses straightened      2
583   his song, though very sweet, was low and faint,      3
254                    what's de use o' gittin' mopy,      2
502             she still must keep the locket to allay     2
509     i think i'll just call up my wife and tell her     2
336                                 like morning glory      1
96                        in their archetypes endure.      2
161                              whatever anybody had      2
852           and seek the danger i was forc’d to shun.      0

New test data:                                                   verse_text  label
1092                   where the warm life we cannot see--       1
```

```
213                nile shall pursue his changeless way:      2
513                 in our embraces we again enfold her,      1
1083                             of my wit or in my mind      2
155                     for the greek must ask elsewhere.     2
803             --the drones of the community; they feed      2
1000                              she sought for flowers      2
419   would my heart and life flow onward, deathward…      3
868           his silent sandals swept the mossy green;       2
901                    with england if the day go hard,       2
```

[61]:

```python
# Load the new train-test data from csv
recombined_train_df = pd.read_csv('new_poem_train_set.csv')
# Load the test dataframe from the CSV file
recombined_test_df = pd.read_csv('new_poem_test_set.csv')

# Show first few rows to ensure loading was successful
print("LOADED TRAIN DATA")
print(recombined_train_df.head())
print("length:", len(recombined_train_df))
print('\n**********************************************************************\n')
print("LOADED TEST DATA")
print(recombined_test_df.head())
print("length:", len(recombined_test_df))

# Check that labels for all classes appear in both train and test data
train_label_counts = recombined_train_df['label'].value_counts()
print("\n Label Counts (Train Data):")
print(train_label_counts)
print("******************************")

# Count occurrences of each label in test dataframe
test_label_counts = recombined_test_df['label'].value_counts()
print("\nLabel Counts (Test Data):")
print(test_label_counts)
```

```
LOADED TRAIN DATA
                                        verse_text  label
0                    o that i were where helen lies      2
1                   and keep my senses straightened      2
2  his song, though very sweet, was low and faint,      3
3                    what's de use o' gittin' mopy,      2
4           she still must keep the locket to allay      2
length: 880

**********************************************************************

LOADED TEST DATA
                              verse_text  label
```

```
0     where the warm life we cannot see--      1
1   nile shall pursue his changeless way:      2
2    in our embraces we again enfold her,      1
3                 of my wit or in my mind      2
4       for the greek must ask elsewhere.      2
length: 221

 Label Counts (Train Data):
label
2    554
0    154
1    133
3     39
Name: count, dtype: int64
*******************************

Label Counts (Test Data):
label
2    139
0     39
1     33
3     10
Name: count, dtype: int64
```

The training set now contains 39/49 = about 80% of total "mixed" data samples (class 3), while test data contains the remaining 20%. This shows that the class is now proportionally spread between the two datasplits.

I will now perform five-fold cross-validation using the training set, by training the Naive-Bayes Classifier on four of the five splits while validating on the remaining split (five times). This process will be repeated for each of the experiments conducted above in the project on the original train-validation-test split. For each experiment/different featureset, the accuracy and macro-average F1-scores will be recorded for each "split", before being summed and averaged across the splits.

At the end of all the experiments, these results will be tabulated to facilitate finding the feature set achieving the highest score for the new training data split. The best-performing techniques will be used to evaluate the performance of the classifier on the test set, to verify if it can generalize to new data. A confusion matrix will then be displayed to show the results of the final feature-extraction method on the performance of Multinomial Naive Bayes on the test set.

[62]:
```
# Extract train and test samples and labels as a list
recombined_train_samples = recombined_train_df['verse_text'].to_list()
recombined_train_labels = recombined_train_df['label'].to_list()
recombined_test_samples = recombined_test_df['verse_text'].to_list()
recombined_test_labels = recombined_test_df['label'].to_list()

# Tokenize the samples in the training and test set
recombined_train_tokens = [word_tokenize(sample) for sample in␣
 ↪recombined_train_samples]
```

```
recombined_test_tokens = [word_tokenize(sample) for sample in␣
↪recombined_test_samples]


# Create a new set of label_names because of the new presence of the "mixed"␣
↪class in the test set
label_names = ['Negative', 'Positive', 'No Impact (neutral)', "Mixed Sentiment"]

##########################################################################################

def cross_validate_train_data_nltkNaiveBayes(samples, labels,␣
↪num_word_features, get_features, k=5):
    """
        A function which applies k-fold cross-validation to the training split␣
↪of the poem dataset,
        and outputs the nltk Multinomial Naive Bayes classifier's mean␣
↪performance scores across the folds, as well
        as the standard deviation of the macro-averaged F1-scores (standard␣
↪deviation) across the folds,
        to ensure consistency of performance.
        Inputs:
            - samples ==> a list-of-lists where each sub-list/sample is a list␣
↪of tokens.
            - labels ==> a list of labels corresponding to each training sample.
            - num_word_features ==> num of word features to use for each␣
↪training-split FreqDist.
            - get_features ==> a function (e.g. "doc_features") that turns the␣
↪token lists into features using the specific number of word features.
            - k ==> an integer representing the number of folds to iterate over␣
↪for k-fold cross-validation
        Outputs:
            - a dictionary containing keys for the mean scores across the␣
↪samples and macro-average F1-score standard deviation
    """
    # Initialize lists of metrics to later calculate the mean
    accuracies = []
    macro_avg_precisions = []
    macro_avg_recalls = []
    macro_avg_f1s = []

    # Use Stratified KFold scikit-learn class with k (nr folds): it outputs␣
↪indices for validation and training folds
    # Shuffle to reduce impact of specific orderings
    SKFGenerator = StratifiedKFold(n_splits=k, shuffle=True, random_state=3)  #␣
↪Use random_state for reproducibility and comparison of results
```

```
    ## Logger to monitor progress
    counter = 1

    # Iterate over the indices to use for each fold
    for train_indices, val_indices in SKFGenerator.split(samples, labels):

        # Create train_set and val_set tuples  each fold using the stratified k␣
↪fold's indices.
        train_set = [(samples[i], labels[i]) for i in train_indices]
        val_set = [(samples[i], labels[i]) for i in val_indices]

        train_tokenlists = [(tokenlist) for (tokenlist, label) in train_set]
        train_vocabulary_list = flatten_list_of_lists(train_tokenlists)
        all_words = nltk.FreqDist(w for w in train_vocabulary_list)
        word_features = list(all_words)[:num_word_features]

        train_featuresets = [(get_features(doc, word_features), label) for␣
↪(doc, label) in train_set]
        val_featuresets = [(get_features(doc, word_features), label) for (doc,␣
↪label) in val_set]

        # Train the Naive Bayes Classifier
        NBclassifier = nltk.NaiveBayesClassifier.train(train_featuresets)

         # Get the true labels and the predicted labels from the classifier
        true = [label for (features, label) in val_featuresets]
        pred = [NBclassifier.classify(features) for (features, label) in␣
↪val_featuresets]

        # Calculate the accuracy for this particular fold.
        acc = accuracy(NBclassifier , val_featuresets)
        accuracies.append(acc)

        # Calculate macro_average precision, recall, and f1-score for this fold
        precision, recall, f1, _ = precision_recall_fscore_support(true, pred,␣
↪average='macro', zero_division=0) # set to 0 to avoid zero-division error
        macro_avg_precisions.append(precision)
        macro_avg_recalls.append(recall)
        macro_avg_f1s.append(f1)
        # Increment counter for logging outputs
        counter += 1

    # Calculate the means for all metrics across the folds
    mean_accuracy = np.mean(accuracies)
    mean_macro_precision = np.mean(macro_avg_precisions)
    mean_macro_recall = np.mean(macro_avg_recalls)
    mean_macro_f1 = np.mean(macro_avg_f1s)
```

```
    # Calculate the standard deviation of macro-average F1 across the folds to␣
↪see if the scores are reasonably similar
    std_macro_f1 = np.std(macro_avg_f1s, ddof=1) # apply Bessel's correction␣
↪for fold std deviation as this is a small sample, not a population

    # Calculate the range of macro f1 scores (to put standard deviation into␣
↪context)
    range_macro_f1 = np.max(macro_avg_f1s) - np.min(macro_avg_f1s)
    print(f"Number of word features: {num_word_features} --- Macro-Avg F1␣
↪standard deviation: {std_macro_f1} --- Macro-Avg F1 Range: {range_macro_f1}")

    # Return the mean scores and standard deviation/range of macro-average F1␣
↪score
    return {
        "mean_accuracy": mean_accuracy,
        "mean_macro_precision": mean_macro_precision,
        "mean_macro_recall": mean_macro_recall,
        "mean_macro_f1": mean_macro_f1,
        "std_macro_f1": std_macro_f1,
        "range_macro_f1": range_macro_f1
    }


# Function to test the performance across different ranges of word features
def test_diff_word_feature_numbers(samples, labels, lower_limit, upper_limit,␣
↪step, get_features, k=5):
    """
        Tests performance of Multinomial Naive Bayes (using stratified k-fold␣
↪cross-validation) on feature sets with different
        numbers of word features.
        Inputs:
            samples = list of token lists representing verse text samples in␣
↪the training set
            labels = list of labels (0-3) for each sample indicating sentiment␣
↪polarity
            lower_limit = lowest number of word features to test
            upper_limit = highest number of word features to test + 1 (due to␣
↪exclusive upper range)
            step = number of steps to take when testing different numbers of␣
↪word features
            get_features = the function to apply to the tokenlists/samples to␣
↪extract specific features (e.g. AFINN sentiment lexicon scores)
        Output:
            word_feature_counts = a numpy array of each number of word features␣
↪that was tested
```

```
            mean_accuracies = a list of the mean accuracy scores achieved using␣
↪cross-validation for each number of word features
            mean_precisions = a list of the mean macro-average precision scores␣
↪achieved using cross-validation for each number of word features
            mean_recalls = a list of the mean macro-average recall scores␣
↪achieved using cross-validation for each number of word features
            mean_f1s = a list of the mean macro-average F1-scores achieved␣
↪using cross-validation for each number of word features
            f1_td_devs = a list of the macro-average F1-score standard␣
↪deviations across each cross-validation fold for each number of word features
            f1_ranges = a list of the macro-average F1-score ranges across each␣
↪cross-validation fold for each number of word features

    """
    word_feature_counts = np.arange(lower_limit, upper_limit, step)
    # Store mean metrics for each number of word features here
    mean_accuracies = []
    mean_precisions = []
    mean_recalls = []
    mean_f1s = []
    f1_std_devs = []
    f1_ranges = []
    for word_count in word_feature_counts:
        # Call the cross-validation function to get the dictionary of the mean␣
↪scores across the folds
        metrics_dict = cross_validate_train_data_nltkNaiveBayes(samples,␣
↪labels, word_count, get_features, k=k)
        # Append mean scores or standard deviation/range for the metrics to the␣
↪lists created above
        mean_accuracies.append(metrics_dict["mean_accuracy"])
        mean_precisions.append(metrics_dict["mean_macro_precision"])
        mean_recalls.append(metrics_dict["mean_macro_recall"])
        mean_f1s.append(metrics_dict["mean_macro_f1"])
        f1_std_devs.append(metrics_dict["std_macro_f1"])
        f1_ranges.append(metrics_dict["range_macro_f1"])
    return word_feature_counts, mean_accuracies, mean_precisions, mean_recalls,␣
↪mean_f1s, f1_std_devs, f1_ranges

def plot_impacts_of_different_word_counts(word_feature_counts, mean_accuracies,␣
↪ mean_f1s, title):
    """
        Plots the number of word features used against the mean accuracy and␣
↪mean macro-average F1-score achieved
        using cross-validation.
        Inputs:
```

```
            word_feature_counts = an array of the numbers of word features that␣
↪were used
            mean_accuracies = mean accuracies achieved for each number of word␣
↪features
            mean_f1s = mean macro-average F1-scores achieved for each number of␣
↪word features
            title = string for the title of the chart
    """
    # Set plot size
    plt.figure(figsize=(12, 6))
    # Plot number of word features against mean accuracies
    plt.plot(word_feature_counts, mean_accuracies, color='red', marker='o', ␣
↪label='Mean Accuracy')
    # Plot number of word features against mean macro-average F1-scores
    plt.plot(word_feature_counts, mean_f1s, color='blue', marker='o',␣
↪label='Mean Macro-avg F1 Score')
    # Add labels and title
    plt.xlabel('Number of Word Features')
    plt.ylabel('Mean Scores (Cross-Validation)')
    plt.title(title)
    # Add legend to explain colours for accuracy and F1-scores
    plt.legend()
    plt.xticks(word_feature_counts)
    # Label the chart

    # Find the index of the first maximum accuracy and F1 score
    max_accuracy_index = np.argmax(mean_accuracies)
    max_f1_score_index = np.argmax(mean_f1s)

    # Get the corresponding values for max accuracy and F1 score
    max_accuracy = mean_accuracies[max_accuracy_index]
    max_f1_score = mean_f1s[max_f1_score_index]

    # Annotate the point where the first max accuracy occurs
    plt.annotate(f"Max: {max_accuracy:.2f}",␣
↪xy=(word_feature_counts[max_accuracy_index], max_accuracy),
                 xytext=(word_feature_counts[max_accuracy_index], max_accuracy␣
↪- 0.05),
                 arrowprops=dict(facecolor='red', shrink=0.05), fontsize=10,␣
↪color='red')

    # Annotate the point where the first max F1 score occurs
    plt.annotate(f"Max: {max_f1_score:.2f}",␣
↪xy=(word_feature_counts[max_f1_score_index], max_f1_score),
                 xytext=(word_feature_counts[max_f1_score_index], max_f1_score␣
↪+ 0.05),
```

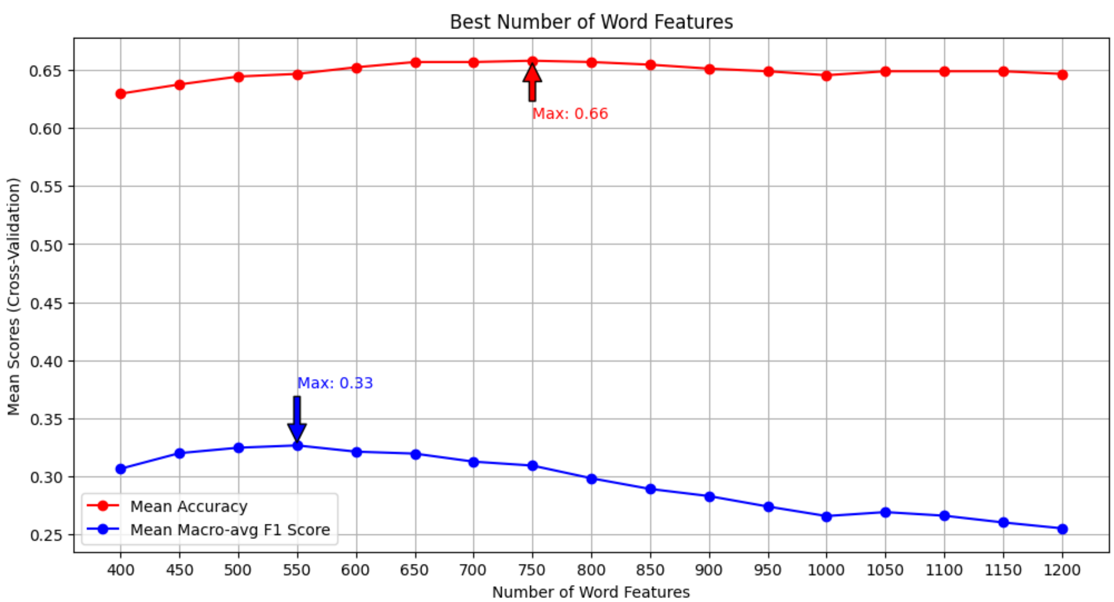
```
                 arrowprops=dict(facecolor='blue', shrink=0.05), fontsize=10,␣
↪color='blue')

    # Display the plot
    plt.grid(True)
    plt.show()
```

2.4.1 Calculating the Performance of the Multinomial Naive Bayes Classifier on the Baseline Features (Simple Tokenization and Top N Word Features) using the Recombined Training Data and Stratified 5-Fold Cross-Validation

[11]:
```
# Test between 400 and 1200 different numbers of word features
word_feature_counts, mean_accuracies, mean_precisions, mean_recalls, mean_f1s,␣
↪f1_std_devs, f1_ranges = test_diff_word_feature_numbers(
                                                                                ␣
↪                                   recombined_train_tokens,
                                                                                ␣
↪                                   recombined_train_labels,
                                                                                ␣
↪                                   # test from 400 to 1200 word features
                                                                                ␣
↪                                   400, 1201, 50, doc_features,
                                                                                ␣
↪                                   k=5
                                                                                ␣
↪                            )

# Plot results to get best number of word features to use
plot_impacts_of_different_word_counts(word_feature_counts, mean_accuracies,␣
↪mean_f1s, "Best Number of Word Features")
```

```
Number of word features: 400 --- Macro-Avg F1 standard deviation:
0.006757800921609499 --- Macro-Avg F1 Range: 0.016934696080674272
Number of word features: 450 --- Macro-Avg F1 standard deviation:
0.021323425504518864 --- Macro-Avg F1 Range: 0.05035854099784165
Number of word features: 500 --- Macro-Avg F1 standard deviation:
0.023667133260398328 --- Macro-Avg F1 Range: 0.062041521994083415
Number of word features: 550 --- Macro-Avg F1 standard deviation:
0.03177718492601444 --- Macro-Avg F1 Range: 0.0822178147725161
Number of word features: 600 --- Macro-Avg F1 standard deviation:
0.014293456096563186 --- Macro-Avg F1 Range: 0.03828551844932793
Number of word features: 650 --- Macro-Avg F1 standard deviation:
0.012482918849056918 --- Macro-Avg F1 Range: 0.03005436510787901
Number of word features: 700 --- Macro-Avg F1 standard deviation:
0.01163634094096908 --- Macro-Avg F1 Range: 0.02792634195073218
Number of word features: 750 --- Macro-Avg F1 standard deviation:
```

```
0.020089355506307017 --- Macro-Avg F1 Range: 0.0479060839556294
Number of word features: 800 --- Macro-Avg F1 standard deviation:
0.020073755348763142 --- Macro-Avg F1 Range: 0.0499844921338824
Number of word features: 850 --- Macro-Avg F1 standard deviation:
0.014915851821718567 --- Macro-Avg F1 Range: 0.03027557085951249
Number of word features: 900 --- Macro-Avg F1 standard deviation:
0.017307009997197857 --- Macro-Avg F1 Range: 0.04097095653683208
Number of word features: 950 --- Macro-Avg F1 standard deviation:
0.0247469430026827 --- Macro-Avg F1 Range: 0.056672789296209114
Number of word features: 1000 --- Macro-Avg F1 standard deviation:
0.021357865694351663 --- Macro-Avg F1 Range: 0.05597871292256884
Number of word features: 1050 --- Macro-Avg F1 standard deviation:
0.0225205423729173 --- Macro-Avg F1 Range: 0.062102515522091783
Number of word features: 1100 --- Macro-Avg F1 standard deviation:
0.031074052173442036 --- Macro-Avg F1 Range: 0.08680711075238007
Number of word features: 1150 --- Macro-Avg F1 standard deviation:
0.03504359620233258 --- Macro-Avg F1 Range: 0.08530165962818848
Number of word features: 1200 --- Macro-Avg F1 standard deviation:
0.029926979920826078 --- Macro-Avg F1 Range: 0.07311262440711663
```



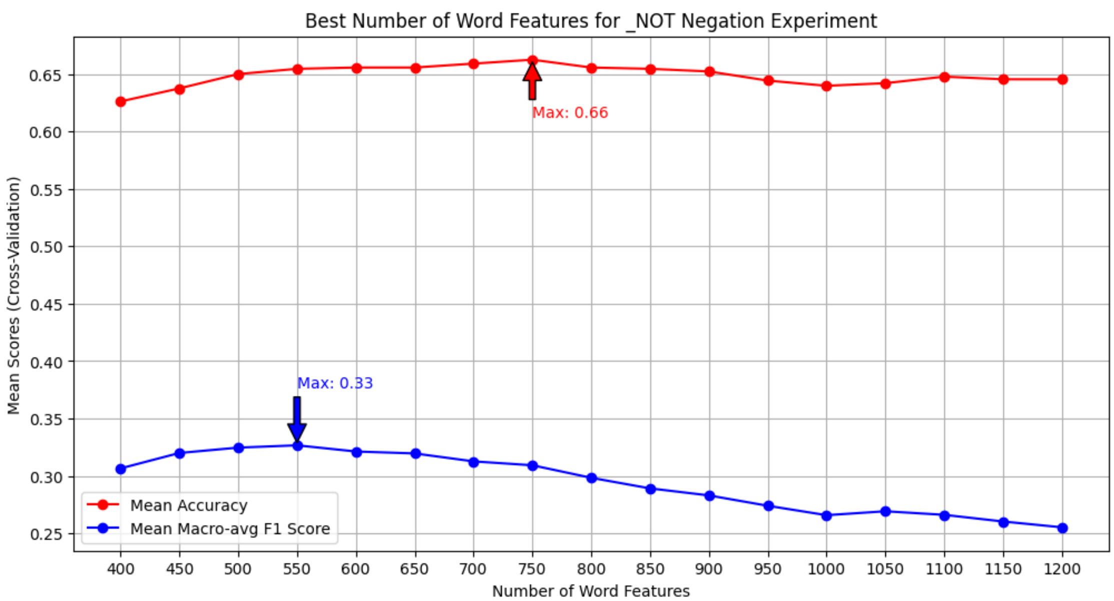
Note: a table containing all of these results for each experiment is included under Experiment 9.

2.4.2 Feature-Engineering Experiment #1 using Five-Fold Cross-Validation: _NOT Negation Features following Jurafsky & Martin Technique (2024)

[96]:
```python
## Use the simple_negation_features function to generate tokens with _NOT
 ↪following negation cues defined above
neg_recombined_train_tokens = [simple_negation_features(token_list) for
 ↪token_list in recombined_train_tokens]

# Repeat testing different numbers of word features
neg_word_feature_counts, neg_mean_accuracies, neg_mean_precisions,
 ↪neg_mean_recalls, neg_mean_f1s, neg_f1_std_devs, neg_f1_ranges =
 ↪test_diff_word_feature_numbers(

 ↪                              neg_recombined_train_tokens,

 ↪                              recombined_train_labels,

 ↪                              # test from 400 to 1200 word features

 ↪                              400, 1201, 50, doc_features,

 ↪                              k=5

 ↪                       )

plot_impacts_of_different_word_counts(neg_word_feature_counts,
 ↪neg_mean_accuracies, mean_f1s, "Best Number of Word Features for _NOT
 ↪Negation Experiment")
```

```
Number of word features: 400 --- Macro-Avg F1 standard deviation:
0.004159753306759044 --- Macro-Avg F1 Range: 0.01004694865732203
Number of word features: 450 --- Macro-Avg F1 standard deviation:
0.018403890667453574 --- Macro-Avg F1 Range: 0.04275071447365386
Number of word features: 500 --- Macro-Avg F1 standard deviation:
0.023934198869034265 --- Macro-Avg F1 Range: 0.05829928166256798
Number of word features: 550 --- Macro-Avg F1 standard deviation:
0.02361813567226613 --- Macro-Avg F1 Range: 0.06127592588266745
Number of word features: 600 --- Macro-Avg F1 standard deviation:
0.012312698670245644 --- Macro-Avg F1 Range: 0.03245560621181909
Number of word features: 650 --- Macro-Avg F1 standard deviation:
0.012458589491901581 --- Macro-Avg F1 Range: 0.02711421745471032
Number of word features: 700 --- Macro-Avg F1 standard deviation:
0.010945750177003355 --- Macro-Avg F1 Range: 0.029106056810504433
Number of word features: 750 --- Macro-Avg F1 standard deviation:
0.007019485272788465 --- Macro-Avg F1 Range: 0.016679114622547453
Number of word features: 800 --- Macro-Avg F1 standard deviation:
0.018747708193432706 --- Macro-Avg F1 Range: 0.04877086510760886
```

```
Number of word features: 850 --- Macro-Avg F1 standard deviation:
0.02066641378202711 --- Macro-Avg F1 Range: 0.04947125126646401
Number of word features: 900 --- Macro-Avg F1 standard deviation:
0.017193254249069295 --- Macro-Avg F1 Range: 0.041394315946881644
Number of word features: 950 --- Macro-Avg F1 standard deviation:
0.027231232854643486 --- Macro-Avg F1 Range: 0.056494705019553865
Number of word features: 1000 --- Macro-Avg F1 standard deviation:
0.030537073295877896 --- Macro-Avg F1 Range: 0.07357278827567448
Number of word features: 1050 --- Macro-Avg F1 standard deviation:
0.03029022570781339 --- Macro-Avg F1 Range: 0.07357278827567448
Number of word features: 1100 --- Macro-Avg F1 standard deviation:
0.03114344292527054 --- Macro-Avg F1 Range: 0.08703566168467133
Number of word features: 1150 --- Macro-Avg F1 standard deviation:
0.032257319433491465 --- Macro-Avg F1 Range: 0.08703566168467133
Number of word features: 1200 --- Macro-Avg F1 standard deviation:
0.03532776577376125 --- Macro-Avg F1 Range: 0.08537566598402846
```



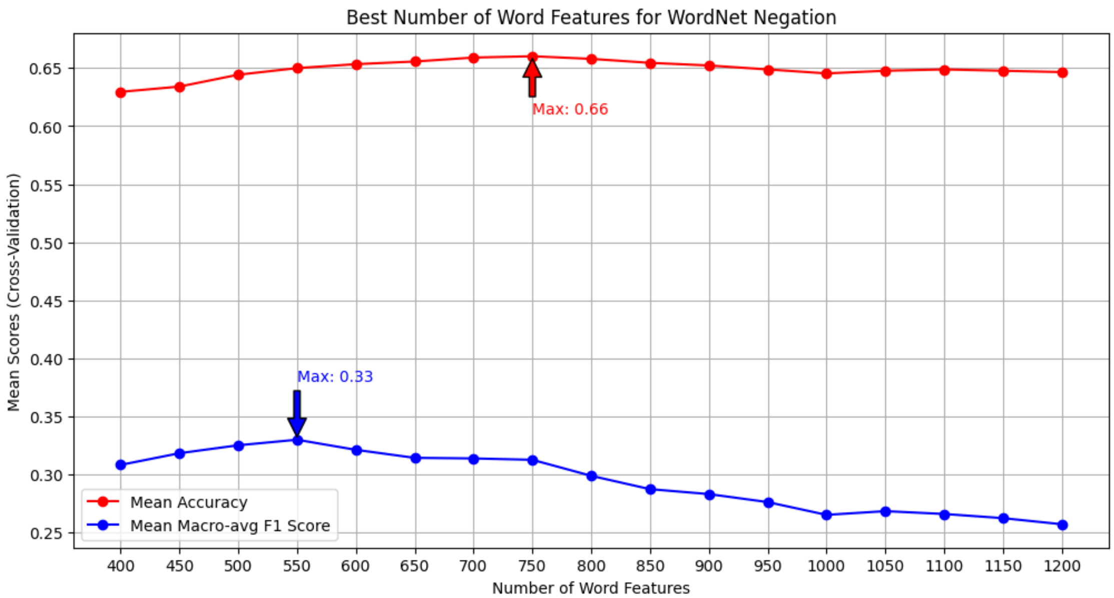
2.4.3 Feature-Engineering Experiment #2 using Five-Fold Cross-Validation: WordNet Negation Features following Utkarsh Lal (2022) Negation Algorithm

[98]:
```
## Use the handle_negation_with_wordnet function to generate tokens with _NOT␣
↪following negation cues
w_neg_recombined_train_tokens = [handle_negation_with_wordnet(token_list,␣
↪negation_patterns) for token_list in recombined_train_tokens]
```

```
w_neg_word_feature_counts, w_neg_mean_accuracies, w_neg_mean_precisions,␣
↪w_neg_mean_recalls, w_neg_mean_f1s, w_neg_f1_std_devs, w_neg_f1_ranges =␣
↪test_diff_word_feature_numbers(
                                                                            ␣
↪                                w_neg_recombined_train_tokens,
                                                                            ␣
↪                                recombined_train_labels,
                                                                            ␣
↪                                # test from 400 to 1200 word features
                                                                            ␣
↪                                400, 1201, 50, doc_features,
                                                                            ␣
↪                                k=5
                                                                            ␣
↪                         )

plot_impacts_of_different_word_counts(w_neg_word_feature_counts,␣
↪w_neg_mean_accuracies, w_neg_mean_f1s,
                                      "Best Number of Word Features for WordNet␣
↪Negation")
```

```
Number of word features: 400 --- Macro-Avg F1 standard deviation:
0.01033351113115565 --- Macro-Avg F1 Range: 0.027595995835560272
Number of word features: 450 --- Macro-Avg F1 standard deviation:
0.020146007537478902 --- Macro-Avg F1 Range: 0.04633326336429133
Number of word features: 500 --- Macro-Avg F1 standard deviation:
0.02192987158121347 --- Macro-Avg F1 Range: 0.05397094903432231
Number of word features: 550 --- Macro-Avg F1 standard deviation:
0.02893198270298793 --- Macro-Avg F1 Range: 0.07543619836995863
Number of word features: 600 --- Macro-Avg F1 standard deviation:
0.011734252519241139 --- Macro-Avg F1 Range: 0.028858390692579727
Number of word features: 650 --- Macro-Avg F1 standard deviation:
0.012516875576322231 --- Macro-Avg F1 Range: 0.024645615148630062
Number of word features: 700 --- Macro-Avg F1 standard deviation:
0.012329635686183792 --- Macro-Avg F1 Range: 0.027544783647724835
Number of word features: 750 --- Macro-Avg F1 standard deviation:
0.020429090817660464 --- Macro-Avg F1 Range: 0.049776055457411206
Number of word features: 800 --- Macro-Avg F1 standard deviation:
0.020117815545819102 --- Macro-Avg F1 Range: 0.051269345043232994
Number of word features: 850 --- Macro-Avg F1 standard deviation:
0.013940517925117041 --- Macro-Avg F1 Range: 0.03333870325718152
Number of word features: 900 --- Macro-Avg F1 standard deviation:
0.0167364032619493 --- Macro-Avg F1 Range: 0.03777178133250508
Number of word features: 950 --- Macro-Avg F1 standard deviation:
0.022278233424085626 --- Macro-Avg F1 Range: 0.05314551666654713
Number of word features: 1000 --- Macro-Avg F1 standard deviation:
0.025317898815145092 --- Macro-Avg F1 Range: 0.06819928010740733
```

```
Number of word features: 1050 --- Macro-Avg F1 standard deviation:
0.026205439366132247 --- Macro-Avg F1 Range: 0.05965285610446902
Number of word features: 1100 --- Macro-Avg F1 standard deviation:
0.03294061251310053 --- Macro-Avg F1 Range: 0.08410302043880075
Number of word features: 1150 --- Macro-Avg F1 standard deviation:
0.036111117561516644 --- Macro-Avg F1 Range: 0.08410302043880075
Number of word features: 1200 --- Macro-Avg F1 standard deviation:
0.033560733278419075 --- Macro-Avg F1 Range: 0.08259756931460915
```



2.4.4 Feature-Engineering Experiment #3 using Five-Fold Cross-Validation: WordNet Negation Features and TF-IDF

[99]:
```
# Define a new cross-validation function to work with TF-IDF and scikit-learn␣
↪Multinomial Naive Bayes classifier instead
def cross_validate_train_data_TF_IDF(samples, labels, k=5):
    """
        Description: A function which applies k-fold cross-validation to the␣
↪training split of the poem dataset,
        applies TF-IDF to the word vectors and outputs the scikit-learn␣
↪Multinomial Naive Bayes classifier's mean
        performance scores across the folds, as well as the variability in␣
↪f1-scores (standard deviation) across the folds.
        Inputs:
            - samples ==> a list-of-lists where each sub-list/sample is a list␣
↪of tokens.
            - labels ==> a list of labels corresponding to each training sample.
```

```python
        - k ==> an integer representing the number of folds to iterate over␣
↪for k-fold cross-validation
    Outputs:
        - a dictionary containing keys for the average accuracy,␣
↪macro-average precision/recall/F1-scores
          and f1 standard deviation across the samples
    """
    # Initialize lists of metrics
    accuracies = []
    macro_avg_precisions = []
    macro_avg_recalls = []
    macro_avg_f1s = []

    # Use Stratified KFold scikit-learn class with k (nr folds): it outputs␣
↪indices
    # Shuffle to reduce impact of specific orderings of the samples
    SKFGenerator = StratifiedKFold(n_splits=k, shuffle=True, random_state=3)  #␣
↪Use random_state for reproducibility and comparison of results

    # logger for displaying the progress made
    counter = 1

    # Iterate over the folds using the outputted indices by StratifiedKFold for␣
↪this dataset
    for train_indices, val_indices in SKFGenerator.split(samples, labels):

        # Create train_set and val_set tuple (sample-label) lists for each fold␣
↪using the stratified k fold's indices.
        train_set = [(samples[i], labels[i]) for i in train_indices]
        val_set = [(samples[i], labels[i]) for i in val_indices]
        # Convert the token lists to strings
        train_texts = [' '.join(tokens) for tokens, label in train_set]
        val_texts = [' '.join(tokens) for tokens, label in val_set]
        # Extract the labels for each text
        train_labels = [label for tokens, label in train_set]
        val_labels = [label for tokens, label in val_set]
        # Initialize TF-IDF vectorizer
        tfidf_vectorizer = TfidfVectorizer()
        # Fit and transform the training data
        train_tfidf = tfidf_vectorizer.fit_transform(train_texts)
        # Transform the validation data
        val_tfidf = tfidf_vectorizer.transform(val_texts)
        NBclassifier = MultinomialNB()
        NBclassifier.fit(train_tfidf, train_labels)
        # Predict labels
        val_predictions = NBclassifier.predict(val_tfidf)
```

```
        # Get the true labels and the predicted labels from the classifier

        # Calculate the accuracy for this particular fold
        acc = accuracy_score(val_labels, val_predictions)
        accuracies.append(acc)

        # Calculate macro-average precision, recall, and f1-score for this fold
        precision, recall, f1, _ = precision_recall_fscore_support(val_labels,␣
↪val_predictions, average='macro', zero_division=0)
        macro_avg_precisions.append(precision)
        macro_avg_recalls.append(recall)
        macro_avg_f1s.append(f1)

        # Increment counter for logging outputs
        counter += 1

    # Calculate the mean for all metrics across the folds.
    mean_accuracy = np.mean(accuracies)
    mean_macro_precision = np.mean(macro_avg_precisions)
    mean_macro_recall = np.mean(macro_avg_recalls)
    mean_macro_f1 = np.mean(macro_avg_f1s)

    # Calculate the standard deviation of f1 across the folds
    std_macro_f1 = np.std(macro_avg_f1s, ddof=1) # apply Bessel's correction␣
↪for fold std deviation as this is a small sample, not a pop.
    # Calculate the range of macro f1 scores (to put std into context)
    range_macro_f1 = np.max(macro_avg_f1s) - np.min(macro_avg_f1s)
    print(f"Macro-Avg F1 standard deviation: {std_macro_f1} --- Macro-Avg F1␣
↪Range: {range_macro_f1}")

    return {
        "mean_accuracy": mean_accuracy,
        "mean_macro_precision": mean_macro_precision,
        "mean_macro_recall": mean_macro_recall,
        "mean_macro_f1": mean_macro_f1,
        "std_macro_f1": std_macro_f1,
        "range_macro_f1": range_macro_f1
    }
```

[100]:
```
tf_idf_metrics_dict =␣
↪cross_validate_train_data_TF_IDF(w_neg_recombined_train_tokens,␣
↪recombined_train_labels)
tf_idf_mean_accuracy_score = tf_idf_metrics_dict["mean_accuracy"]
tf_idf_mean_f1_score = tf_idf_metrics_dict["mean_macro_f1"]
print(f"MEAN ACCURACY: {tf_idf_mean_accuracy_score}\nMEAN MACRO-AVG F1 SCORE:␣
↪{tf_idf_mean_f1_score}")
```

```
Macro-Avg F1 standard deviation: 0.008716353655225915 --- Macro-Avg F1 Range:
0.020266770266770234
MEAN ACCURACY: 0.6306818181818182
MEAN MACRO-AVG F1 SCORE: 0.19700430554089088
```

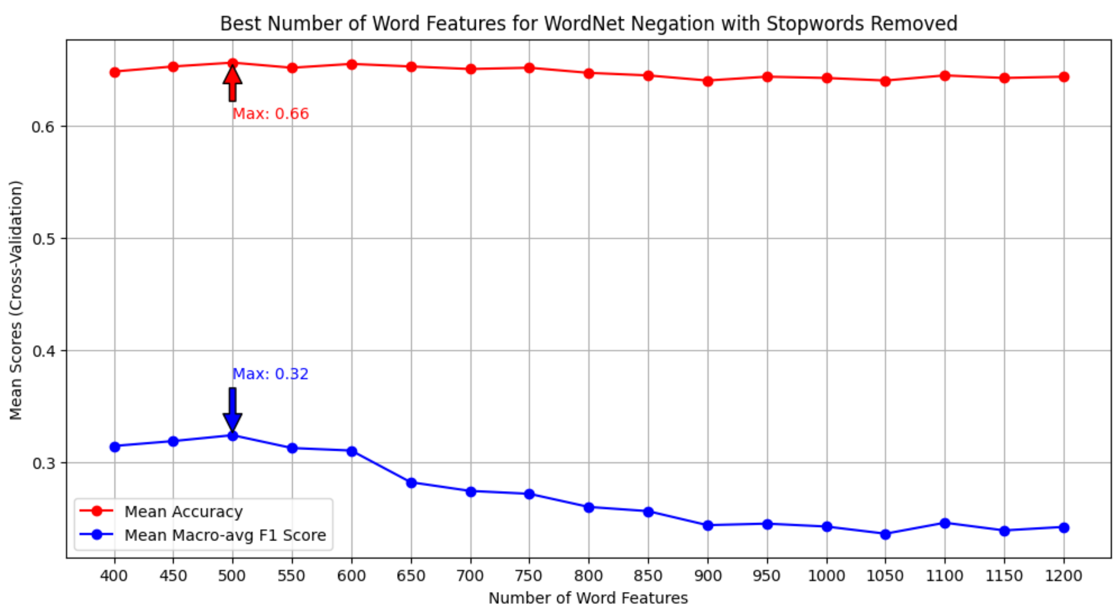
This experiment does not have a graph for finding optimal number of word features. This is because TF-IDF requires inputs constructed by inputtinge entire strings into the TF-IDF Vectorizer class instead.

2.4.5 Feature-Engineering Experiment #4 using Five-Fold Cross-Validation: WordNet Negation Features and Stopword Removal

[101]:
```
# Store tokens for training set after applying remove_stopwords_from_tokens to␣
↪each tokenset.
w_neg_recombined_train_tokens_no_stopwords = [
    remove_stopwords_from_tokens(tokens, non_negated_english_stopwords)
    for tokens in w_neg_recombined_train_tokens
]

# Test different numbers of features for tokens with the stopwords removed
(
    w_neg_word_feature_counts_no_stopwords, w_neg_mean_accuracies_no_stopwords,
    w_neg_mean_precisions_no_stopwords, w_neg_mean_recalls_no_stopwords,
    w_neg_mean_f1s_no_stopwords, w_neg_f1_std_devs_no_stopwords,␣
↪w_neg_f1_ranges_no_stopwords
) = test_diff_word_feature_numbers(
    w_neg_recombined_train_tokens_no_stopwords,
    recombined_train_labels,
    400,  # Start testing from 400 word features
    1201,  # End testing at 1200 word features (1201 is exclusive)
    50,  # Test every 50 word features
    doc_features,
    k=5
)


plot_impacts_of_different_word_counts(w_neg_word_feature_counts_no_stopwords,␣
↪w_neg_mean_accuracies_no_stopwords, w_neg_mean_f1s_no_stopwords,
                                      "Best Number of Word Features for WordNet␣
↪Negation with Stopwords Removed")
```

```
Number of word features: 400 --- Macro-Avg F1 standard deviation:
0.03427908734481318 --- Macro-Avg F1 Range: 0.07024391446033235
Number of word features: 450 --- Macro-Avg F1 standard deviation:
0.04533744402913512 --- Macro-Avg F1 Range: 0.12409536821786449
Number of word features: 500 --- Macro-Avg F1 standard deviation:
0.04008072434470242 --- Macro-Avg F1 Range: 0.11097869706505353
Number of word features: 550 --- Macro-Avg F1 standard deviation:
```

```
0.042312300842311974 --- Macro-Avg F1 Range: 0.1127297423710838
Number of word features: 600 --- Macro-Avg F1 standard deviation:
0.03406268974328833 --- Macro-Avg F1 Range: 0.09122988122988124
Number of word features: 650 --- Macro-Avg F1 standard deviation:
0.01927605126022682 --- Macro-Avg F1 Range: 0.05184014856773145
Number of word features: 700 --- Macro-Avg F1 standard deviation:
0.014615556338771358 --- Macro-Avg F1 Range: 0.040550876847939565
Number of word features: 750 --- Macro-Avg F1 standard deviation:
0.015939310067425643 --- Macro-Avg F1 Range: 0.03957038498501503
Number of word features: 800 --- Macro-Avg F1 standard deviation:
0.02133648167924917 --- Macro-Avg F1 Range: 0.05748263230099793
Number of word features: 850 --- Macro-Avg F1 standard deviation:
0.027934555567158997 --- Macro-Avg F1 Range: 0.07441318401506808
Number of word features: 900 --- Macro-Avg F1 standard deviation:
0.03338501578622686 --- Macro-Avg F1 Range: 0.09165456332541291
Number of word features: 950 --- Macro-Avg F1 standard deviation:
0.029893155013277477 --- Macro-Avg F1 Range: 0.07588793429568655
Number of word features: 1000 --- Macro-Avg F1 standard deviation:
0.027674807160052738 --- Macro-Avg F1 Range: 0.0646290588976485
Number of word features: 1050 --- Macro-Avg F1 standard deviation:
0.026789709462013794 --- Macro-Avg F1 Range: 0.06493372219178675
Number of word features: 1100 --- Macro-Avg F1 standard deviation:
0.01778377208577465 --- Macro-Avg F1 Range: 0.041936590586871075
Number of word features: 1150 --- Macro-Avg F1 standard deviation:
0.02393905135453315 --- Macro-Avg F1 Range: 0.05881645045346115
Number of word features: 1200 --- Macro-Avg F1 standard deviation:
0.023584420409402576 --- Macro-Avg F1 Range: 0.0591533777157692
```

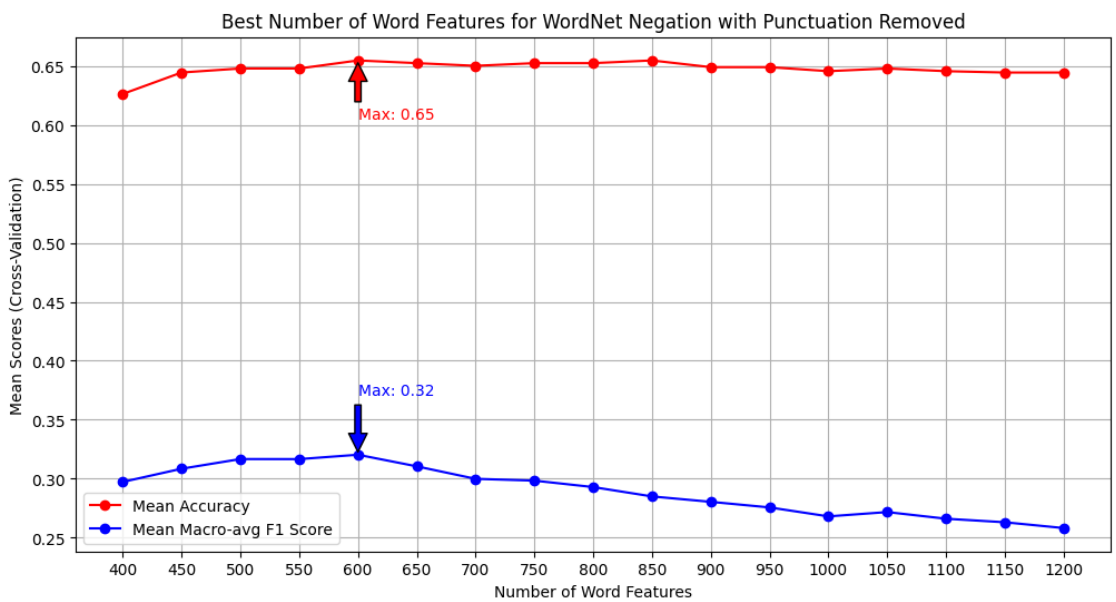


2.4.6 Feature-Engineering Experiment #5 using Five-Fold Cross-Validation: WordNet Negation Features and Punctuation Removal

[102]:
```
# Store tokens for training set after applying remove_punctuation to each␣
↪tokenset.
w_neg_recombined_train_tokens_no_punctuation = [
    remove_punctuation(tokens)
    for tokens in w_neg_recombined_train_tokens
]

# Run the comparison fnuction for different counts of most-common word features
(
    w_neg_word_feature_counts_no_punct,
    w_neg_mean_accuracies_no_punct,
    w_neg_mean_precisions_no_punct,
    w_neg_mean_recalls_no_punct,
    w_neg_mean_f1s_no_punct,
    w_neg_f1_std_devs_no_punct,
    w_neg_f1_ranges_no_punct
) = test_diff_word_feature_numbers(
    w_neg_recombined_train_tokens_no_punctuation,
    recombined_train_labels,
    400,  # Start testing from 400 word features
    1201,  # End testing at 1200 word features (1201 is exclusive)
    50,  # Test every 50 word features
    doc_features,
    k=5
)


plot_impacts_of_different_word_counts(w_neg_word_feature_counts_no_punct,␣
↪w_neg_mean_accuracies_no_punct, w_neg_mean_f1s_no_punct,
                                      "Best Number of Word Features for WordNet␣
↪Negation with Punctuation Removed")
```

```
Number of word features: 400 --- Macro-Avg F1 standard deviation:
0.01572824366574214 --- Macro-Avg F1 Range: 0.03601163198314011
Number of word features: 450 --- Macro-Avg F1 standard deviation:
0.016639921290850523 --- Macro-Avg F1 Range: 0.04251813149982492
Number of word features: 500 --- Macro-Avg F1 standard deviation:
0.026264077242224532 --- Macro-Avg F1 Range: 0.06674444735178547
Number of word features: 550 --- Macro-Avg F1 standard deviation:
0.02986472116257937 --- Macro-Avg F1 Range: 0.08329593746820829
Number of word features: 600 --- Macro-Avg F1 standard deviation:
0.03195105668620066 --- Macro-Avg F1 Range: 0.08784641672317012
Number of word features: 650 --- Macro-Avg F1 standard deviation:
0.02317445175869598 --- Macro-Avg F1 Range: 0.056057377938338926
Number of word features: 700 --- Macro-Avg F1 standard deviation:
```

```
0.019625282842951855 --- Macro-Avg F1 Range: 0.04948849008914674
Number of word features: 750 --- Macro-Avg F1 standard deviation:
0.02615251290725541 --- Macro-Avg F1 Range: 0.0646226298161145
Number of word features: 800 --- Macro-Avg F1 standard deviation:
0.026770429338077068 --- Macro-Avg F1 Range: 0.07048720193413283
Number of word features: 850 --- Macro-Avg F1 standard deviation:
0.018034262737691015 --- Macro-Avg F1 Range: 0.044416786503860606
Number of word features: 900 --- Macro-Avg F1 standard deviation:
0.022495430119526018 --- Macro-Avg F1 Range: 0.05287948647995014
Number of word features: 950 --- Macro-Avg F1 standard deviation:
0.028609506930738333 --- Macro-Avg F1 Range: 0.07058756579883341
Number of word features: 1000 --- Macro-Avg F1 standard deviation:
0.03175937581934173 --- Macro-Avg F1 Range: 0.07060530309737195
Number of word features: 1050 --- Macro-Avg F1 standard deviation:
0.026823829975766947 --- Macro-Avg F1 Range: 0.0679774828077035
Number of word features: 1100 --- Macro-Avg F1 standard deviation:
0.023871591719188304 --- Macro-Avg F1 Range: 0.05507882008885534
Number of word features: 1150 --- Macro-Avg F1 standard deviation:
0.022811551298736087 --- Macro-Avg F1 Range: 0.05732271542229461
Number of word features: 1200 --- Macro-Avg F1 standard deviation:
0.023550685058623535 --- Macro-Avg F1 Range: 0.05633350955824376
```

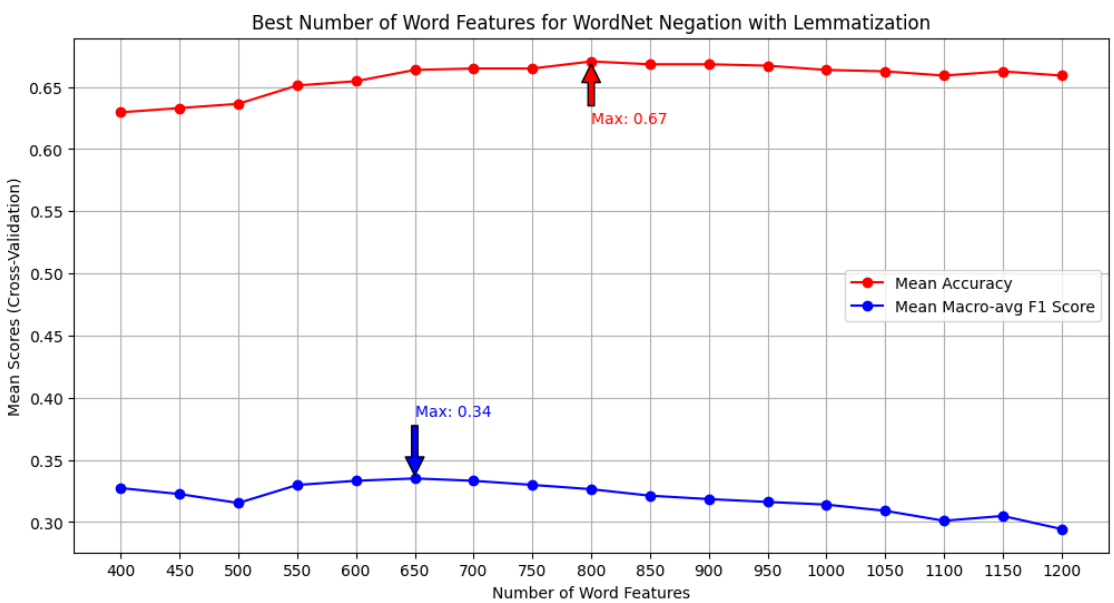


2.4.7 Feature-Engineering Experiment #6 using Five-Fold Cross-Validation: WordNet Negation Features and Lemmatization

[103]:
```
# Instantiate lemmatizer
lemmatizer = WordNetLemmatizer()

# Get the tokens after applying the WordNetLemmatizer lemmatization function
w_neg_recombined_train_tokens_lemmas = [
    lemmatize_tokens(lemmatizer, tokens)
    for tokens in w_neg_recombined_train_tokens
]

# Store the results for each number of word features used
(
    w_neg_word_feature_counts_lemmas,
    w_neg_mean_accuracies_lemmas,
    w_neg_mean_precisions_lemmas,
    w_neg_mean_recalls_lemmas,
    w_neg_mean_f1s_lemmas,
    w_neg_f1_std_devs_lemmas,
    w_neg_f1_ranges_lemmas
) = test_diff_word_feature_numbers(
    w_neg_recombined_train_tokens_lemmas,
    recombined_train_labels,
    400,  # Start testing from 400 word features
    1201,  # End testing at 1200 word features (1201 is exclusive)
    50,  # Test every 50 word features
    doc_features,
    k=5
)


plot_impacts_of_different_word_counts(w_neg_word_feature_counts_lemmas,␣
↪w_neg_mean_accuracies_lemmas, w_neg_mean_f1s_lemmas,
                                      "Best Number of Word Features for WordNet␣
↪Negation with Lemmatization")
```

```
Number of word features: 400 --- Macro-Avg F1 standard deviation:
0.042297545532571555 --- Macro-Avg F1 Range: 0.11527580509970947
Number of word features: 450 --- Macro-Avg F1 standard deviation:
0.04952747023727463 --- Macro-Avg F1 Range: 0.13870573398366937
Number of word features: 500 --- Macro-Avg F1 standard deviation:
0.023911345096130788 --- Macro-Avg F1 Range: 0.06054746556956281
Number of word features: 550 --- Macro-Avg F1 standard deviation:
0.01759032102882962 --- Macro-Avg F1 Range: 0.04819453922764749
Number of word features: 600 --- Macro-Avg F1 standard deviation:
0.012467202995294866 --- Macro-Avg F1 Range: 0.02927136251066409
Number of word features: 650 --- Macro-Avg F1 standard deviation:
```

```
0.014721447651263883 --- Macro-Avg F1 Range: 0.033840375015528334
Number of word features: 700 --- Macro-Avg F1 standard deviation:
0.01501877528462517 --- Macro-Avg F1 Range: 0.03664450854968848
Number of word features: 750 --- Macro-Avg F1 standard deviation:
0.014220589412950038 --- Macro-Avg F1 Range: 0.033626557299441084
Number of word features: 800 --- Macro-Avg F1 standard deviation:
0.026000173267966116 --- Macro-Avg F1 Range: 0.06177802103923591
Number of word features: 850 --- Macro-Avg F1 standard deviation:
0.023272189725427144 --- Macro-Avg F1 Range: 0.05313861655773422
Number of word features: 900 --- Macro-Avg F1 standard deviation:
0.030120691361582547 --- Macro-Avg F1 Range: 0.07757648953301133
Number of word features: 950 --- Macro-Avg F1 standard deviation:
0.029970559192850313 --- Macro-Avg F1 Range: 0.07829589517315155
Number of word features: 1000 --- Macro-Avg F1 standard deviation:
0.031081237062695655 --- Macro-Avg F1 Range: 0.08509898777869684
Number of word features: 1050 --- Macro-Avg F1 standard deviation:
0.03177886062735424 --- Macro-Avg F1 Range: 0.0849790813811469
Number of word features: 1100 --- Macro-Avg F1 standard deviation:
0.037380874876526214 --- Macro-Avg F1 Range: 0.100482308463125
Number of word features: 1150 --- Macro-Avg F1 standard deviation:
0.041100820930117236 --- Macro-Avg F1 Range: 0.10964631556386881
Number of word features: 1200 --- Macro-Avg F1 standard deviation:
0.03829505733043279 --- Macro-Avg F1 Range: 0.10661436767304996
```



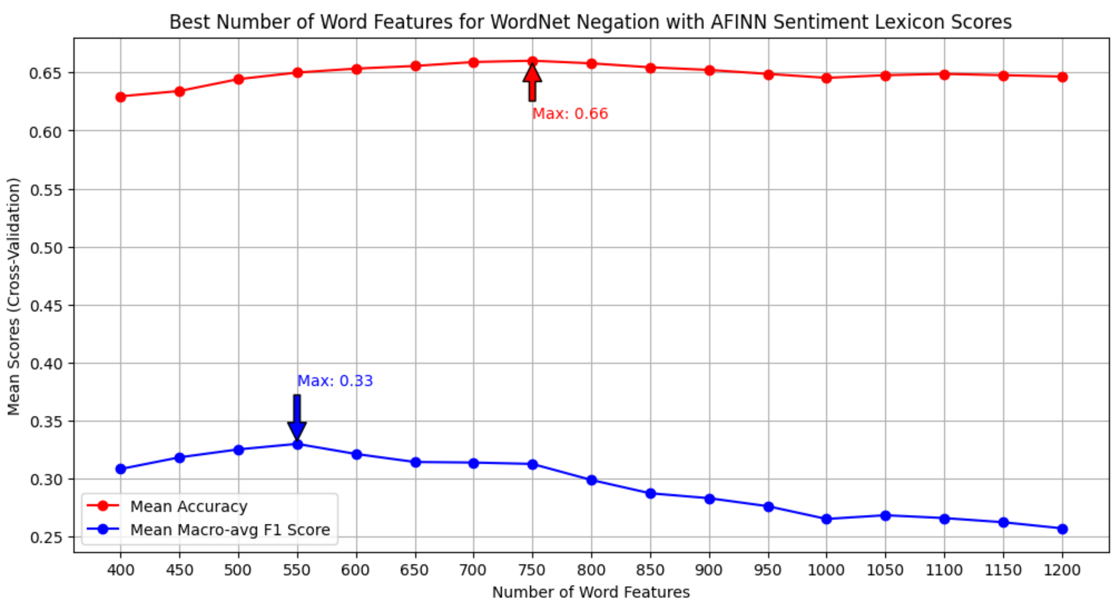
2.4.8 Feature-Engineering Experiment #7 using Five-Fold Cross-Validation: WordNet Negation Features and AFINN Sentiment Scores

[104]:

```
# Initialize the sentiment lexicon
afinn = Afinn()

# Store the average scores for each number of word features used
(
    w_neg_word_feature_counts_afinn,
    w_neg_mean_accuracies_afinn,
    w_neg_mean_precisions_afinn,
    w_neg_mean_recalls_afinn,
    w_neg_mean_f1s_afinn,
    w_neg_f1_std_devs_afinn,
    w_neg_f1_ranges_afinn
) = test_diff_word_feature_numbers(
    w_neg_recombined_train_tokens,
    recombined_train_labels,
    400,  # Start testing from 400 word features
    1201,  # End testing at 1200 word features (1201 is exclusive)
    50,  # Test every 50 word features
    doc_features_with_afinn_sentiment_scores, # replace the normal doc_features␣
 ↪function with this for adding AFINN scores as features
    k=5
)

# Plot the mean accuracies and mean macro-avg F1 scores for features including␣
 ↪AFINN scores
plot_impacts_of_different_word_counts(w_neg_word_feature_counts_afinn,␣
 ↪w_neg_mean_accuracies_afinn, w_neg_mean_f1s_afinn,
                                      "Best Number of Word Features for WordNet␣
 ↪Negation with AFINN Sentiment Lexicon Scores")
```

```
Number of word features: 400 --- Macro-Avg F1 standard deviation:
0.01033351113115565 --- Macro-Avg F1 Range: 0.027595995835560272
Number of word features: 450 --- Macro-Avg F1 standard deviation:
0.020146007537478902 --- Macro-Avg F1 Range: 0.04633326336429133
Number of word features: 500 --- Macro-Avg F1 standard deviation:
0.02192987158121347 --- Macro-Avg F1 Range: 0.05397094903432231
Number of word features: 550 --- Macro-Avg F1 standard deviation:
0.02893198270298793 --- Macro-Avg F1 Range: 0.07543619836995863
Number of word features: 600 --- Macro-Avg F1 standard deviation:
0.011734252519241139 --- Macro-Avg F1 Range: 0.028858390692579727
Number of word features: 650 --- Macro-Avg F1 standard deviation:
0.012516875576322231 --- Macro-Avg F1 Range: 0.024645615148630062
Number of word features: 700 --- Macro-Avg F1 standard deviation:
0.012329635686183792 --- Macro-Avg F1 Range: 0.027544783647724835
Number of word features: 750 --- Macro-Avg F1 standard deviation:
```

```
0.020429090817660464 --- Macro-Avg F1 Range: 0.049776055457411206
Number of word features: 800 --- Macro-Avg F1 standard deviation:
0.020117815545819102 --- Macro-Avg F1 Range: 0.05126934504323299
Number of word features: 850 --- Macro-Avg F1 standard deviation:
0.013940517925117041 --- Macro-Avg F1 Range: 0.03333870325718152
Number of word features: 900 --- Macro-Avg F1 standard deviation:
0.0167364032619493 --- Macro-Avg F1 Range: 0.03777178133250508
Number of word features: 950 --- Macro-Avg F1 standard deviation:
0.022278233424085626 --- Macro-Avg F1 Range: 0.05314551666654713
Number of word features: 1000 --- Macro-Avg F1 standard deviation:
0.025317898815145092 --- Macro-Avg F1 Range: 0.06819928010740733
Number of word features: 1050 --- Macro-Avg F1 standard deviation:
0.026205439366132247 --- Macro-Avg F1 Range: 0.05965285610446902
Number of word features: 1100 --- Macro-Avg F1 standard deviation:
0.03294061251310053 --- Macro-Avg F1 Range: 0.08410302043880075
Number of word features: 1150 --- Macro-Avg F1 standard deviation:
0.036111117561516644 --- Macro-Avg F1 Range: 0.08410302043880075
Number of word features: 1200 --- Macro-Avg F1 standard deviation:
0.033560733278419075 --- Macro-Avg F1 Range: 0.08259756931460915
```



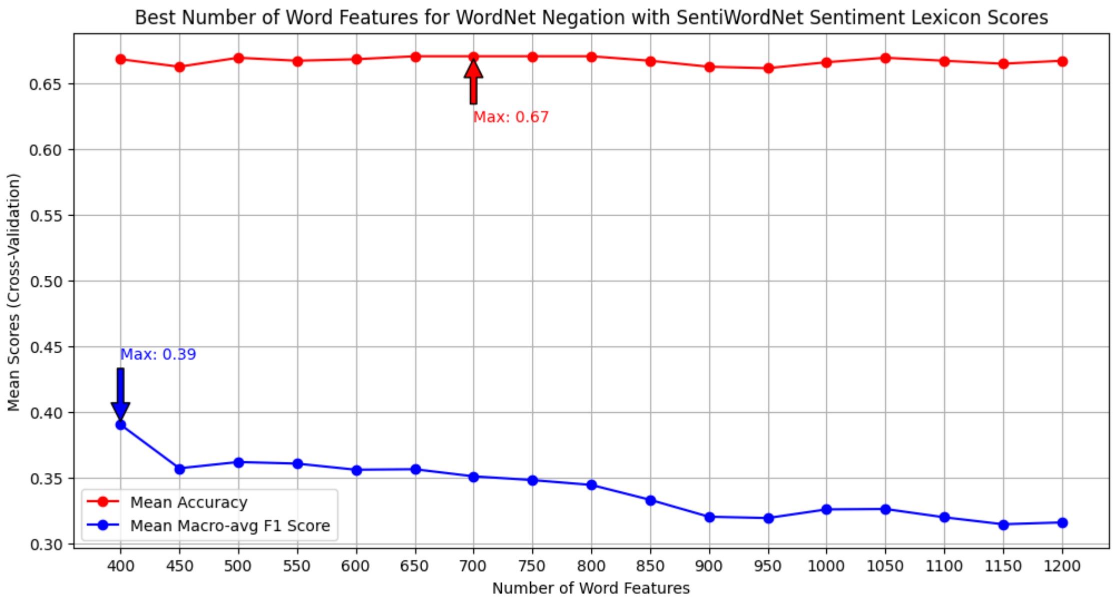
2.4.9 Feature-Engineering Experiment #8 using Five-Fold Cross-Validation: WordNet Negation Features and SentiWordNet Sentiment Scores

[105]:
```
# Store the average scores for each number of word features used
(
    w_neg_word_feature_counts_swt,
    w_neg_mean_accuracies_swt,
```

```
    w_neg_mean_precisions_swt,
    w_neg_mean_recalls_swt,
    w_neg_mean_f1s_swt,
    w_neg_f1_std_devs_swt,
    w_neg_f1_ranges_swt
) = test_diff_word_feature_numbers(
    w_neg_recombined_train_tokens,
    recombined_train_labels,
    400,  # Start testing from 400 word features
    1201,  # End testing at 1200 word features (1201 is exclusive)
    50,  # Test every 50 word features
    doc_features_with_swn_sentiment_scores, # replace the doc_features function␣
↪with this one for adding SentiWordNet score features
    k=5
)

# Plot the mean accuracies and mean macro-avg F1 scores for features including␣
↪SentiWordNet scores
plot_impacts_of_different_word_counts(w_neg_word_feature_counts_swt,␣
↪w_neg_mean_accuracies_swt, w_neg_mean_f1s_swt,
                                      "Best Number of Word Features for WordNet␣
↪Negation with SentiWordNet Sentiment Lexicon Scores")
```

```
Number of word features: 400 --- Macro-Avg F1 standard deviation:
0.03787817764191531 --- Macro-Avg F1 Range: 0.08215749225735047
Number of word features: 450 --- Macro-Avg F1 standard deviation:
0.013314068347908775 --- Macro-Avg F1 Range: 0.03261861363676061
Number of word features: 500 --- Macro-Avg F1 standard deviation:
0.014443001956581757 --- Macro-Avg F1 Range: 0.033003653794680954
Number of word features: 550 --- Macro-Avg F1 standard deviation:
0.018763306060283154 --- Macro-Avg F1 Range: 0.0438129063129063
Number of word features: 600 --- Macro-Avg F1 standard deviation:
0.018010251083597547 --- Macro-Avg F1 Range: 0.042585459509234425
Number of word features: 650 --- Macro-Avg F1 standard deviation:
0.02492494588809242 --- Macro-Avg F1 Range: 0.05890582697386293
Number of word features: 700 --- Macro-Avg F1 standard deviation:
0.025033542300305852 --- Macro-Avg F1 Range: 0.06332107205550219
Number of word features: 750 --- Macro-Avg F1 standard deviation:
0.0222250670486428 --- Macro-Avg F1 Range: 0.05585923111842911
Number of word features: 800 --- Macro-Avg F1 standard deviation:
0.027305692929341856 --- Macro-Avg F1 Range: 0.06637344456569705
Number of word features: 850 --- Macro-Avg F1 standard deviation:
0.03024333977672036 --- Macro-Avg F1 Range: 0.06795981498106873
Number of word features: 900 --- Macro-Avg F1 standard deviation:
0.034869000613650114 --- Macro-Avg F1 Range: 0.09126540126540122
Number of word features: 950 --- Macro-Avg F1 standard deviation:
0.031626841709972424 --- Macro-Avg F1 Range: 0.08291896592923548
```

```
Number of word features: 1000 --- Macro-Avg F1 standard deviation:
0.030329763210142226 --- Macro-Avg F1 Range: 0.08352947192892896
Number of word features: 1050 --- Macro-Avg F1 standard deviation:
0.02406770463823311 --- Macro-Avg F1 Range: 0.06358309218191893
Number of word features: 1100 --- Macro-Avg F1 standard deviation:
0.02111011210179634 --- Macro-Avg F1 Range: 0.05322283696872254
Number of word features: 1150 --- Macro-Avg F1 standard deviation:
0.01827999970913581 --- Macro-Avg F1 Range: 0.04770813108636962
Number of word features: 1200 --- Macro-Avg F1 standard deviation:
0.018891670528204438 --- Macro-Avg F1 Range: 0.049937869626312414
```



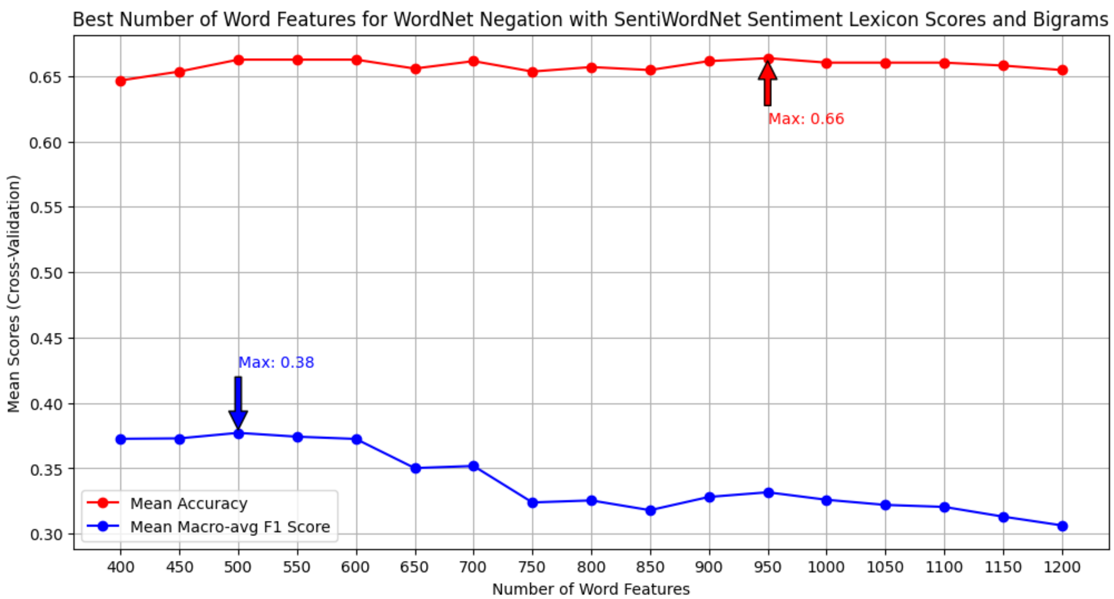
2.4.10 Feature-Engineering Experiment #9 using Five-Fold Cross-Validation: WordNet Negation Features, SentiWordNet Sentiment Scores and Bigrams

[106]:
```
# Add bigrams to token lists/samples
wordnet_negated_train_tokens_with_bigrams = [list(bigrams(sample)) + sample for␣
 ↪sample in w_neg_recombined_train_tokens]

(
    w_neg_word_feature_counts_swt_bg,
    w_neg_mean_accuracies_swt_bg,
    w_neg_mean_precisions_swt_bg,
    w_neg_mean_recalls_swt_bg,
    w_neg_mean_f1s_swt_bg,
    w_neg_f1_std_devs_swt_bg,
    w_neg_f1_ranges_swt_bg
) = test_diff_word_feature_numbers(
```

```
    wordnet_negated_train_tokens_with_bigrams,
    recombined_train_labels,
    400,  # Start testing from 400 word features
    1201,  # End testing at 1200 word features (1201 is exclusive)
    50,  # Test every 50 word features
    doc_features_with_swn_sentiment_scores_bigrams,
    k=5
)

# Plot the mean accuracies and mean macro-avg F1 scores for features including␣
↪SentiWordNet scores
plot_impacts_of_different_word_counts(w_neg_word_feature_counts_swt_bg,␣
↪w_neg_mean_accuracies_swt_bg, w_neg_mean_f1s_swt_bg,
                                      "Best Number of Word Features for WordNet␣
↪Negation with SentiWordNet Sentiment Lexicon Scores and Bigrams")
```

```
Number of word features: 400 --- Macro-Avg F1 standard deviation:
0.03578333877418685 --- Macro-Avg F1 Range: 0.09192385139535353
Number of word features: 450 --- Macro-Avg F1 standard deviation:
0.026798411032210274 --- Macro-Avg F1 Range: 0.06553395764553815
Number of word features: 500 --- Macro-Avg F1 standard deviation:
0.024668379945311743 --- Macro-Avg F1 Range: 0.0614862836516985
Number of word features: 550 --- Macro-Avg F1 standard deviation:
0.030215763693012796 --- Macro-Avg F1 Range: 0.07346178645233875
Number of word features: 600 --- Macro-Avg F1 standard deviation:
0.029153169423170616 --- Macro-Avg F1 Range: 0.0758788206684533
Number of word features: 650 --- Macro-Avg F1 standard deviation:
0.0456128226876324 --- Macro-Avg F1 Range: 0.12498765738673595
Number of word features: 700 --- Macro-Avg F1 standard deviation:
0.04071057452533891 --- Macro-Avg F1 Range: 0.10219232521864102
Number of word features: 750 --- Macro-Avg F1 standard deviation:
0.03277498754553256 --- Macro-Avg F1 Range: 0.07289156252321649
Number of word features: 800 --- Macro-Avg F1 standard deviation:
0.020356066886325755 --- Macro-Avg F1 Range: 0.055693366808343914
Number of word features: 850 --- Macro-Avg F1 standard deviation:
0.0117481035331114 --- Macro-Avg F1 Range: 0.03177490910565273
Number of word features: 900 --- Macro-Avg F1 standard deviation:
0.008191669772781003 --- Macro-Avg F1 Range: 0.020635380346945176
Number of word features: 950 --- Macro-Avg F1 standard deviation:
0.011681590780378646 --- Macro-Avg F1 Range: 0.027981740481740514
Number of word features: 1000 --- Macro-Avg F1 standard deviation:
0.012600346631277355 --- Macro-Avg F1 Range: 0.03095637883251323
Number of word features: 1050 --- Macro-Avg F1 standard deviation:
0.022777878926709787 --- Macro-Avg F1 Range: 0.06044655687626599
Number of word features: 1100 --- Macro-Avg F1 standard deviation:
0.020720577571838285 --- Macro-Avg F1 Range: 0.05147553145787215
Number of word features: 1150 --- Macro-Avg F1 standard deviation:
```

```
0.025567684213676476 --- Macro-Avg F1 Range: 0.06591432912338263
Number of word features: 1200 --- Macro-Avg F1 standard deviation:
0.021117166874537463 --- Macro-Avg F1 Range: 0.05471511980933286
```



2.5 Results Table for Experiments using Recombined Dataset and Stratified Five-Fold Cross-Validation

Experiment	Highest Accuracy and Word Feature Count	Highest Macro-Avg F1 and Word Feature Count
Basic Features (tokenization and word features)	0.66 (750)	0.33 (550)
Negation Handling using _NOT after negation cue	0.66 (750)	0.33 (550)
Wordnet Negation Handling	0.66 (750)	0.33 (550)
Wordnet Negation Handling and TF-IDF	0.63 (N/A)	0.20 (N/A)
Wordnet Negation Handling and Stopword Removal	0.66 (500)	0.32 (500)
Wordnet Negation Handling and Punct. Removal	0.65 (600)	0.32 (600)
Wordnet Negation Handling and Lemmatization	0.67 (800)	0.34 (650)
Wordnet Negation Handling and AFINN Lexicon	0.66 (750)	0.33 (550)
Wordnet Negation Handling and SWN Lexicon	**0.67 (700)**	**0.39 (400)**

Experiment	Highest Accuracy and Word Feature Count	Highest Macro-Avg F1 and Word Feature Count
Wordnet Negation Handling, SWN Lexicon & Bigrams	0.66 (950)	0.38 (500)

2.6 Discussion of Multinomial Naive Bayes Classifier Performance using the Recombined Dataset with Stratified Five-Fold Cross Validation

Despite these efforts, the accuracy and F1-scores remained low across all the folds of the training set. The mean validation accuracy was lower than on the original data split, at around 66%. Given there were 554 instances of the majority (neutral) class in this training set, and 880 samples overall, the effect of simply predicting the majority class every time would have been about 63% (as the validation set for each fold was stratified to contain the same proportion of each class every time). Consequently, even with the highest accuracy of 67%, the classifier only performs 4% better than making trivial predictions. The inclusion of the "mixed" sentiment samples in the validation data could be one reason for the inferior scores, due to both the small amount of these samples in the whole dataset, as well as the larger degree of ambiguity in their sentiment polarity.

When the WordNet negation handling technique with SentiWordNet sentiment scores was used, the F1-score was slightly better than for the other results (going from around 33% for the majority of the other experiments to 39%). As before, the F1-score using TF-IDF was extremely low - corroborating the suggestions made by Jurafsky and Martin, that word occurance vectors (binary vectorization) works better for sentiment analysis tasks than weighted frequency counts (2024).

The next step will be to evaluate the best scoring technique (WordNet Negation Handling with SentiWordNet Lexicon scores) on the recombined test set containing "mixed" sentiment samples, and to display the confusion matrix and to assess where the classifier struggled in more detail.

2.7 Evaluating the Multinomial Naive Bayes Classifier using Optimized Features (WordNet Negation Handling and SentiWordNet Lexicon Score) on the Recombined Test Set containing "Mixed" Sentiment Samples

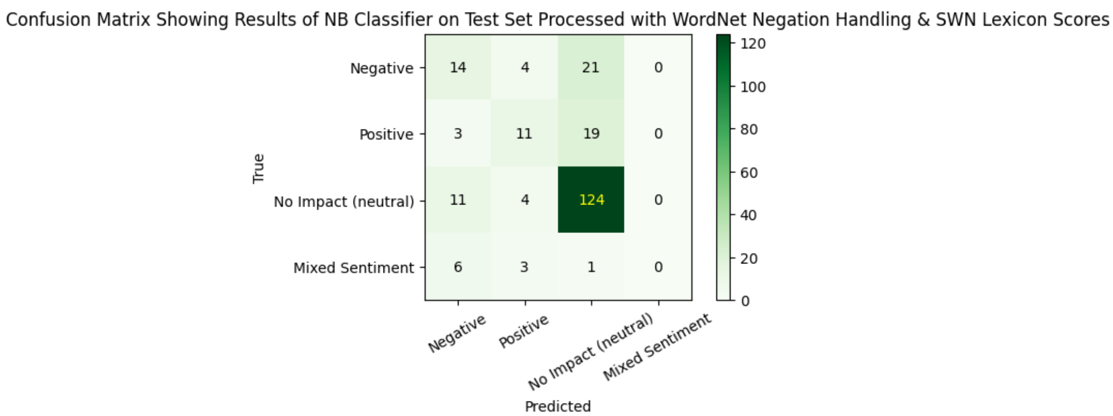
[90]:
```
# Apply negation handling to test token sets
w_neg_recombined_test_tokens = [handle_negation_with_wordnet(token_list,␣
↪negation_patterns) for token_list in recombined_test_tokens]

# Convert tokens and labels into tuples for␣
↪doc_features_with_swn_sentiment_scores function
all_train_tuples = list(zip(w_neg_recombined_train_tokens,␣
↪recombined_train_labels))
test_tuples = list(zip(w_neg_recombined_test_tokens, recombined_test_labels))

# Apply the SentiWordNet scores to the samples as well as creating the normal␣
↪absence-presence word features
all_train_featuresets = [(doc_features_with_swn_sentiment_scores(doc,␣
↪word_features), label) for (doc, label) in all_train_tuples]
```

```python
test_featuresets = [(doc_features_with_swn_sentiment_scores(doc,␣
↪word_features), label) for (doc, label) in test_tuples]

# Train NB Classifier on the entire stratified training set with mixed samples
NBclassifier = nltk.NaiveBayesClassifier.train(all_train_featuresets)

# Store predicted labels from the test set here
test_predictions= []

# Generate predictions with the trained NB classifier on each of the test␣
↪samples
for features_dict, label in test_featuresets:
    predicted_label = NBclassifier.classify(features_dict)
    test_predictions.append(predicted_label)

print(f"Classification report:\n{class_report}")
print(classification_report(
    recombined_test_labels,
    test_predictions,
    target_names=label_names,
    zero_division=0.0
    )
)
```

```
Classification report:
{'0': {'precision': 0.4117647058823529, 'recall': 0.358974358974359, 'f1-score':
0.3835616438356165, 'support': 39.0}, '1': {'precision': 0.5, 'recall':
0.3333333333333333, 'f1-score': 0.4, 'support': 33.0}, '2': {'precision':
0.7515151515151515, 'recall': 0.8920863309352518, 'f1-score':
0.8157894736842105, 'support': 139.0}, '3': {'precision': 0.0, 'recall': 0.0,
'f1-score': 0.0, 'support': 10.0}, 'accuracy': 0.6742081447963801, 'macro avg':
{'precision': 0.41581996434937607, 'recall': 0.396098505810736, 'f1-score':
0.3998377793799568, 'support': 221.0}, 'weighted avg': {'precision':
0.6199974189593566, 'recall': 0.6742081447963801, 'f1-score':
0.6405142124511055, 'support': 221.0}}
                     precision    recall  f1-score   support

           Negative       0.41      0.36      0.38        39
           Positive       0.50      0.33      0.40        33
No Impact (neutral)       0.75      0.89      0.82       139
    Mixed Sentiment       0.00      0.00      0.00        10

           accuracy                           0.67       221
          macro avg       0.42      0.40      0.40       221
       weighted avg       0.62      0.67      0.64       221
```

[88]:
```
# Re-use the generate_and_show_confusion_matrix function to visualize findings
generate_and_show_confusion_matrix(
    recombined_test_labels,
    test_predictions,
    label_names=label_names,
    classifier_description="Confusion Matrix Showing Results of NB Classifier␣
 ↪on Test Set Processed with WordNet Negation Handling & SWN Lexicon Scores"
)
```

```
[0 1 2 3]
```



- Compared to the performance on the original test split, all the scores on the new test set containing "mixed" sentiment classes were signficantly lower. The accuracy score was reduced from 71% to 67%, barely above the 63% required to outperform a random classifier. Macro-averaged recall, precision, and F1-scores were also much lower - reflected in the confusion matrix above, where *none* of the "mixed" sentiment classes were correctly identified (the 0 in the bottom-right corner). Logging the standard deviations across the five folds showed that the variation of macro-average F1-scores was very low (standard mostly less than 0.05), indicating similar validation performance across the folds. The similariy of the mean macro-average F1-score obtained using cross-validation (39%) to the score achieved on the test set (40%) shows that there has been less overfitting to the peculiarities of the validation data then when using the same original validation set for every experiment.
- However, despite the decrease in average scores, the confusion matrix shows that (14/25) = 56% of negative samples in the test set were correctly classified, compared to (4/15) = 27% in the test set of the original dataset split. Nonetheless, hardly any of the positive samples were correctly identified (only approximately 9%) with most being classified (19/33) as "neutral". Therefore, the classifier seems to struggle in distinguishing instances of positive from neutral sentiment. Performance on the majority ("neutral") class was decent, with 124 out of 139 samples being correctly predicted.
- Overall, this might suggest that the small size and unbalanced nature of the dataset constitutes a significant limitation that degrades the ability of the Naive Bayes classifier to learn the core relationships between word counts and sentiment polarity scores. As such, the curation

and publication of a much larger English poetry-sentiment dataset would be beneficial for further research projects in this direction. Additionally, the inability of traditional statistical methods to model the context of words may mean that more data is required for decent performance.

The next section will explore the effectiveness of a deep-learning transformer-based model (DistilBERT) when classifying such a difficult dataset. The deep-learning model has been pre-trained on a large corpus of web data, which can hopefully help mitigate some of the shortcomings related to small dataset size.

2.8 Poem Dataset Sentiment Analysis using the DistilBERT Uncased Deep-Learning Model

BERT-like transformer-based models are considered one of the state-of-the-art models for NLP classification tasks today (Tunstall, von Werra, & Wolf, 2022), using embeddings (dense vectors representing word meanings) as inputs. The original BERT model was trained on Wikipedia and BookCorpus (a set of unpublished books from different genres). However, the full BERT model has millions of learned parameters (weights), thus it would take a long time and intense computational resources to run experiments for fine-tuning the model for this specific task. Consequently, the Hugging Face DistilBERT model will be used instead. This is a smaller or "distilled" version of BERT, which the same "general architecture" as BERT but fewer hidden layers - yet retains 97% of the language understanding capability that BERT has (Sanh et al., 2020). Considering the time constraints and limited computational resources available for this project, DistilBERT seems like a reasonable choice. Additionally, as Tunstall, von Werra and Wolf state, BERT-based models are particularly well-suited to natural language *understanding* tasks (e.g. text classification), compared to GPT or T5 models geared towards translation or text generation tasks (2022). Using a pre-trained model can also help mitigate the problem of the small number of certain class instances in the dataset.

2.8.1 Why choose a transformer-based deep-learning model?

Prior to transformer-based models (developed c. 2017), the state-of-the-art in natural language processing involved using RNNs (recurrent neural networks) and LSTMs (long short-term memory networks). However, these networks were limited in several ways when compared to transformer-based models. First, they took a long time to train because they were unable to learn the contextual meanings of words in a sentence simultaneously or in parallel, thus are much more computationally expensive. Although training a transfomer-based model such as BERT from scratch takes a very long time and a large quantity of data, here a pre-trained model will be downloaded and merely fine-tuned on the poem dataset to avoid this problem. Furthermore, transformer-based models have outperformed RNNs and LSTMs in terms of accuracy and other metrics due to their ability to infer the context of a unit (token) in a text by simultaneously taking into account the tokens on *both* the left and right of that particular token, hence the term "bidirectional" in *Bidirectional Encoder Representations from Transformers* (BERT) (Tunstall, von Werra, & Wolf, 2022). This breakthrough was achieved by mobilizing the "attention mechanism" technique, detailed in the foundational text by Vaswani et al., *Attention Is All You Need" (Vaswani et al., 2023). As explained by Tunstall, von Werra and Wolf, the transformer-based network contains "self-attention layers" which allow the neural network to assign "a different amount of weight or 'attention' to each element [token embedding] in a sequence" of text (2022). The transformer-based network can therefore more successfully resolve the meaning of ambiguous tokens by constructing vectors representing the attention/importance of other tokens in relation to the token-in-question. This is done by taking the weighted average of different possible embeddings for a token, enabling a deeper understanding of context. For instance, the embedding for "apple" will be updated to be more "company-like" and less "fruit-like" if surrounded by words such as "phone" or "technology" (Tunstall, von Werra, & Wolf, 2022).

Thus, the decision to select a BERT-like model over RNNs/LSTMs for sequential text processing was informed by transformer-based models' improvements both in terms of computational efficiency and ability to infer word meaning by accounting for context on both sides of a word/token.

2.8.2 Comparison of Text-Processing Techniques for DistilBERT vs. for Multinomial Naive Bayes

The pre-processing of texts for deep-learning models such as DistilBERT involves a series of different techniques than those used above to construct inputs for the Naive Bayes experiments. These differences and the reasons for them will be discussed here.

- First, transformer-based models also require the raw text fragments to be tokenized - however, when using pre-trained models such as DistilBERT, it is imperative to use tokenizer class provided for this particular model. Using a different tokenizer would be like "shuffling the vocabulary" - for instance, replacing each instance of "night" with the word "grape", which would have an extremely negative impact on the model's performance (Tunstall, von Werra, & Wolf, 2022). As such, the *DistilBertTokenizer* will be downloaded from Hugging Face. These provided auto-tokenizers convert the tokens to unique integer IDs based on the vocabulary on which DistilBERT was trained. Frequently, special tokens such as [CLS] and [SEP] are added to indicate the start and end of a sentence. Frequently, auto-tokenizers also split texts into subwords in order to handle unusual words or to split words into roots and inflections (such as dividing "tokenizing" into "token" and "izing") (Tunstall, von Werra, & Wolf, 2022). As such the pre-trained tokenizer outputs a set of numbers (input IDs) for each sample. Each sample/text is also "padded" to make inputs the same size). The dictionary output of the pre-trained tokenizer thus contains an "attention mask" as a key-value pair containing a vector of 1s and 0s, with 1 representing a real word in the text and 0 representing a "padded" token. Making all the input tensors the same size facilitates the transformations applied to the texts by the neural network.

- Second, after receiving the token encodings as inputs, the transformer-based neural network the converts these token sequences into two kinds of embeddings: token embeddings and *positional embeddings* (which contain information about the **ordering** of the tokens in the text sample). These embeddings can be understood as a mapping of tokens to a point in a multi-dimensional space, where **tokens similar in meaning are physically closer to one another** (CodeEmporium, 2020). Each embedding is a dense vector (i.e. consisting of non-zero values). The task of the neural network is then to update the values in the embeddings for each token by passing them through multi-headed "attention layers" ("multi-headed" meaning that each layer focuses on different types of patterns - e.g. "subject-verb interaction" or finding "nearby adjectives") [Tunstall, von Werra, & Wolf, 2022]). These use the self-attention mechanism to compute how relevant each other token in the sentence is to that particular token. The other kind of layer used in BERT-like models are ordinary feed-forward neural layers that transform each attention vector to a form interpretable by the next layers

- Third, these pre-trained models have been trained on large corpora of data to learn patterns from different texts. Some of the pre-processing and feature extraction techniques employed above for training a Naive Bayes classifier are not applicable here for the following reasons:
  - First of all, adding negation markers such as ”_NOT” could hinder the performance of the pre-trained transformer-based models if such constructions are not part of the base vocabulary used when pre-training.
  - Second, the model has already been trained on large chunks of text data and has presumably learned patterns of how stopwords and punctuation contribute to the meaning of a sentence: the crude removal of these components could negatively impact the model’s ability to utilize the pre-learned patterns to the new prediction tasks. The removal of stopwords such as “not” or “but” can change the meaning of the entire text fragment. The transformer-based networks’ ability to model the entire context of a specific word in a text thus reduces the need for these pre-processing techniques.
  - As explained by Da (2019), the removal of stopwords can completely diminish a classifier’s ability to distinguish between different genres or types of literary texts. Additionally, normalizing tokens by lemmatizing or stemming them may interfer with the auto-tokenizer’s process of tokenizing words into subwords to mark certain inflections and the presence of certain grammatical patterns. Consequently, applying these feature-extraction techniques in this context could degrade the model’s ability to make predictions based on previously-learned patterns..

2.8.3 DistilBERT Optimization: Hyperparameter Tuning Algorithms

Although DistilBERT has already been pre-trained on Wikipedia and BookCorpus data, to further enhance its performance, the model will be optimized by fine-tuning it: this involves training it on the poem sentiment dataset. The models (such as this one) available on *HuggingFace* are perfectly suited for fine-tuning on “downstream” tasks such as this one, and involves slightly adjusting the learned weights of the original pre-trained model to adapt to the current dataset (Devlin, Chang, Lee, & Toutanova, 2019).

As before, the model will first be fine-tuned on the original dataset split (and evaluated on the original test set). Subsequently, the model will then be trained on different folds of the recombined dataset using stratified five-fold cross-validation as used previously for Naive Bayes.

While DistilBERT does not demand the same kind of intricate feature-extraction techniques that a statistical model does, its performance can greatly be optimized by the choice of model *hyperparameters* using this particular dataset. As the BERT authors state, while large datasets are not very sensitive to hyperparameter choice, with a learning rate out of [5e-5, 3e-5, 2e-5] usually being appropriate for most tasks (Devlin, Chang, Lee, & Toutanova, 2019), this dataset is very small and contains niche and ambiguous language. Therefore, it is probable that finding the optimal configuration of hyperparameters such as learning rate or number of training epochs is imperative for achieving decent performance on the test set.

Consequently, whereas finding the optimal hyperparameters using a basic grid search approach would be suitable for a larger dataset, where a relative small set of optimal configurations can be assessed (using the validation set or splits in the case of k-fold cross-validation), in this case, finding a reasonable set of hyperparameters is a more complex task. Initially, I attempted to use the "population based hyperparameter optimization" algorithm as outlined by Kamsetty, Fricke & Liaw (2020), which the authors report frequently leads to higher accuracy scores than grid search or Bayesian optimization. This algorithm, unlike Bayesian Optimization, "doesn't need to restart training for new hyperparameter configurations" but copies the "network weights and hyperparameters" of good trials/configurations to new trials, acting as a kind of evolutionary algorithm. Unfortunately, when trying to run algorithm this on the available machine using the *Ray Tune* optimization library, the computer would crash after training for a long time.

After repeated attempts, Bayesian optimization (also available with the *Ray Tune* library) with 10 trials was used to find a decent combination of hyperparameters. Bayesian search also uses the results of previous trials to model probability distributions to search for the next-best configuration that has the highest chance of improving a specific metric (e.g. F1-score), and can thus be much faster than a simple grid search that iterates through all the possible combinations of hyperparameters.

2.9 Optimizing DistilBERT Hyperparameters on the Original Dataset Test-Val-Train Split

[2]:
```
# Load in original train-val-test datasets and extract samples and labels
train_set = pd.read_csv("original_train_df.csv")
val_set = pd.read_csv("original_val_df.csv")
test_set = pd.read_csv("original_test_df.csv")
print(train_set.head(5))
# Convert from pandas Series to list as this is what the distilbert tokenizer␣
 ↪requires as inputs
train_texts = train_set["verse_text"].to_list()
train_labels = train_set["label"].to_list()
val_texts = val_set["verse_text"].to_list()
val_labels = val_set["label"].to_list()
test_texts = test_set["verse_text"].to_list()
test_labels = test_set["label"].to_list()

# Combine train and val set for final-model training to train on all the data␣
 ↪using the best hyperparameters before evaluating on the test set
combined_train_set = pd.concat([train_set, val_set], axis=0).
 ↪reset_index(drop=True)
print("len of combined trains set: ", len(combined_train_set))
combined_texts = combined_train_set["verse_text"].to_list()
combined_labels = combined_train_set["label"].to_list()

# Initializer the pre-trained DistilBERT tokenizer
tokenizer = DistilBertTokenizer.from_pretrained('distilbert-base-uncased')
# Find longest token length in dataset for the max_length parameter for the␣
 ↪tokenizer
max_length = 0
```

```python
# Tokenize each text sample and find max length to determine the␣
↪DistilBertTokenize input argument for max_length
for text in train_set["verse_text"]:
    # Tokenize text
    input_ids = tokenizer.encode(text, add_special_tokens=True) # special␣
↪tokens = indicate start of sequence, end of sequence, separation
    # Update max length
    max_length = max(max_length, len(input_ids))
print("Maximum sequence length:", max_length) # print max length for samples in␣
↪train set

max_length = 0
# Tokenize each val sample and find max length
for text in val_set["verse_text"]:
    # Tokenize text
    input_ids = tokenizer.encode(text, add_special_tokens=True) # special␣
↪tokens = indicate start of sequence, end of sequence, separation
    # Update max length
    max_length = max(max_length, len(input_ids))
print("Maximum sequence length:", max_length) # print max length for samples in␣
↪val set

max_length = 0
# Tokenize each val sample and find max length
for text in test_set["verse_text"]:
    # Tokenize text
    input_ids = tokenizer.encode(text, add_special_tokens=True) # special␣
↪tokens = indicate start of sequence, end of sequence, separation
    # Update max length
    max_length = max(max_length, len(input_ids))
print("Maximum sequence length:", max_length) # print max length for samples in␣
↪test set
```

```
   id                                         verse_text  label
0   0  with pale blue berries. in these peaceful shad…      1
1   1                 it flows so long as falls the rain,      2
2   2                  and that is why, the lonesome day,      0
3   3  when i peruse the conquered fame of heroes, an…      3
4   4             of inward strife for truth and liberty.      3
len of combined trains set:  997
Maximum sequence length: 28
Maximum sequence length: 27
Maximum sequence length: 20
```

[3]:

```
## Code adapted from: https://www.sunnyville.ai/
↪fine-tuning-distilbert-multi-class-text-classification-using-transformers-and-tensorflow/
↪
"""
    Fine-tuning in the HuggingFace's transformers library involves using a␣
↪pre-trained model and a tokenizer
    that is compatible with that model's architecture and input requirements.
    Each pre-trained model in transformers can be accessed using the right model
    class and be used with the associated tokenizer class. Since we want to use
    DistilBert for a classification task, we will use the DistilBertTokenizer␣
↪tokenizer
    class to tokenize our texts and then use␣
↪TFDistilBertForSequenceClassification
    model class in a later section to fine-tune the pre-trained model using the␣
↪output from the tokenizer.
    The DistilBertTokenizer generates input_ids and attention_mask as outputs.
    This is what is required by a DistilBert model as its inputs.
"""

tokenizer = DistilBertTokenizer.from_pretrained('distilbert-base-uncased')
train_encodings = tokenizer(train_texts, truncation=True, padding=True,␣
↪max_length=27)
val_encodings = tokenizer(val_texts, truncation=True, padding=True,␣
↪max_length=27)
test_encodings = tokenizer(test_texts, truncation=True, padding=True,␣
↪max_length=27)
combined_train_encodings = tokenizer(combined_texts, truncation=True,␣
↪padding=True, max_length=27)
print(train_encodings.keys())
# Code from: https://medium.com/@raoashish10/
↪fine-tuning-a-pre-trained-bert-model-for-classification-using-native-pytorch-c5f33e87616e
print(val_encodings['input_ids'][8])
print(val_encodings['attention_mask'][8])
print(tokenizer.decode(train_encodings['input_ids'][8]))
```

```
dict_keys(['input_ids', 'attention_mask'])
[101, 2010, 2132, 2003, 11489, 1012, 2002, 6732, 2006, 2273, 1998, 5465, 1012,
102, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0]
[CLS] and so on. then a worthless gaud or two, [SEP] [PAD] [PAD] [PAD] [PAD]
[PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD]
```

[4]:

```
## Code adapted from: https://huggingface.co/transformers/v3.4.0/
↪custom_datasets.html
```

```
# and from here: # https://medium.com/@vmn11/
↪text-classification-with-hugging-face-trainer-and-pytorch-8c0b07e67b4a
""" We put the data in this format so that the data can be easily batched such␣
↪that each key in the batch encoding
    corresponds to a named parameter of the forward() method of the model we␣
↪will train. """

# Create a custom dataset class inheriting from PyTorch's Dataset class -->␣
↪required as inputs to the DistilbertModel
class PoemDataset(torch.utils.data.Dataset):
    def __init__(self, encodings, labels):
        # Store list of encoded tokens here
        self.encodings = encodings
        # Store list of corresponding labels for each sample here
        self.labels = labels

    # This special __ getter function enables retrieving items in an encoding␣
↪using []-notation and 'idx' as the integer to index encodings
    def __getitem__(self, idx):
        # dict comprehension: creates a key for each key in the encoding for a␣
↪specific idx/sample: e.g., 'input_ids'
        # the values will be the list containing the encoded tokens, attention␣
↪masks etc.
        item = {key: torch.tensor(val[idx]) for key, val in self.encodings.
↪items()}
        # add key-value pair for label of indexed sample
        item['labels'] = torch.tensor(self.labels[idx])
        return item

    # Returns size of dataset
    def __len__(self):
        return len(self.labels)

train_dataset = PoemDataset(train_encodings, train_labels)
eval_dataset = PoemDataset(val_encodings, val_labels)
test_dataset = PoemDataset(test_encodings, test_labels)
combined_train_dataset = PoemDataset(combined_train_encodings, combined_labels)

# Verify this has worked by taking the first train sample as an example
print(train_texts[0], train_labels[0], '\n')
print(train_dataset[0])
```

```
with pale blue berries. in these peaceful shades-- 1

{'input_ids': tensor([  101,  2007,  5122,  2630, 22681,  1012,  1999,  2122,
9379, 13178,
         1011,  1011,   102,     0,     0,     0,     0,     0,     0,     0,
```

```
            0,      0,      0,      0,      0,      0,      0]), 'attention_mask':
tensor([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0]), 'labels': tensor(1)}
```

[1]:
```
# Reference: https://docs.ray.io/en/latest/train/getting-started-transformers.
↪html
# Metrics dictionary to use for evaluating performance with Ray Tune␣
↪hyperparameter search
def compute_metrics(prediction_obj):
    """
        Input:
            prediction_obj = the function takes in a prediction object from the␣
↪Ray Tune Bayes Optimizer containing the true labels and
            predictions for the validation set.
        Output:
            a dictionary containing the scores for accuracy, and macro-average␣
↪precision, recall and F1-scores
    """
    true_labels = prediction_obj.label_ids
    predicted_labels = prediction_obj.predictions.argmax(-1)
    accuracy = accuracy_score(true_labels, predicted_labels)
    precision, recall, f1, _ = precision_recall_fscore_support(true_labels,␣
↪predicted_labels, average='macro', zero_division=0.0)
    return {
        'eval_accuracy': accuracy,
        'eval_precision': precision,
        'eval_recall': recall,
        'eval_f1': f1,
    }
```

[ ]:
```
## Code adapted from the following:
# - https://huggingface.co/docs/transformers/en/hpo_train
# - https://huggingface.co/blog/ray-tune
# - https://wandb.ai/amogkam/transformers/reports/
↪Hyperparameter-Optimization-for-HuggingFace-Transformers--VmlldzoyMTc2ODI#bayesian-search-w
# - https://docs.ray.io/en/latest/tune/api/suggestion.html
def train_fn(config):
    """
        Trains a model using different combinations of hyperparameters searched␣
↪for using the Bayes Optimization algorithm
        and returns the performance scores for the validation set.
        Input:
            config = dictionary specifying the ranges for each hyperparameter
        Output:
            metrics = a dictionary containing the macro-average F1-score (main␣
↪metric for optimizing hyperparameters)
    """
```

```
    model = DistilBertForSequenceClassification.
↪from_pretrained("distilbert-base-uncased", num_labels=4)
    training_args = TrainingArguments(
        output_dir='./original_dataset_results',
        per_device_train_batch_size=config["per_device_train_batch_size"],
        per_device_eval_batch_size=config["per_device_eval_batch_size"],
        warmup_steps=config["warmup_steps"],
        weight_decay=config["weight_decay"],
        logging_dir='./logs',
        logging_steps=10,
        num_train_epochs=config["num_train_epochs"],
        learning_rate=config["learning_rate"]
    )
    trainer = Trainer(
        model=model,
        args=training_args,
        compute_metrics=compute_metrics,
        train_dataset=train_dataset,
        eval_dataset=eval_dataset
    )
    # Start training
    trainer.train()
    # Evaluate the model on the validation split
    eval_results = trainer.evaluate(eval_dataset=eval_dataset)
    print(eval_results)
    metrics = dict(eval_f1=eval_results["eval_f1"])

    ray.train.report(dict(eval_f1=eval_results["eval_f1"]))

    return metrics

"""
    Define the hyperparameter search space i.e.ranges to search for optimal␣
↪hyperparameters
    Values adapted slightly from these: https://medium.com/
↪distributed-computing-with-ray/
↪hyperparameter-optimization-for-transformers-a-guide-c4e32c6c989b
"""
# Define the hyperparameter search space i.e.ranges to search for optimal␣
↪hyperparameters
# Values adapted slightly from these: https://medium.com/
↪distributed-computing-with-ray/
↪hyperparameter-optimization-for-transformers-a-guide-c4e32c6c989b
search_space = {
    "num_train_epochs": tune.uniform(6, 11),  # Continuous range between 6 and␣
↪10
```

```
    "per_device_train_batch_size": 16,
    "per_device_eval_batch_size": 16,
    # From" https://www.geeksforgeeks.org/
 ↪in-the-context-of-deep-learning-what-is-training-warmup-steps/
    # "Warmup steps are often used in conjunction with learning rate schedules␣
 ↪such as learning rate decay or cyclic learning rates.
    # By initially increasing the learning rate, warmup steps help the model to␣
 ↪adapt more effectively to these schedules and optimize
    # the learning process."
    "warmup_steps": tune.uniform(0, 1000),  # Continuous range between 0 and 999
    # Values based on this article:
    "weight_decay": tune.uniform(0.001, 0.1),  # Continuous range between 0.001␣
 ↪and 0.1
    "learning_rate": tune.uniform(1e-5, 5e-5)  # Evaluate a continuous range␣
 ↪between 1e-5 and 5e-5
}


"""
        # Utility kwargs used from: https://docs.ray.io/en/latest/tune/examples/
 ↪includes/bayesopt_example.html
        # Explanation: https://www.geeksforgeeks.org/
 ↪upper-confidence-bound-algorithm-in-reinforcement-learning/
        `Upper-Confidence Bound action selection uses uncertainty in the␣
 ↪action-value estimates for balancing exploration and exploitation.`
"""
bayesopt = BayesOptSearch(
    metric="eval_f1",
    mode="max",
    utility_kwargs={
        "kind": "ucb",   # Use Upper Confidence Bound algorithm to balance␣
 ↪exploring the hyperparameter space and exploitation of existing solutions
        "kappa": 2.5,    # The UCB parameter for exploration-exploitation␣
 ↪trade-off
        "xi": 0.0
    }
)

# Reference: https://docs.ray.io/en/latest/tune/api/schedulers.html
asha_scheduler = ASHAScheduler(
    max_t=20,  # limit the maximum number of training epochs
    grace_period=5,  # minimum number epochs to run before stopping a trial
    reduction_factor=3  # a factor to reduce the number of trials each␣
 ↪iteration and speed up the process
)
```

```python
# Defines a short string to shorten the path to the trial logs
def short_trial_dirname_creator(trial):
    return f"tune_trial_{trial.trial_id}"

# Shut down any old already running instances of the Ray tuner
ray.shutdown()

# Initialize Ray and run hyperparameter search with Ray Tune
ray.init(ignore_reinit_error=True)
analysis = tune.run(
    train_fn, # function for training the model and evaluating performance on␣
 ↪the validation set
    config=search_space, # hyperparameter space
    search_alg=bayesopt,  # use Bayesian Optimization algorithm for␣
 ↪hyperparameter optimization
    scheduler=asha_scheduler,  # use ASHA scheduler to manage max and min␣
 ↪training epochs to run for
    resources_per_trial={"cpu": 4},
    num_samples=10, # number of trials/configurations of hyperparameters to test
    progress_reporter=tune.CLIReporter(metric_columns=["eval_f1"]), # report␣
 ↪macro-average F1-scores to view progress
    trial_dirname_creator=short_trial_dirname_creator, # where to save the logs
    local_dir='./ray', # local directory to save the training outputs
    mode="max",
    metric="eval_f1"
)
```

2.9.1 Last Lines of Output of Bayes Optimization Above

Copied and pasted, otherwise the progress logs took up 4000 pages of the output PDF

(train_fn pid=32680) {'train_runtime': 1298.275, 'train_samples_per_second': 4.458, 'train_steps_per_second': 0.28, 'train_loss': 0.9015562481932587, 'epoch': 6.5} 0%| | 0/7 [00:00<?, ?it/s] 29%| | 2/7 [00:00<00:01, 4.08it/s] 43%| | 3/7 [00:00<00:01, 2.99it/s] 100%| | 364/364 [21:38<00:00, 3.57s/it] [repeated 2x across cluster] == Status == Current time: 2024-06-27 02:53:02 (running for 02:08:15.82) Using AsyncHyperBand: num_stopped=0 Bracket: Iter 15.000: None | Iter 5.000: None Logical resource usage: 4.0/12 CPUs, 0/0 GPUs Current best trial: aa52c7e4 with eval_f1=0.8388250319284802 and parameters={'num_train_epochs': 6.779972601681013, 'per_device_train_batch_size': 16, 'per_device_eval_batch_size': 16, 'warmup_steps': 58.08361216819946, 'weight_decay': 0.08675143843171859, 'learning_rate': 1.624074561769746e-05} Result logdir: C:/Users/ophel/AppData/Local/Temp/ray/session_2024-06-27_00-44-42_896202_21148/artifacts/2024-06-27_00-44-46/train_fn_2024-06-27_00-44-46/driver_artifacts Number of trials: 10/10 (1 RUNNING, 9 TERMINATED)

57%| | 4/7 [00:01<00:01, 2.74it/s] 71%| | 5/7 [00:01<00:00, 2.58it/s] 86%| | 6/7 [00:02<00:00, 2.54it/s] 2024-06-27 02:53:03,537 WARNING experiment_state.py:205 – Experiment state snapshotting has been triggered multiple times in the last 5.0 seconds. A snapshot is forced if `CheckpointConfig(num_to_keep)` is set, and a trial has checkpointed >= `num_to_keep` times since the last snapshot. You may want to consider increasing the `CheckpointConfig(num_to_keep)` or decreasing the frequency of saving checkpoints. You can suppress this error by setting the environment variable TUNE_WARN_EXCESSIVE_EXPERIMENT_CHECKPOINT_SYNC_THRESHOLD_S to a smaller value than the current threshold (5.0). 2024-06-27 02:53:03,559 INFO tune.py:1016 – Wrote the latest version of all result files and experiment state to 'C:/Users/ophel/ray_results/train_fn_2024-06-27_00-44-46' in 0.0361s. 2024-06-27 02:53:03,567 INFO tune.py:1048 – Total run time: 7697.02 seconds (7696.95 seconds for the tuning loop). (train_fn pid=32680) {'eval_accuracy': 0.8857142857142857, 'eval_precision': 0.903639240506329, 'eval_recall': 0.7729618163054696, 'eval_f1': 0.8187473187473188, 'eval_loss': 0.4741998314857483, 'eval_runtime': 2.9204, 'eval_samples_per_second': 35.954, 'eval_steps_per_second': 2.397, 'epoch': 6.5} == Status == Current time: 2024-06-27 02:53:03 (running for 02:08:16.99) Using AsyncHyperBand: num_stopped=0 Bracket: Iter 15.000: None | Iter 5.000: None Logical resource usage: 4.0/12 CPUs, 0/0 GPUs Current best trial: aa52c7e4 with eval_f1=0.8388250319284802 and parameters={'num_train_epochs': 6.779972601681013, 'per_device_train_batch_size': 16, 'per_device_eval_batch_size': 16, 'warmup_steps': 58.08361216819946, 'weight_decay': 0.08675143843171859, 'learning_rate': 1.624074561769746e-05} Result logdir: C:/Users/ophel/AppData/Local/Temp/ray/session_2024-06-27_00-44-42_896202_21148/artifacts/2024-06-27_00-44-46/train_fn_2024-06-27_00-44-46/driver_artifacts Number o

Trial name	status	loc	learning_rate	num_train_epochs	warmup_steps	weight_decay	eval_f1
train_fn_48c3d075	TERMINATED	127.0.0.1:2841	2.49816e-05	10.7536	731.994	0.0602672	0.826909
train_fn_aa52c7e4	TERMINATED	127.0.0.1:2092	1.62407e-05	6.77997	58.0836	0.0867514	0.838825
train_fn_59b32e3e	TERMINATED	127.0.0.1:7172	3.40446e-05	9.54036	20.5845	0.0970211	0.597115
train_fn_b804f1e6	TERMINATED	127.0.0.1:2290	4.32977e-05	7.0617	181.825	0.019157	0.821013
train_fn_3aafa104	TERMINATED	127.0.0.1:2215	2.621697e-05	8.62378	431.945	0.0298317	0.595399
train_fn_fe2b6e42	TERMINATED	127.0.0.1:1858	3.44741e-05	6.69747	292.145	0.0372698	0.603247
train_fn_6bd6bb5b	TERMINATED	127.0.0.1:9628	2.82428e-05	9.92588	199.674	0.0519092	0.587841
train_fn_f50c1ee1	TERMINATED	127.0.0.1:2882	3.36966e-05	6.23225	607.545	0.0178819	0.809794
train_fn_d3658f19	TERMINATED	127.0.0.1:1223	1.226021e-05	10.7444	965.632	0.0810313	0.837876
train_fn_f2e7ffaa	TERMINATED	127.0.0.1:3268	2.021846e-05	6.48836	684.233	0.0445751	**0.818747**
0.818747							

—————-+—————–-+————+

—+——————-+————+

2.10 Testing the Fine-Tuned DistilBERT Model on the Original Test Set

[12]:
```python
# Retrieve and print the best hyperparameters
best_trial = analysis.get_best_trial(metric="eval_f1", mode="max")
print("Best hyperparameters found:", best_trial.config)
```

```
Best hyperparameters found: {'num_train_epochs': 6.779972601681013,
'per_device_train_batch_size': 16, 'per_device_eval_batch_size': 16,
'warmup_steps': 58.08361216819946, 'weight_decay': 0.08675143843171859,
'learning_rate': 1.624074561769746e-05}
```

[8]:
```python
# Now train the model on all of the train and validation data and evaluate on␣
↪test set
model = DistilBertForSequenceClassification.
↪from_pretrained('distilbert-base-uncased', num_labels=4)

# Store hyperparameter values
best_num_training_epochs = 6
batch_size = 16
best_number_warmup_steps = 58
best_weight_decay = 0.08675
best_learning_rate = 1.624074561769746e-05

# Define training_args by using the optimum hyperparameters found using␣
↪Bayesian optimization search
training_args = TrainingArguments(
    output_dir='./original_dataset_training_results',  # output directory
    num_train_epochs=best_num_training_epochs, # total number of training epochs
    per_device_train_batch_size=batch_size,  # batch size per device during␣
↪training
    per_device_eval_batch_size=batch_size,   # batch size for evaluation
    warmup_steps=best_number_warmup_steps,   # number of warmup steps for␣
↪learning rate scheduler
    weight_decay=best_weight_decay,           # strength of weight decay
    logging_dir='./original_dataset_training_logs', # directory for storing logs
    logging_steps=10,
    evaluation_strategy="no",  # No validation set during training, as we're␣
↪training on the entire training data set
    learning_rate=best_learning_rate
)

# Create a Trainer class provided by the HuggingFace transformers library
trainer = Trainer(
    model=model,
    args=training_args,
```

```
    train_dataset=combined_train_dataset, # train on the whole training set now
 ↪with the optimal parameters
    eval_dataset=None,  # No validation set during training for evaluation on
 ↪the test set
    compute_metrics=compute_metrics  # Include compute_metrics here
)

trainer.train()
```

```
Some weights of DistilBertForSequenceClassification were not initialized from
the model checkpoint at distilbert-base-uncased and are newly initialized:
['classifier.bias', 'classifier.weight', 'pre_classifier.bias',
'pre_classifier.weight']
You should probably TRAIN this model on a down-stream task to be able to use it
for predictions and inference.
C:\Users\ophel\AppData\Local\Programs\Python\Python38\lib\site-
packages\transformers\training_args.py:1474: FutureWarning:
`evaluation_strategy` is deprecated and will be removed in version 4.46 of
Transformers. Use `eval_strategy` instead
  warnings.warn(
```

```
<IPython.core.display.HTML object>
```

[8]:
```
TrainOutput(global_step=378, training_loss=0.5484916041137049,
metrics={'train_runtime': 534.4647, 'train_samples_per_second': 11.193,
'train_steps_per_second': 0.707, 'total_flos': 41789262768624.0, 'train_loss':
0.5484916041137049, 'epoch': 6.0})
```

[9]:
```
# Evaluate the model on the test dataset
eval_results = trainer.evaluate(eval_dataset=test_dataset)
print(eval_results)

# Get the predicted values
predictions = trainer.predict(test_dataset=test_dataset)
predicted_labels = np.argmax(predictions.predictions, axis=1)
true_labels = test_dataset.labels

true_labels = np.array(true_labels)
predicted_labels = np.array(predicted_labels)
```

```
<IPython.core.display.HTML object>
```

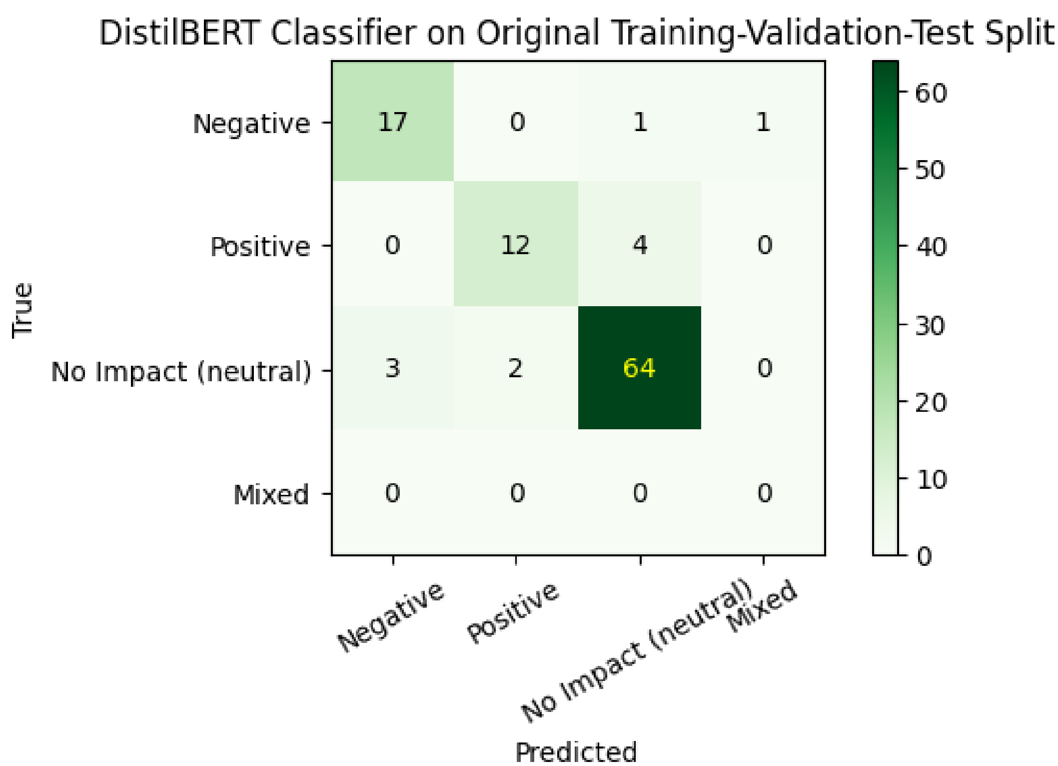
```
{'eval_accuracy': 0.8942307692307693, 'eval_precision': 0.6586697722567287,
'eval_recall': 0.6430682684973303, 'eval_f1': 0.6498327759197324, 'eval_loss':
0.30655357241630554, 'eval_runtime': 1.6983, 'eval_samples_per_second': 61.237,
'eval_steps_per_second': 4.122, 'epoch': 6.0}
```

[11]:
```python
# Store a set of all the target/label names in ascending order for easier␣
↪interpretation of the classification report and confusion matrix.
# Reminder: 0 = negative, 1 = positive, 2 = neutral, 3 =mixed (missing from␣
↪original test set)
label_names = ["Negative", "Positive", "No Impact (neutral)", "Mixed"]

# Print classification report to view the precision, recall, f1 score for each␣
↪class and macro-average
print("CLASSIFICATION REPORT:\n")
print(classification_report(
    true_labels, # labels are the same for the negated featureset as for the␣
↪original featureset
    predicted_labels,
    target_names=label_names)
)
```

```
CLASSIFICATION REPORT:

                     precision    recall  f1-score   support

           Negative       0.85      0.89      0.87        19
           Positive       0.86      0.75      0.80        16
No Impact (neutral)       0.93      0.93      0.93        69
              Mixed       0.00      0.00      0.00         0

           accuracy                           0.89       104
          macro avg       0.66      0.64      0.65       104
       weighted avg       0.90      0.89      0.90       104

C:\Users\ophel\AppData\Local\Programs\Python\Python38\lib\site-
packages\sklearn\metrics\_classification.py:1471: UndefinedMetricWarning: Recall
and F-score are ill-defined and being set to 0.0 in labels with no true samples.
Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
C:\Users\ophel\AppData\Local\Programs\Python\Python38\lib\site-
packages\sklearn\metrics\_classification.py:1471: UndefinedMetricWarning: Recall
and F-score are ill-defined and being set to 0.0 in labels with no true samples.
Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
C:\Users\ophel\AppData\Local\Programs\Python\Python38\lib\site-
packages\sklearn\metrics\_classification.py:1471: UndefinedMetricWarning: Recall
and F-score are ill-defined and being set to 0.0 in labels with no true samples.
Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
```

```
[14]: generate_and_show_confusion_matrix(
        true_labels, predicted_labels,
        label_names,
        "DistilBERT Classifier on Original Training-Validation-Test Split",␣
 ↪matrix_color=plt.cm.Greens, # default: green colour-coded c matrix
        above_threshold_text_color="yellow" # color in which to show the text␣
 ↪of counts above the threshold
    )
```

```
[0 1 2 3]
```




```
[15]: # Print each sample/text with true and predicted labels where misclassified␣
 ↪example
print("\nERROR ANALYSIS:\n")
for i, (text, true_label, pred_label) in enumerate(zip(test_texts, true_labels,␣
 ↪predicted_labels)):
    if true_label != pred_label:
        print(f"Sample {i + 1}:")
        print(f"Text: {text}")
        print(f"True Label: {label_names[true_label]}")
        print(f"Predicted Label: {label_names[pred_label]}\n")
```

ERROR ANALYSIS:

Sample 5:
Text: if they are hungry, paradise
True Label: No Impact (neutral)
Predicted Label: Negative

Sample 18:
Text: for penance, by a saintly styrian monk
True Label: No Impact (neutral)
Predicted Label: Positive

Sample 22:
Text: how hearts were answering to his own,
True Label: Positive
Predicted Label: No Impact (neutral)

Sample 25:
Text: 'tis to behold his vengeance for my son.
True Label: Negative
Predicted Label: Mixed

Sample 47:
Text: midway the floor (with thatch was it strewn) burned ever the fire-flame
True Label: No Impact (neutral)
Predicted Label: Negative

Sample 48:
Text: "i rather should have hewn your limbs away,
True Label: Negative
Predicted Label: No Impact (neutral)

Sample 63:
Text: dauntless he rose, and to the fight return'd;
True Label: Positive
Predicted Label: No Impact (neutral)

Sample 74:
Text: of your strong and pliant branches,
True Label: Positive
Predicted Label: No Impact (neutral)

Sample 78:
Text: i see the thing as clear as light,--
True Label: Positive
Predicted Label: No Impact (neutral)

```
Sample 83:
Text: like those famed seven who slept three ages.
True Label: No Impact (neutral)
Predicted Label: Positive

Sample 104:
Text: daring to ask for naught, and having naught received.
True Label: No Impact (neutral)
Predicted Label: Negative
```

The error analysis highlights why the classifier may have struggled to differentiate between positive and neutral samples in particular. Some of the samples annotated with a "positive" label, such as "how hearts were answering to his own," or "of your strong and pliant branches," do not seem as though they clearly reflect "positive" sentiment. Without any additional context, it is easy to see how these samples might have been classed as "neutral". This might be due to labelling errors by the annotators or the subjective nature of assessing sentiment in poetry.

2.11 Training and Optimizing DistilBERT Hyperparameters on Recombined Dataset with Five-Fold Cross-Validation

[8]:
```python
# Load in the recombined datasets as pandas DataFrames
recombined_train_set = pd.read_csv("new_poem_train_set.csv")
recombined_test_set = pd.read_csv("new_poem_test_set.csv")

# Extract the columns of samples and labels and store as lists
recombined_train_texts = recombined_train_set["verse_text"].to_list()
recombined_train_labels = recombined_train_set["label"].to_list()
recombined_test_texts = recombined_test_set["verse_text"].to_list()
recombined_test_labels = recombined_test_set["label"].to_list()

# Tokenize the samples using the pre-trained DistilBERT tokenizer
tokenizer = DistilBertTokenizer.from_pretrained('distilbert-base-uncased')
recombined_train_encodings = tokenizer(recombined_train_texts, truncation=True,␣
 ↪padding=True, max_length=27)
recombined_test_encodings = tokenizer(recombined_test_texts, truncation=True,␣
 ↪padding=True, max_length=27)

# Convert tokens and labels to a PyTorch dataset for trainer input
recombined_train_dataset = PoemDataset(recombined_train_encodings,␣
 ↪recombined_train_labels)
recombined_test_dataset = PoemDataset(recombined_test_encodings ,␣
 ↪recombined_test_labels)
```

[13]:
```python
# Function to pass into ray.tune Bayesian Optimization hyperparameter search
def train_fn(config):
```

```
    output_dir = f"recombined_dataset_results"
    model = DistilBertForSequenceClassification.
↪from_pretrained("distilbert-base-uncased", num_labels=4)

    training_args = TrainingArguments(
        output_dir='./recombined_dataset_results',
        per_device_train_batch_size=config["per_device_train_batch_size"],
        per_device_eval_batch_size=config["per_device_eval_batch_size"],
        warmup_steps=config["warmup_steps"],
        weight_decay=config["weight_decay"],
        logging_dir='./logs',
        logging_steps=10,
        num_train_epochs=config["num_train_epochs"],
        learning_rate=config["learning_rate"]
    )
    trainer = Trainer(
        model=model,
        args=training_args,
        compute_metrics=compute_metrics,
        train_dataset=train_dataset,
        eval_dataset=eval_dataset
    )

    # Start training
    trainer.train()
    # Evaluate the model
    eval_results = trainer.evaluate(eval_dataset=eval_dataset)
    print(eval_results)

    # Return and log all the metrics this time
    metrics = dict(
        eval_accuracy=eval_results["eval_accuracy"],
        eval_precision=eval_results["eval_precision"],
        eval_recall=eval_results["eval_recall"],
        eval_f1=eval_results["eval_f1"]
    )

    ray.train.report(
        dict(
            eval_accuracy=eval_results["eval_accuracy"],
            eval_precision=eval_results["eval_precision"],
            eval_recall=eval_results["eval_recall"],
            eval_f1=eval_results["eval_f1"]
        )
    )
    return metrics
```

```
 # Define a shorter path for trial logs
def short_trial_dirname_creator(trial):
    return f"tune_trial_{trial.trial_id}_recombined_ds"

# Define the hyperparameter space
search_space = {
    "num_train_epochs": tune.uniform(6, 11),  # Continuous range between 6 and␣
 ↪10
    "per_device_train_batch_size": 16,
    "per_device_eval_batch_size": 16,
    "warmup_steps": tune.uniform(0, 1000),  # Continuous range between 0 and 999
    "weight_decay": tune.uniform(0.001, 0.1),  # Continuous range between 0.001␣
 ↪and 0.1
    "learning_rate": tune.uniform(1e-5, 5e-5)  # Continuous range between 1e-5␣
 ↪and 5e-5
}


# Create stratified cross-validation index generator using the same␣
 ↪random_state (3) as before for reproducibility of results.
SKFGenerator = StratifiedKFold(n_splits=5, shuffle=True, random_state=3)

# Arrays to store the results of each cross-validation fold in for comparisons
best_configurations = []
best_trial_metric_scores = []
```

[ ]:
```
# Iterate over the 5 folds, enumerate to retrieve the index of the fold, and␣
 ↪the selected indices for the new train and val splits per fold
for fold, (train_indices, eval_indices) in enumerate(SKFGenerator.
 ↪split(recombined_train_texts, recombined_train_labels)):
    print(f"\n****************************************************FOLD {fold +␣
 ↪1}****************************************************")

    """
        Store the lists of encodings from recombined_train_encodings (from␣
 ↪DistilBERT tokenizer that outputs token-lists and attention mask),
        by creating a dictionary comprehension that iterates over each␣
 ↪key-value pair in the DistilBERT encodings and selects
        the key-value pairs corresponding to the indices of this fold's train␣
 ↪and val sets.
        Each sample will be a dict consisting of the keys input_ids (tokens/
 ↪encodings) and attention_mask.
    """
```

```
    train_encodings = {key: [val[i] for i in train_indices] for key, val in recombined_train_encodings.items()}
    eval_encodings = {key: [val[i] for i in eval_indices] for key, val in recombined_train_encodings.items()}

    # Store the labels for the current fold's train and val indices from recombined_train_labels
    train_labels = [recombined_train_labels[i] for i in train_indices]
    eval_labels = [recombined_train_labels[i] for i in eval_indices]

    # Convert the current DistilBERT encodings into two PoemDataset (torch) instances to be the proper inputs to the Trainer
    train_dataset = PoemDataset(train_encodings, train_labels)
    eval_dataset = PoemDataset(eval_encodings, eval_labels)

    # Create an instance of a Bayes Optimizer instance and metric to use (evaluation set macro-average F1-score) for making improvements
    # Reference: https://docs.ray.io/en/latest/tune/api/doc/ray.tune.search.bayesopt.BayesOptSearch.html
    bayesopt = BayesOptSearch(
        metric="eval_f1",
        mode="max",
        utility_kwargs={
            "kind": "ucb",   # Use Upper Confidence Bound (UCB) utility function
            "kappa": 2.5,    # UCB parameter for exploration-exploitation trade-off
            "xi": 0.0        # Expected Improvement (EI) parameter, trade-off between certain and possible improvements
        }
    )
    # Create an instance of a ASHA Scheduler to define max number of epochs to train for
    asha_scheduler = ASHAScheduler(
        max_t=20,  # Max number of epochs
        grace_period=5,  # Min number of epochs to run before stopping a trial
        reduction_factor=3  # Factor to reduce the number of trials each iteration
    )

    # Shut down the previous ray optimizer if there is one
    ray.shutdown()
    # Re-initialize the ray optimizer
    ray.init(ignore_reinit_error=True)
    # Run the ray optimizer with Bayesian Optimization and save the results
    results = tune.run(
        train_fn, # Pass in the Trainer function defined in the cell above
```

```
        search_alg=bayesopt,  # Use BayesOptSearch for hyperparameter␣
↪optimization
        scheduler=asha_scheduler,  # Use ASHA scheduler to manage epochs
        config=search_space,
        resources_per_trial={"cpu": 4},
        num_samples=10, # Trials to run
        progress_reporter=tune.CLIReporter(metric_columns=["eval_accuracy",␣
↪"eval_precision", "eval_recall", "eval_f1"]), # Metrics to print out
        trial_dirname_creator=short_trial_dirname_creator,
        local_dir='./ray_recombined',
        mode="max", # max F1 score is the metric to optimize
        metric="eval_f1"
    )
    best_trial = results.get_best_trial(metric="eval_f1", mode="max")
    print("Best hyperparameters found:", best_trial.config)
    best_configurations.append(best_trial.config)
    best_trial_metrics = best_trial.last_result
    print("Best trial metrics:", best_trial_metrics)
    best_trial_metric_scores.append(best_trial_metrics)
```

2.11.1 Last Lines of Output of Hyperparameter Optimization

(otherwise 1000s of pages of progress of Bayesian Optimization) == Status == Current time: 2024-06-28 11:01:33 (running for 01:39:53.29) Using AsyncHyperBand: num_stopped=0 Bracket: Iter 15.000: None | Iter 5.000: None Logical resource usage: 4.0/12 CPUs, 0/0 GPUs Current best trial: 06c2a1e8 with eval_f1=0.7183126027230485 and parameters={'num_train_epochs': 7.061695553391381, 'per_device_train_batch_size': 16, 'per_device_eval_batch_size': 16, 'warmup_steps': 181.82496720710063, 'weight_decay': 0.01915704647548995, 'learning_rate': 4.329770563201687e-05} Result logdir: C:/Users/ophel/AppData/Local/Temp/ray/session_2024-06-28_09-21-36_840424_20544/artifacts/2024-06-28_09-21-40/train_fn_2024-06-28_09-21-40/driver_artifacts Number of trials: 10/10 (10 TERM

Trial name	status	loc	learning_rate	num_train_epochs	warmup_steps	weight_decay	eval_accuracy	eval_precision	eval_recall	eval_f1
train_fn_[illegible]	TERMINATED	127.0.0.1:[illegible]	[illegible]e-05	10.7536	731.994	0.060267[illegible]	0.823864	0.66884	0.688206	0.673753
train_fn_[illegible]	TERMINATED	127.0.0.1:[illegible]	[illegible]e-05	6.77997	58.0836	0.086751[illegible]	0.8125	0.562087	0.626901	0.588243
train_fn_[illegible]	TERMINATED	127.0.0.1:[illegible]	[illegible]e-05	9.54036	20.5845	0.097021[illegible]	0.795455	0.661988	0.669874	0.663041
train_fn_06c2a1e8	TERMINATED	127.0.0.1:[illegible]	4.32977e-05	7.0617	181.825	0.019157	0.840909	0.752128	0.69964	**0.718313**
train_fn_[illegible]	TERMINATED	127.0.0.1:[illegible]	[illegible]e-05	8.62378	431.945	0.029831[illegible]	0.806818	0.705536	0.644229	0.632926
train_fn_[illegible]	TERMINATED	127.0.0.1:[illegible]	[illegible]e-05	6.69747	292.145	0.037269[illegible]	0.784091	0.654469	0.589817	0.594106

Trial name	status	loc	learning_rate	num_train_epochs	warmup_steps	weight_decay	eval_accuracy	eval_precision	eval_recall	eval_f1
train_fn_[illegible]	TERMINATED	127.0.0.1:[illegible]	2.82428e-05	9.92588	199.674	0.0519092	0.789773	0.66047	0.650208	0.647057
train_fn_[illegible]	TERMINATED	127.0.0.1:[illegible]	[illegible]e-05	6.23225	607.545	0.0178819	0.789773	0.556146	0.543529	0.547285
train_fn_[illegible]	TERMINATED	127.0.0.1:[illegible]	[illegible]e-05	10.7444	965.632	0.0810313	0.8125	0.585795	0.607136	0.586569
train_fn_[illegible]	TERMINATED	127.0.0.1:[illegible]	[illegible]e-05	6.48836	684.233	0.0445751	0.806818	0.578108	0.597877	0.580236

——+———+

(train_fn pid=5196) {'eval_accuracy': 0.8068181818181818, 'eval_precision': 0.5781076965669989, 'eval_recall': 0.5978766156185511, 'eval_f1': 0.5802364864864865, 'eval_loss': 0.5972080230712891, 'eval_runtime': 5.206, 'eval_samples_per_second': 33.807, 'eval_steps_per_second': 2.113, 'epoch': 6.5} Best hyperparameters found: {'num_train_epochs': 7.061695553391381, 'per_device_train_batch_size': 16, 'per_device_eval_batch_size': 16, 'warmup_steps': 181.82496720710063, 'weight_decay': 0.01915704647548995, 'learning_rate': 4.329770563201687e-05} Best trial metrics: {'eval_accuracy': 0.8409090909090909, 'eval_precision': 0.7521277980218636, 'eval_recall': 0.69963954056696, 'eval_f1': 0.7183126027230485, 'timestamp': 1719562210, 'checkpoint_dir_name': None, 'done': True, 'training_iteration': 2, 'trial_id': '06c2a1e8', 'date': '2024-06-28_10-10-10', 'time_this_iter_s': 0.0019960403442382812, 'time_total_s': 1395.4847493171692, 'pid': 21340, 'hostname': 'OpheliaPC', 'node_ip': '127.0.0.1', 'config': {'num_train_epochs': 7.061695553391381, 'per_device_train_batch_size': 16, 'per_device_eval_batch_size': 16, 'warmup_steps': 181.82496720710063, 'weight_decay': 0.01915704647548995, 'learning_rate': 4.329770563201687e-05}, 'time_since_restore': 1395.4847493171692, 'iterations_since_restore': 2, 'experiment_tag': '4_learning_rate=0.0000,num_train_epochs=7.0617,per_device_eval_batch_size=16,per_device_train_batch_size=16

[15]:
```
best_configurations
```

[15]:
```
[{'num_train_epochs': 9.925879806965067,
  'per_device_train_batch_size': 16,
  'per_device_eval_batch_size': 16,
  'warmup_steps': 199.67378215835973,
  'weight_decay': 0.051909209402947555,
  'learning_rate': 2.824279936868144e-05},
 {'num_train_epochs': 7.061695553391381,
  'per_device_train_batch_size': 16,
  'per_device_eval_batch_size': 16,
  'warmup_steps': 181.82496720710063,
  'weight_decay': 0.01915704647548995,
  'learning_rate': 4.329770563201687e-05},
 {'num_train_epochs': 7.061695553391381,
  'per_device_train_batch_size': 16,
```

```
 'per_device_eval_batch_size': 16,
 'warmup_steps': 181.82496720710063,
 'weight_decay': 0.01915704647548995,
 'learning_rate': 4.329770563201687e-05},
{'num_train_epochs': 9.925879806965067,
 'per_device_train_batch_size': 16,
 'per_device_eval_batch_size': 16,
 'warmup_steps': 199.67378215835973,
 'weight_decay': 0.051909209402947555,
 'learning_rate': 2.824279936868144e-05},
{'num_train_epochs': 7.061695553391381,
 'per_device_train_batch_size': 16,
 'per_device_eval_batch_size': 16,
 'warmup_steps': 181.82496720710063,
 'weight_decay': 0.01915704647548995,
 'learning_rate': 4.329770563201687e-05}]
```

[17]:
```
best_trial_metric_scores
```

[17]:
```
[{'eval_accuracy': 0.8352272727272727,
  'eval_precision': 0.8088219840269021,
  'eval_recall': 0.6868430930930931,
  'eval_f1': 0.72932843565081,
  'timestamp': 1719540391,
  'checkpoint_dir_name': None,
  'done': True,
  'training_iteration': 2,
  'trial_id': '3e29987b',
  'date': '2024-06-28_04-06-31',
  'time_this_iter_s': 0.002001047134399414,
  'time_total_s': 2199.600620508194,
  'pid': 33736,
  'hostname': 'OpheliaPC',
  'node_ip': '127.0.0.1',
  'config': {'num_train_epochs': 9.925879806965067,
   'per_device_train_batch_size': 16,
   'per_device_eval_batch_size': 16,
   'warmup_steps': 199.67378215835973,
   'weight_decay': 0.051909209402947555,
   'learning_rate': 2.824279936868144e-05},
  'time_since_restore': 2199.600620508194,
  'iterations_since_restore': 2,
  'experiment_tag': '7_learning_rate=0.0000,num_train_epochs=9.9259,per_device_e
val_batch_size=16,per_device_train_batch_size=16,warmup_steps=199.6738,weight_de
cay=0.0519'},
 {'eval_accuracy': 0.8238636363636364,
  'eval_precision': 0.7669057884825057,
```

```
  'eval_recall': 0.6476887868016901,
  'eval_f1': 0.6779614564467592,
  'timestamp': 1719543747,
  'checkpoint_dir_name': None,
  'done': True,
  'training_iteration': 2,
  'trial_id': 'a07e8a40',
  'date': '2024-06-28_05-02-27',
  'time_this_iter_s': 0.01622939109802246,
  'time_total_s': 1454.9051315784454,
  'pid': 22292,
  'hostname': 'OpheliaPC',
  'node_ip': '127.0.0.1',
  'config': {'num_train_epochs': 7.061695553391381,
   'per_device_train_batch_size': 16,
   'per_device_eval_batch_size': 16,
   'warmup_steps': 181.82496720710063,
   'weight_decay': 0.01915704647548995,
   'learning_rate': 4.329770563201687e-05},
  'time_since_restore': 1454.9051315784454,
  'iterations_since_restore': 2,
  'experiment_tag': '4_learning_rate=0.0000,num_train_epochs=7.0617,per_device_e
val_batch_size=16,per_device_train_batch_size=16,warmup_steps=181.8250,weight_de
cay=0.0192'},
 {'eval_accuracy': 0.8409090909090909,
  'eval_precision': 0.7734540050846851,
  'eval_recall': 0.7202589531218564,
  'eval_f1': 0.7295404752716816,
  'timestamp': 1719549995,
  'checkpoint_dir_name': None,
  'done': True,
  'training_iteration': 2,
  'trial_id': '13af170e',
  'date': '2024-06-28_06-46-35',
  'time_this_iter_s': 0.001007080078125,
  'time_total_s': 1526.5889155864716,
  'pid': 7248,
  'hostname': 'OpheliaPC',
  'node_ip': '127.0.0.1',
  'config': {'num_train_epochs': 7.061695553391381,
   'per_device_train_batch_size': 16,
   'per_device_eval_batch_size': 16,
   'warmup_steps': 181.82496720710063,
   'weight_decay': 0.01915704647548995,
   'learning_rate': 4.329770563201687e-05},
  'time_since_restore': 1526.5889155864716,
  'iterations_since_restore': 2,
```

```
  'experiment_tag': '4_learning_rate=0.0000,num_train_epochs=7.0617,per_device_e
val_batch_size=16,per_device_train_batch_size=16,warmup_steps=181.8250,weight_de
cay=0.0192'},
 {'eval_accuracy': 0.8409090909090909,
  'eval_precision': 0.7393939393939394,
  'eval_recall': 0.7526282042411074,
  'eval_f1': 0.7449354260935143,
  'timestamp': 1719558446,
  'checkpoint_dir_name': None,
  'done': True,
  'training_iteration': 2,
  'trial_id': '427a4215',
  'date': '2024-06-28_09-07-26',
  'time_this_iter_s': 0.010030984878540039,
  'time_total_s': 2258.306706905365,
  'pid': 22192,
  'hostname': 'OpheliaPC',
  'node_ip': '127.0.0.1',
  'config': {'num_train_epochs': 9.925879806965067,
   'per_device_train_batch_size': 16,
   'per_device_eval_batch_size': 16,
   'warmup_steps': 199.67378215835973,
   'weight_decay': 0.051909209402947555,
   'learning_rate': 2.824279936868144e-05},
  'time_since_restore': 2258.306706905365,
  'iterations_since_restore': 2,
  'experiment_tag': '7_learning_rate=0.0000,num_train_epochs=9.9259,per_device_e
val_batch_size=16,per_device_train_batch_size=16,warmup_steps=199.6738,weight_de
cay=0.0519'},
 {'eval_accuracy': 0.8409090909090909,
  'eval_precision': 0.7521277980218636,
  'eval_recall': 0.69963954056696,
  'eval_f1': 0.7183126027230485,
  'timestamp': 1719562210,
  'checkpoint_dir_name': None,
  'done': True,
  'training_iteration': 2,
  'trial_id': '06c2a1e8',
  'date': '2024-06-28_10-10-10',
  'time_this_iter_s': 0.0019960403442382812,
  'time_total_s': 1395.4847493171692,
  'pid': 21340,
  'hostname': 'OpheliaPC',
  'node_ip': '127.0.0.1',
  'config': {'num_train_epochs': 7.061695553391381,
   'per_device_train_batch_size': 16,
   'per_device_eval_batch_size': 16,
```

```
  'warmup_steps': 181.82496720710063,
  'weight_decay': 0.01915704647548995,
  'learning_rate': 4.329770563201687e-05},
 'time_since_restore': 1395.4847493171692,
 'iterations_since_restore': 2,
 'experiment_tag': '4_learning_rate=0.0000,num_train_epochs=7.0617,per_device_e
val_batch_size=16,per_device_train_batch_size=16,warmup_steps=181.8250,weight_de
cay=0.0192'}]
```

[18]:
```
## Important: this code took all night to run, so save the best hyperparameters␣
↪scores!

# import pickle
# with open('best_hyperparam_dicts_for_cross_val.pkl', 'wb') as f:
#     pickle.dump(best_configurations, f)

# with open('best_metric_scores_for_cross_val.pkl', 'wb') as f:
#     pickle.dump(best_trial_metric_scores, f)
```

[9]:
```
# Unpickle the dicts generated from Bayesian Optimization Five Fold␣
↪Hyperparameters Searches
with open('best_hyperparam_dicts_for_cross_val.pkl', 'rb') as f:
    best_hyperparam_combos_for_each_fold = pickle.load(f)

print("\t\t\t\t\t\t\t\tBest hyperparameter combinations per fold:")
for idx, hyperparams in enumerate(best_hyperparam_combos_for_each_fold):
    print(f"Fold {idx + 1}: {hyperparams}\n")
print("\n******************************************************************************

with open('best_metric_scores_for_cross_val.pkl', 'rb') as f:
    best_metric_scores_for_each_fold = pickle.load(f)

print("\t\t\t\t\t\t\t\tBest metric scores per fold:")
for idx, metric_dict in enumerate(best_metric_scores_for_each_fold):
    print(f"Fold {idx + 1}: {metric_dict}\n")
```

```
                                                        Best hyperparameter
combinations per fold:
Fold 1: {'num_train_epochs': 9.925879806965067, 'per_device_train_batch_size':
16, 'per_device_eval_batch_size': 16, 'warmup_steps': 199.67378215835973,
'weight_decay': 0.051909209402947555, 'learning_rate': 2.824279936868144e-05}

Fold 2: {'num_train_epochs': 7.061695553391381, 'per_device_train_batch_size':
16, 'per_device_eval_batch_size': 16, 'warmup_steps': 181.82496720710063,
'weight_decay': 0.01915704647548995, 'learning_rate': 4.329770563201687e-05}

Fold 3: {'num_train_epochs': 7.061695553391381, 'per_device_train_batch_size':
```

16, 'per_device_eval_batch_size': 16, 'warmup_steps': 181.82496720710063, 'weight_decay': 0.01915704647548995, 'learning_rate': 4.329770563201687e-05}

Fold 4: {'num_train_epochs': 9.925879806965067, 'per_device_train_batch_size': 16, 'per_device_eval_batch_size': 16, 'warmup_steps': 199.67378215835973, 'weight_decay': 0.05190920940294755, 'learning_rate': 2.824279936868144e-05}

Fold 5: {'num_train_epochs': 7.061695553391381, 'per_device_train_batch_size': 16, 'per_device_eval_batch_size': 16, 'warmup_steps': 181.82496720710063, 'weight_decay': 0.01915704647548995, 'learning_rate': 4.329770563201687e-05}

****

Best metric scores per fold:
Fold 1: {'eval_accuracy': 0.8352272727272727, 'eval_precision': 0.8088219840269021, 'eval_recall': 0.6868430930930931, 'eval_f1': 0.72932843565081, 'timestamp': 1719540391, 'checkpoint_dir_name': None, 'done': True, 'training_iteration': 2, 'trial_id': '3e29987b', 'date': '2024-06-28_04-06-31', 'time_this_iter_s': 0.002001047134399414, 'time_total_s': 2199.600620508194, 'pid': 33736, 'hostname': 'OpheliaPC', 'node_ip': '127.0.0.1', 'config': {'num_train_epochs': 9.925879806965067, 'per_device_train_batch_size': 16, 'per_device_eval_batch_size': 16, 'warmup_steps': 199.67378215835973, 'weight_decay': 0.05190920940294755, 'learning_rate': 2.824279936868144e-05}, 'time_since_restore': 2199.600620508194, 'iterations_since_restore': 2, 'experiment_tag': '7_learning_rate=0.0000,num_train_epochs=9.9259,per_device_eval_batch_size=16,per_device_train_batch_size=16,warmup_steps=199.6738,weight_decay=0.0519'}

Fold 2: {'eval_accuracy': 0.8238636363636364, 'eval_precision': 0.7669057884825057, 'eval_recall': 0.6476887868016901, 'eval_f1': 0.6779614564467592, 'timestamp': 1719543747, 'checkpoint_dir_name': None, 'done': True, 'training_iteration': 2, 'trial_id': 'a07e8a40', 'date': '2024-06-28_05-02-27', 'time_this_iter_s': 0.01622939109802246, 'time_total_s': 1454.9051315784454, 'pid': 22292, 'hostname': 'OpheliaPC', 'node_ip': '127.0.0.1', 'config': {'num_train_epochs': 7.061695553391381, 'per_device_train_batch_size': 16, 'per_device_eval_batch_size': 16, 'warmup_steps': 181.82496720710063, 'weight_decay': 0.01915704647548995, 'learning_rate': 4.329770563201687e-05}, 'time_since_restore': 1454.9051315784454, 'iterations_since_restore': 2, 'experiment_tag': '4_learning_rate=0.0000,num_train_epochs=7.0617,per_device_eval_batch_size=16,per_device_train_batch_size=16,warmup_steps=181.8250,weight_decay=0.0192'}

Fold 3: {'eval_accuracy': 0.8409090909090909, 'eval_precision': 0.7734540050846851, 'eval_recall': 0.7202589531218564, 'eval_f1': 0.7295404752716816, 'timestamp': 1719549995, 'checkpoint_dir_name': None, 'done': True, 'training_iteration': 2, 'trial_id': '13af170e', 'date': '2024-06-28_06-46-35', 'time_this_iter_s': 0.001007080078125, 'time_total_s': 1526.5889155864716, 'pid': 7248, 'hostname': 'OpheliaPC', 'node_ip': '127.0.0.1', 'config': {'num_train_epochs': 7.061695553391381, 'per_device_train_batch_size': 16, 'per_device_eval_batch_size': 16, 'warmup_steps': 181.82496720710063, 'weight_decay': 0.01915704647548995, 'learning_rate': 4.329770563201687e-05}, 'time_since_restore': 1526.5889155864716, 'iterations_since_restore': 2, 'experiment_tag': '4_learning_rate=0.0000,num_train_epochs=7.0617,per_device_eval_batch_size=16,per_device_train_batch_size=16,warmup_steps=181.8250,weight_decay=0.0192'}

Fold 4: {'eval_accuracy': 0.8409090909090909, 'eval_precision': 0.7393939393939394, 'eval_recall': 0.7526282042411074, 'eval_f1': 0.7449354260935143, 'timestamp': 1719558446, 'checkpoint_dir_name': None, 'done': True, 'training_iteration': 2, 'trial_id': '427a4215', 'date': '2024-06-28_09-07-26', 'time_this_iter_s': 0.010030984878540039, 'time_total_s': 2258.306706905365, 'pid': 22192, 'hostname': 'OpheliaPC', 'node_ip': '127.0.0.1', 'config': {'num_train_epochs': 9.925879806965067, 'per_device_train_batch_size': 16, 'per_device_eval_batch_size': 16, 'warmup_steps': 199.67378215835973, 'weight_decay': 0.051909209402947555, 'learning_rate': 2.824279936868144e-05}, 'time_since_restore': 2258.306706905365, 'iterations_since_restore': 2, 'experiment_tag': '7_learning_rate=0.0000,num_train_epochs=9.9259,per_device_eval_batch_size=16,per_device_train_batch_size=16,warmup_steps=199.6738,weight_decay=0.0519'}

Fold 5: {'eval_accuracy': 0.8409090909090909, 'eval_precision': 0.7521277980218636, 'eval_recall': 0.69963954056696, 'eval_f1': 0.7183126027230485, 'timestamp': 1719562210, 'checkpoint_dir_name': None, 'done': True, 'training_iteration': 2, 'trial_id': '06c2a1e8', 'date': '2024-06-28_10-10-10', 'time_this_iter_s': 0.0019960403442382812, 'time_total_s': 1395.4847493171692, 'pid': 21340, 'hostname': 'OpheliaPC', 'node_ip': '127.0.0.1', 'config': {'num_train_epochs': 7.061695553391381, 'per_device_train_batch_size': 16, 'per_device_eval_batch_size': 16, 'warmup_steps': 181.82496720710063, 'weight_decay': 0.01915704647548995, 'learning_rate': 4.329770563201687e-05}, 'time_since_restore': 1395.4847493171692, 'iterations_since_restore': 2, 'experiment_tag': '4_learning_rate=0.0000,num_train_epochs=7.0617,per_device_eval_batch_size=16,per_device_train_batch_size=16,warmup_steps=181.8250,weight_decay=0.0192'}

The best performance occurred on Fold 4, achieving an accuracy of 84% and average F1 of 74%. The optimal hyperparameters for this fold, found using Bayesian Optimization, were 9.9 training epochs, 199.7 warmup steps, weight decay of 0.05 and learning rate of 2.824279936868144e-05.

To identify the best hyperparameters to use for training the model on the whole training set (and then evaluating on the test set), we could take the average of each hyperparameter - however, due to significant differences between optimal hyperparameter values (e.g. epochs) on folds, taking the mean of each hyperparameter would lead to completely different values, which could lead to sub-optimal performance. Therefore, the optimal hyperparameters found for the best-performing fold (fold 4) will be taken to train the model on the entire training set, before evaluating on the test set.

2.12 Testing DistilBERT Model on the Recombined Test Set

[10]:
```
## Training the model on the whole dataset using hyperparameters found using␣
↪Bayesian Optimization

hyperparams = best_hyperparam_combos_for_each_fold[3] # get hyperparams from␣
↪4th fold
print(hyperparams)
# Round num of epochs to train for, as this should be an integer
num_train_epochs = round(hyperparams["num_train_epochs"])
per_device_train_batch_size = hyperparams["per_device_train_batch_size"]
per_device_eval_batch_size = hyperparams["per_device_eval_batch_size"]
# Round num of warmup steps, as this also has to be an integer
warmup_steps = round(hyperparams["warmup_steps"])
weight_decay = hyperparams["weight_decay"]
learning_rate = hyperparams["learning_rate"]
```

```
{'num_train_epochs': 9.925879806965067, 'per_device_train_batch_size': 16,
'per_device_eval_batch_size': 16, 'warmup_steps': 199.67378215835973,
'weight_decay': 0.051909209402947555, 'learning_rate': 2.824279936868144e-05}
```

[11]:
```
# Now train the DistilBERT model AGAIN on all of the training data and evaluate␣
↪on test set
model = DistilBertForSequenceClassification.
↪from_pretrained('distilbert-base-uncased', num_labels=4)

# Define training arguments by using the optimum hyperparameters found
training_args = TrainingArguments(
    output_dir='./recombined_dataset_whole_training_results',  # output␣
↪directory
    num_train_epochs=num_train_epochs,              # total number of training␣
↪epochs
    per_device_train_batch_size=per_device_train_batch_size,  # batch size per␣
↪device during training
    per_device_eval_batch_size=per_device_eval_batch_size,   # batch size for␣
↪evaluation
    warmup_steps=warmup_steps,                # number of warmup steps for␣
↪learning rate scheduler
    weight_decay=weight_decay,               # strength of weight decay
    logging_dir='./recombined_dataset_whole_training_logs',            #␣
↪directory for storing logs
    logging_steps=10,
    evaluation_strategy="no",  # No evaluation during training, as we're␣
↪training on the entire training data set
```

```
    learning_rate=learning_rate
)

trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=recombined_train_dataset, # train on the whole training set␣
 ↪now with the optimal parameters
    eval_dataset=None,  # No evaluation during training, will eval on testset␣
 ↪later
    compute_metrics=compute_metrics  # Include compute_metrics here
)

trainer.train()
```

```
Some weights of DistilBertForSequenceClassification were not initialized from
the model checkpoint at distilbert-base-uncased and are newly initialized:
['classifier.bias', 'classifier.weight', 'pre_classifier.bias',
'pre_classifier.weight']
You should probably TRAIN this model on a down-stream task to be able to use it
for predictions and inference.
C:\Users\ophel\AppData\Local\Programs\Python\Python38\lib\site-
packages\transformers\training_args.py:1474: FutureWarning:
`evaluation_strategy` is deprecated and will be removed in version 4.46 of
Transformers. Use `eval_strategy` instead
  warnings.warn(
```

```
<IPython.core.display.HTML object>
```

[11]:
```
TrainOutput(global_step=550, training_loss=0.37369350039146165,
metrics={'train_runtime': 967.8715, 'train_samples_per_second': 9.092,
'train_steps_per_second': 0.568, 'total_flos': 61475344761600.0, 'train_loss':
0.37369350039146165, 'epoch': 10.0})
```

[14]:
```
# Evaluate the model on the test dataset
eval_results = trainer.evaluate(eval_dataset=recombined_test_dataset)
print(eval_results)

# Get predictions
predictions = trainer.predict(test_dataset=recombined_test_dataset)
predicted_labels = np.argmax(predictions.predictions, axis=1)
true_labels = recombined_test_dataset.labels

true_labels = np.array(true_labels)
predicted_labels = np.array(predicted_labels)
```

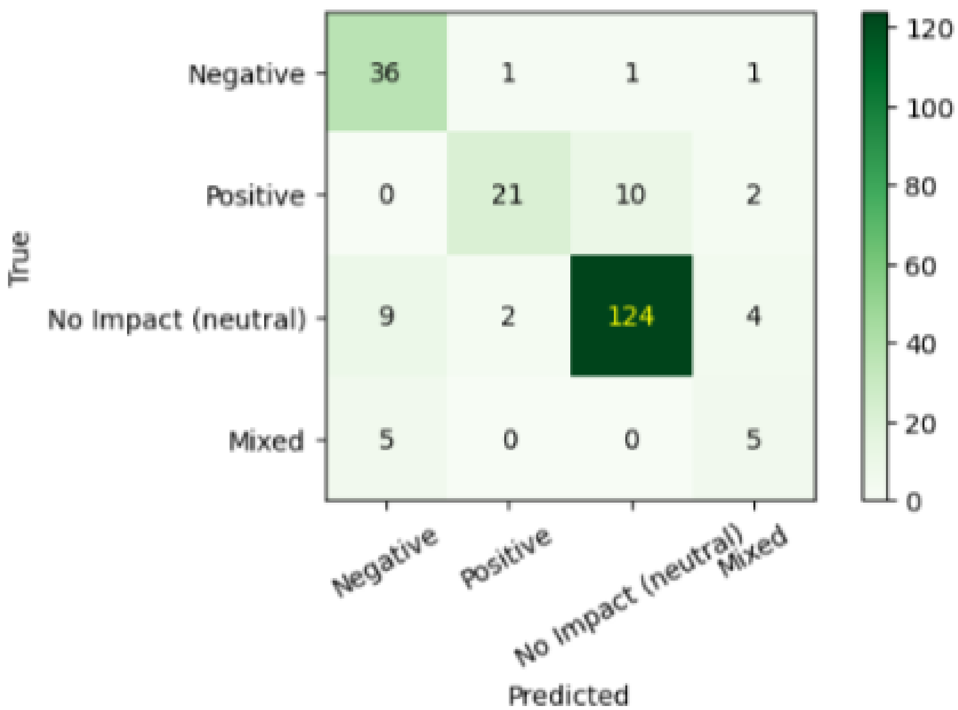
```
<IPython.core.display.HTML object>
```

```
{'eval_accuracy': 0.8416289592760181, 'eval_precision': 0.7325462962962962,
'eval_recall': 0.7378817225939529, 'eval_f1': 0.7263714532261628, 'eval_loss':
0.6726983785629272, 'eval_runtime': 3.9867, 'eval_samples_per_second': 55.434,
'eval_steps_per_second': 3.512, 'epoch': 10.0}
```

[15]:
```
# Print the classification report to view the precision, recall, f1 score for␣
↪each class and macro-average scores
print("CLASSIFICATION REPORT:\n")
print(classification_report(
    true_labels,
    predicted_labels,
    target_names=label_names)
)
```

```
CLASSIFICATION REPORT:

                     precision    recall  f1-score   support

           Negative       0.72      0.92      0.81        39
           Positive       0.88      0.64      0.74        33
No Impact (neutral)       0.92      0.89      0.91       139
              Mixed       0.42      0.50      0.45        10

           accuracy                           0.84       221
          macro avg       0.73      0.74      0.73       221
       weighted avg       0.85      0.84      0.84       221
```

[12]:
```
generate_and_show_confusion_matrix(
        true_labels, predicted_labels,
        label_names,
        # By accident, I forgot to replace the title of the confusion matrix␣
↪but it would have taken a very long time to
        # the model again, so I saved the image of the confusion matrix and␣
↪display it in a markdown cell below this.
        # the title should have been this instead.
        "Performance of DistilBERT Classifier on Recombined Test Split",
        matrix_color=plt.cm.Greens, # default: green colour-coded c matrix
        above_threshold_text_color="yellow" # color in which to show the text␣
↪of counts above the threshold
    )
```



Performance of DistilBERT Classifier on Recombined Test Split

[19]:
```
# Print each sample/text with true and predicted labels where misclassified␣
↪example
print("\nERROR ANALYSIS:\n")
for i, (text, true_label, pred_label) in enumerate(zip(test_texts, true_labels,␣
↪predicted_labels)):
    if true_label != pred_label:
        print(f"Sample {i + 1}:")
        print(f"Text: {text}")
        print(f"True Label: {label_names[true_label]}")
        print(f"Predicted Label: {label_names[pred_label]}\n")
```

```
ERROR ANALYSIS:

Sample 3:
Text: and when they reached the strait symplegades
True Label: Positive
Predicted Label: No Impact (neutral)

Sample 10:
Text: of the boulder-strewn mountain, and when they will crop.
True Label: No Impact (neutral)
Predicted Label: Negative
```

```
Sample 18:
Text: for penance, by a saintly styrian monk
True Label: Positive
Predicted Label: Mixed

Sample 19:
Text: upon a mountain crag, young angelo--
True Label: No Impact (neutral)
Predicted Label: Positive

Sample 29:
Text: those wastes of frozen billows that were hurled
True Label: No Impact (neutral)
Predicted Label: Negative

Sample 32:
Text: bertram finished the last pages, while along the silence ever
True Label: Negative
Predicted Label: Positive

Sample 38:
Text: of robert burns and alexander.
True Label: Mixed
Predicted Label: Negative

Sample 44:
Text: the life of love that gave it--settles.
True Label: Positive
Predicted Label: No Impact (neutral)

Sample 59:
Text: he might have been as doubtful once
True Label: Mixed
Predicted Label: Negative

Sample 61:
Text: a fleecy cloud,
True Label: Positive
Predicted Label: No Impact (neutral)

Sample 62:
Text: shall dwell in the house of my fathers and the land of the people's
praise;
True Label: Mixed
Predicted Label: Negative

Sample 85:
Text: with the whole world gone blind,
```

```
True Label: No Impact (neutral)
Predicted Label: Mixed

Sample 86:
Text: and, wildly tossed from cheeks and chin,
True Label: Positive
Predicted Label: No Impact (neutral)

Sample 88:
Text: the night had found (to him a night of wo)
True Label: Positive
Predicted Label: No Impact (neutral)

Sample 89:
Text: but, fixing on the horrid maid his eye,
True Label: No Impact (neutral)
Predicted Label: Negative

Sample 94:
Text: shall troy renew'd be forc'd and fir'd again?
True Label: No Impact (neutral)
Predicted Label: Mixed

Sample 98:
Text: thus while the trojan prince employs his eyes,
True Label: Positive
Predicted Label: No Impact (neutral)
```

2.12.1 DistilBERT Performance on Recombined Testset

The confusion matrix for the recombined dataset (with mixed sentiment samples) shows that the class that was most frequently misclassified was the "positive" sentiment class, with 10 out of 31 positive samples being classed as "no impact (neutral)". An error analysis is printed above to show the difference in predicted and true labels for misclassified samples. This error analysis demonstrates several key reasons for why positive samples were so frequently misclassified as neutral, and clarifies some of the challenges inherent in using this dataset for this sentiment analysis task.

First, as the dataset contains samples of poetic language, some of the verse texts contain unusual words, such as "symplegades" (a reference to a pair of rocks from Greek mythology). Even a large pre-trained model such as DistilBERT probably did not have access to many texts featuring this kind of vocabulary, hence the difficulty in recognizing patterns for texts containing such niche terms. Perhaps the meaning in the Greek text of this landscape was positive, hence the positive annotation, but it would be difficult for a deep-learning classifier not trained on ancient Greek texts to identify these connotations.

Second, a larger issue seems to be with the quality of the annotations for some of the samples in this dataset. Despite a couple of misclassifications being classed as positive but predicted as neutral, such as "the life of love that gave it–settles", the error analysis raises some issues about the suitability of the “true” label selected by the annotators. An examples of this is the line “but, fixing on the horrid maid his eye”, which was annotated as “neutral” but predicted as “negative”. This line of verse seems to contain an adjective (“horrid”) describing a specific entity (the maid), thus it seems like the connotations of this text should have been labelled as negative - in fact, the DistilBERT model arguably corrected this error by classing the sample as “negative”. Another example is “thus while the trojan prince employs his eyes” which has been annotated as “positive”, but it is unclear what the rationale would be, as “neutral” appears to be a more fitting description of this text. This could be due to the inherent properties of the dataset: extracting merely a single line of poetry out of a work of literature makes it difficult to assess whether the sentiment of a text is truly negative, positive, or neutral. This highlights the need for a publicly-annotated poetry dataset for sentiment or emotion analysis containing a much greater number of samples and longer texts. Furthermore, the frequent references to ancient Greek or Roman mythology in English poetry exposes another challenge for any classifier: one would have to train it on examples of texts from classical literature and history to truly be able to grasp the sentimental connotations of this kind of domain specific language. Moreover, the subjective nature of poetic language can also lead to annotations that do not truly reflect the sentiment expressed in just one line of verse: the annotator might know that a certain landmark or person (e.g. a Trojan primce) might have positive connotations in literature, or that a line has been extracted from a poem that is overall positive, thus affecting their judgement - the actual line itself might be more neutral in its polarity, with the additional real-world contextual knowledge of the annotator is not reflected by it.

Another challenge is highlighted by examining the misclassification of “of robert burns and alexander.” The “true” label is “mixed” but the classifier has predicted the class as “negative”. Possibly, this may be due to the ambiguity of the word “burns” and the fact that the original dataset texts were all already lower-cased. In lower case, “burns” is a verb that frequently carries with it negative connotations, but the original text, where this would have been written in upper case, references a poet whose surname is “Burns”. This example of word-level ambiguity clearly showcases one of the key challenges faced in Natural Language Processing.

Additionally, the small size of the “mixed” sample class means that it is very challenging for *any* statistical classifer to learn enough patterns to classify these samples correctly.

Overall, despite the significant limitations of this dataset and the challenges outlined above, the DistilBERT classifier that was fine-tuned on this dataset was still surprisingly effective, achieving a 81% macro-average F1-score on negative samples and 74% on positive samples. This transformer-based model clearly outperforms the traditional Multinomial Naive Bayes classification algorithm on this dataset, on both the original dataset splits and the recombined dataset. In the following section, I will discuss and critically evaluate the difference between the performance of both approaches.

2.13 Results Table: Best Scores for Each Dataset Split and Classification Algorithm

Classifier	Dataset Split	Accuracy	Macro-Avg Recall	Macro-Avg Precision	Macro-Avg F1-Score
Baseline Naive Bayes (evaluated on val set)	Original (train-val-test)	0.73	0.50	0.79	0.53
MNB Classifier using WordNet Negation Handling and Senti-WordNet (evaluated on test set)	Original (train-val-test)	0.71	0.46	0.66	0.48
MNB Classifier using WordNet Negation Handling and Senti-WordNet (evaluated on test set)	Recombined (train-test) Dataset using Stratified Cross-Validation and Mixed Sentiment Scores	0.67	0.40	0.42	0.40
DistilBERT Classifier (evaluated on test set)	Original (train-val-test)	0.89	0.64	0.66	0.65
DistilBERT Classifier (evaluated on test set)	Recombined (train-test) Dataset using Stratified Cross-Validation and Mixed Sentiment Scores	0.84	0.74	0.73	0.73

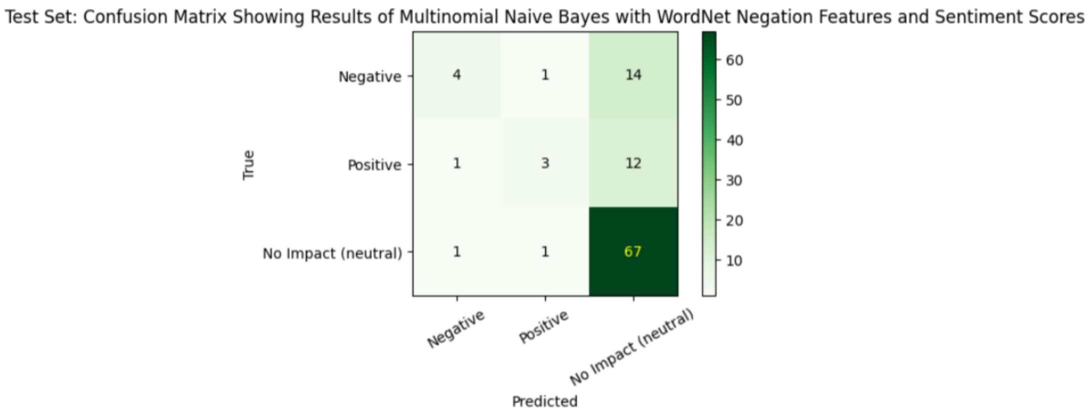
3 III. Conclusions

3.1 Performance Analysis & Comparative Discussion

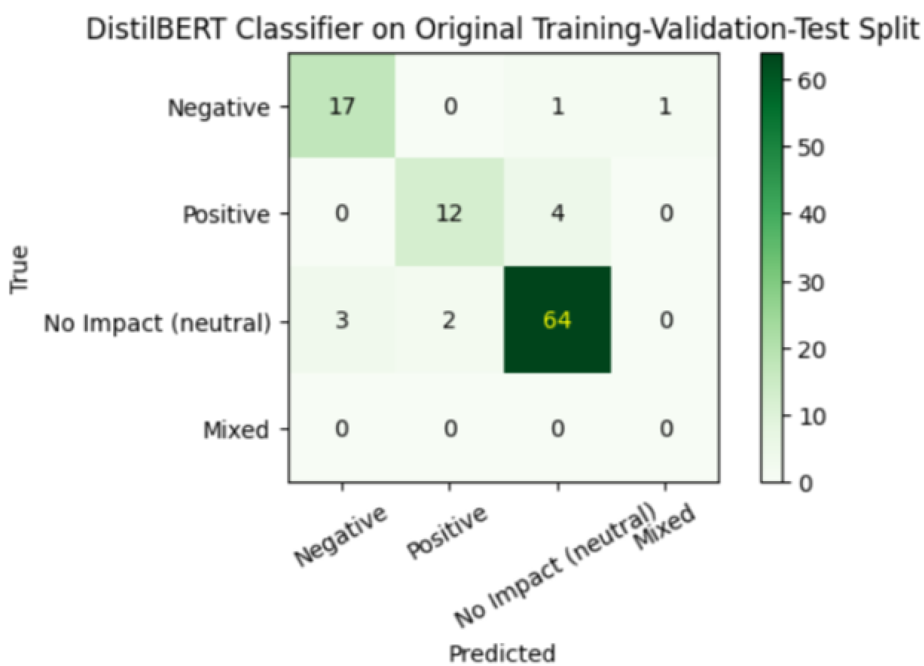
In short, DistilBERT performed substantially better than Multinomial Naive Bayes.

3.1.1 Performance on Original Test Set

The highest accuracy score achieved using Naive Bayes was 73%: only marginally higher than if predicting the majority class (66%). The best macro-average F1-score achieved was 53%. The confusion matrix shows that the model classified most of the positive and negative samples as "neutral":



Extensive feature-engineering experiments failed to increase the performance of Naïve Bayes. In contrast, DistilBERT obtained an accuracy score of 89% and a macro-average F1-score of 65%.

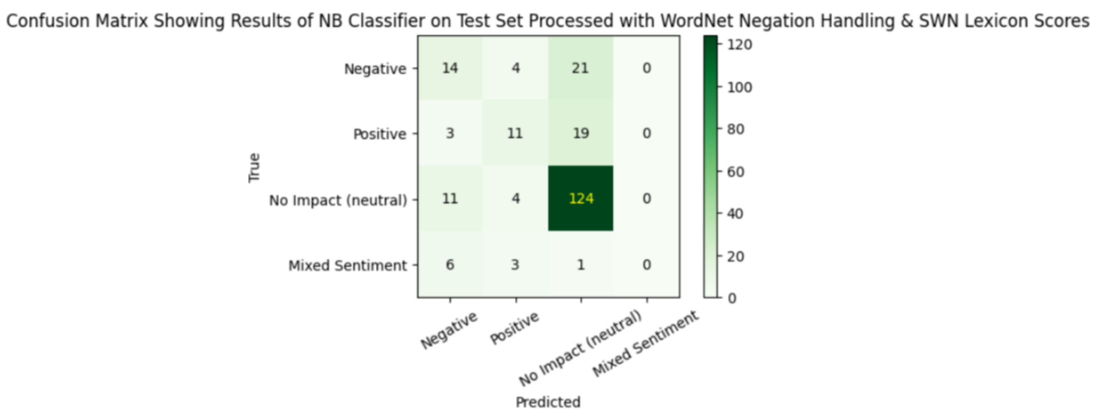


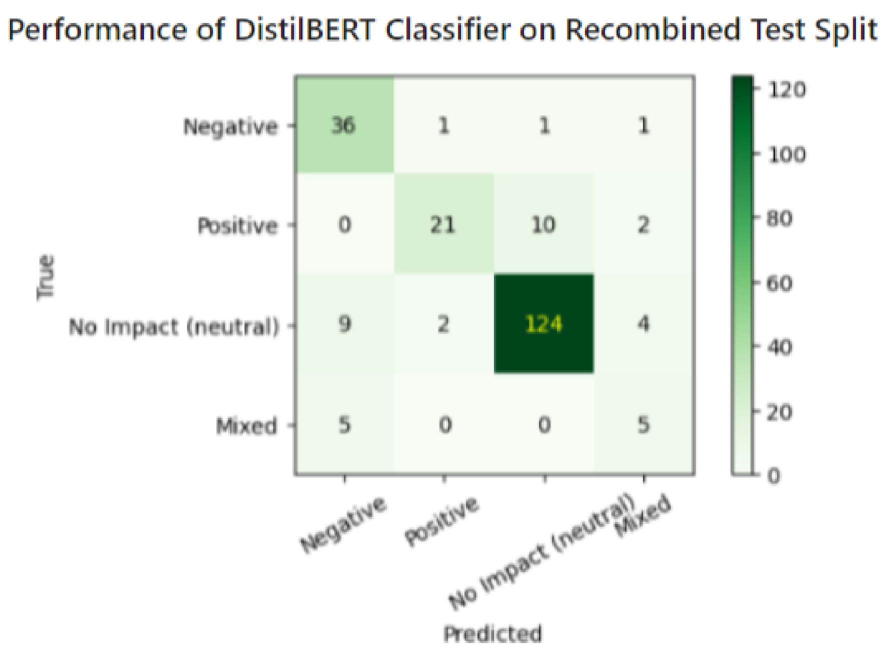
The confusion matrix shows that DistilBERT performed particularly well predicting the correct labels for the negative and neutral classes. The accuracy score on the original test set of 89% even outperformed the original authors' accuracy of 84.6% and reached the same value as AIManatee's accuracy score using RoBERTa, despite achieving a much lower F1-score (AIManatee achieved a 90%, but does not explain if this is a micro or macro average).

3.1.2 Performance on the Recombined Dataset using Five-Fold Cross-Validation for Model Optimization

DistilBERT also performed much better on the recombined dataset, achieving 84% accuracy and 73% F1-score, while the best-performing Naive Bayes classifier obtained only 67% accuracy and 40% F1. The confusion matrix shows that Naïve Bayes failed to identify even half of the negative or positive samples correctly

Interestingly, DistilBERT performed better (73% vs. 65% F1-score) on the recombined test set containing "mixed" samples compared to the original test set. This suggests that experimenting with different hyperparameter settings on the same validation set may have led to overfitting. .





Interestingly, DistilBERT performed better overall (73% vs. 65% F1-score) on the recombined test set containing "mixed" samples compared to the original test set. This difference suggests that experimenting with different hyperparameter settings on the same, original validation set may have led to overfitting.

3.1.3 Comparisons Between Models

One advantage of DistilBERT is that the attention-mechanism reflects the *context* of each token by considering the relevance of the previous and the subsequent words. While Naïve Bayes is simpler and faster, it is limited in its ability to model complex *relationships* between the words. As such, it may not be appropriate for cases such as literary sentiment analysis, as already argued by Da (2019), who maintains that "detecting patterns based in word counts" is a fairly limited approach for this use case. Moreover, Naive Bayes makes predictions by modelling the probabilities (likelihoods) of word occurrences as *individual* features for a specific class, working on the assumption that each feature is independent. However, occurrences of words (features) in a text are rarely independent from the occurrences of other words.

While the performance of the deep-learning model on even this small, unbalanced dataset is reasonably good, it is generally more difficult to interpret why a transformer-based network makes certain decisions (Tunstall, von Werra, & Wolf, 2022). As Kim and Klinger argue (2019), in the humanities, the *interpretability* of a model is paramount. In this context, statistical models may be preferable despite their lower accuracy.

Nevertheless, in a business scenario oriented around modelling the nuances of figurative language in customer reviews, training a transformer-based model on literary and poetic texts may be useful, as performance is usually prioritized over interpretability.lity.

W.C.: 494

3.2 Project Summary and Reflections

By comparing the performance of a Naive Bayes classifier to that of a DistilBERT in detecting the sentiment polarity of poetry samples, this project highlights the trade-off between the speed and interpretability of a traditional statistical model, and the excellent predictive power of neural networks. The models' performance on the original dataset was compared to that on a stratified dataset to address class imbalance, using a random_state input argument, to ensure the reproducibility of results. This study also emphasises the importance of evaluating classifier performance using metrics such as F1-score as well as accuracy: the original authors who used this dataset to address societal bias in a poetry-generation program boast of an accuracy score of 84.6% (Sheng & Uthus, 2020) but this can be an overly optimistic evaluation of performance given the unbalanced nature of the dataset.

As with other neural networks, "transformers are to a large extent opaque" (Tunstall, von Werra, & Wolf, 2022). In the digital humanities, the model's interpretability is crucial. Despite this, transformers could still be used to corroborate existing theses - for instance, to verify a theory that "Romantic" poetry contains more expressions of negative sentiment than "Victorian" poetry. The power of transformers for text classification partly refutes Da's argument (2019): that computational methods should be disregarded in the humanities, as they rely on basic word counts while dismissing complex contextual dependencies.

Moreover, evaluating the performance of a BERT-like model on the sentiment analysis of poetic texts can provide a benchmark for measuring the ability of transformer-based models to handle extremely subjective, ambiguous and indirect language. Evaluating performance on this kind of "difficult" language can thus improve the robustness of classifiers when dealing with forum posts or customer reviews, which also (albeit less frequently) contain this kind of language.

An important limitation was the scarcity of available datasets for the sentiment analysis of poetic language. Not only did this dataset contain extremely small numbers of specific classes, error analysis revealed that some of the annotators' labels seemed inconsistent. Additionally, the poetry samples in this dataset were extremely short, and it can be difficult to ascertain the sentiment polarity without additional context. A valuable research direction would be improving the quantity of publicly-available poetry data. One could start either by adding more samples to this dataset or augmenting the dataset using back-translation (Tunstall, von Werra, & Wolf, 2022).

W.C.: 397 words

---

4 IV. References

AiManatee. (2024). RoBERTa_poem_sentiment [Machine learning model]. Hugging Face. Retrieved June 15, 2024, from https://huggingface.co/AiManatee/RoBERTa_poem_sentiment

Akbani, R., Kwek, S., & Japkowicz, N. (2004). Applying Support Vector Machines to Imbalanced Data Sets. Lecture Notes in Artificial Intelligence, 3201, 39-50. https://doi.org/10.1007/978-3-540-30115-8_7

Bahrawi. (2019). Sentiment analysis using random forest algorithm-online social media based. Journal of Information Technology and Its Utilization, 2, 29. https://doi.org/10.30818/jitu.2.2.2695

Bird, S., Klein, E., & Loper, E. (2019). Natural language processing with Python. O'Reilly Media, Inc. Retrieved June 15, 2024, from https://www.nltk.org/book/

Bramer, M. (2007). Avoiding Overfitting of Decision Trees. In M. Bramer (Ed.), Principles of Data Mining (pp. 119-134). Springer London. https://doi.org/10.1007/978-1-84628-766-4_8

Chaurasia, A. (2023, January 4). Hyperparameter optimization for HuggingFace Transformers. Weights & Biases. Retrieved June 25, 2024, from https://wandb.ai/amogkam/transformers/reports/Hyperparameter-Optimization-for-HuggingFace-Transformers–VmlldzoyMTc2ODI#bayesian-search-with-asynchronous-hyperopt

CodeEmporium. (2020, January 13). Transformer neural networks - EXPLAINED! (Attention is all you need) [Video]. YouTube. Retrieved June 29, 2024, from https://www.youtube.com/watch?v=TQQlZhbC5ps

Creative Commons. (n.d.). Attribution 4.0 International (CC BY 4.0). Retrieved June 18, 2024, from https://creativecommons.org/licenses/by/4.0/

Da, N. Z. (2019). The computational case against computational literary studies. Critical Inquiry, 45(3), 601-639. https://doi.org/10.1086/702594

Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2019). BERT: Pre-training of deep bidirectional transformers for language understanding. arXiv. https://arxiv.org/abs/1810.04805

Evidently AI Team. (2024). Accuracy, precision, and recall in multi-class classification. Evidently AI. Retrieved June 18, 2024, from https://www.evidentlyai.com/classification-metrics/multi-class-metrics#:~:text=Macro%2Daveraging%20gives%20equal%20weight,same%20and%20identical%20to%20accuracy.

GeeksforGeeks. (2024, February 16). In the context of deep learning, what is training warmup steps? GeeksforGeeks. Retrieved June 29, 2024, from https://www.geeksforgeeks.org/in-the-context-of-deep-learning-what-is-training-warmup-steps/

GeeksForGeeks. (2020, February 19). Upper confidence bound algorithm in reinforcement learning. GeeksForGeeks. Retrieved June 29, 2024, from https://www.geeksforgeeks.org/upper-confidence-bound-algorithm-in-reinforcement-learning/

Hugging Face. (n.d.). Fine-tuning with custom datasets. Hugging Face Transformers Documentation. Retrieved June 25, 2024, from https://huggingface.co/transformers/v3.4.0/custom_datasets.html

Hugging Face. (n.d.). Hyperparameter Search using Trainer API. HuggingFace. Retrieved June 25, 2024, from https://huggingface.co/docs/transformers/en/hpo_train

Hugging Face. (n.d.). Introduction to NLP with Hugging Face. Hugging Face. Retrieved June 15, 2024, from https://huggingface.co/learn/nlp-course/en/chapter1/4

Ilmawan, L. B., Muladi, M., & Prasetya, D. D. (2024). Negation handling for sentiment analysis task: Approaches and performance analysis. International Journal of Electrical and Computer Engineering (IJECE), 14(3), 3382-3393. https://doi.org/10.11591/ijece.v14i3.pp3382-3393

Jain, Y. (2022, February 22). Lemmatization [NLP, Python]. Medium. Retrieved June 23, 2024,from https://medium.com/@yashj302/lemmatization-f134b3089429.

Jayaswal, V. (2020, November 22). Laplace smoothing in Naïve Bayes algorithm: Solving the zero probability problem in Naïve Bayes algorithm. Towards Data Science. Retrieved June 20, 2024, from https://towardsdatascience.com/laplace-smoothing-in-na%C3%AFve-bayes-algorithm-9c237a8bdece.

Jurafsky, D., & Martin, J. H. (2024). Speech and language processing (3rd ed. draft, February 3, 2024). Online Draft. Stanford University. Retrieved June 15, 2024, from https://web.stanford.edu/~jurafsky/slp3/

Kamsetty, A., Fricke, K., & Liaw, R. (2020, August 26). Hyperparameter optimization for Transformers: A guide. Distributed Computing with Ray. Medium. Retrieved June 25, 2024, from https://medium.com/distributed-computing-with-ray/hyperparameter-optimization-for-transformers-a-guide-c4e32c6c989b

Kim, E., & Klinger, R. (2019). A survey on sentiment and emotion analysis for computational literary studies. Zeitschrift für digitale Geisteswissenschaften, Herzog August Bibliothek. Retrieved June 15, 2024, from https://zfdg.de/2019_008_v1

Köbis, N., & Mossink, L. D. (2021). Artificial intelligence versus Maya Angelou: Experimental evidence that people cannot differentiate AI-generated from human-written poetry. Computers in Human Behavior, 114, 106553. https://doi.org/10.1016/j.chb.2020.106553

Lal, U. (2022, May 27). Increasing accuracy of sentiment classification using negation handling: A novel approach towards increasing accuracy of sentiment classification quickly and efficiently.

Towards Data Science. Retrieved June 18, 2024, from https://towardsdatascience.com/increasing-accuracy-of-sentiment-classification-using-negation-handling-5c962d79c1f3

Lal, U., & Kamath, P. (2022). Effective negation handling approach for sentiment classification using synsets in the WordNet lexical database. In 2022 First International Conference on Electrical, Electronics, Information and Communication Technologies (ICEEICT) (pp. 01–07). https://doi.org/10.1109/ICEEICT53079.2022.9768641

Li, Z., Zhu, H., Lu, Z., & Yin, M. (2023). Synthetic data generation with large language models for text classification: Potential and limitations. arXiv preprint arXiv:2310.07849. Retrieved from https://arxiv.org/abs/2310.07849

Liu, B. (2012). Sentiment analysis and opinion mining (1st ed.). Springer. https://doi.org/10.1007/978-3-031-02145-9

McHugh, M. L. (2012). Interrater reliability: The kappa statistic. Biochemia Medica (Zagreb), 22(3), 276-282. PMID: 23092060; PMCID: PMC3900052.

Monitsky, V. (2023, October 16). Text classification with Hugging Face Trainer and PyTorch. Medium. Retrieved June 25, 2024, from https://medium.com/@vmn11/text-classification-with-hugging-face-trainer-and-pytorch-8c0b07e67b4a

Nagaraj, Y. (2023, October 20). Stratified K-Fold Cross-Validation: An in-depth look . LinkedIn. Retrieved June 18, 2024, from https://www.linkedin.com/pulse/stratified-k-fold-cross-validation-in-depth-look-yeshwanth-n/.

Nidhika. (2023, February 23). Sentiment analysis with Python: SentiWordNet. AI & Tech. Medium. Retrieved June 23, 2024, https://medium.com/@nidhikayadav/sentiment-analysis-with-python-sentiwordnet-fd07ffc557.

Nigam, K., Lafferty, J. D., & McCallum, A. (1999). Using Maximum Entropy for Text Classification. Retrieved June 18, 2024, from https://api.semanticscholar.org/CorpusID:574041

Patil, S. [@patil-suraj]. (n.d.). Suraj Patil. GitHub. Retrieved June 18, 2024, from https://github.com/patil-suraj

Polat, E. I. (2023, May 2). Plot and customize multiple confusion matrices with matplotlib. Retrieved June 25, 2024, Medium. https://medium.com/@eceisikpolat/plot-and-customize-multiple-confusion-matrices-with-matplotlib-a19ed00ca16c

rakshita_iyer. (2023, February 17). Python – Sentiment analysis using Affin. GeeksForGeeks. Retrieved June 22, 2024, Medium. https://www.geeksforgeeks.org/python-sentiment-analysis-using-affin/

Ray Team. (n.d.). BayesOptSearch. Ray Tune Documentation. Retrieved June 25, 2024, from https://docs.ray.io/en/latest/tune/api/doc/ray.tune.search.bayesopt.BayesOptSearch.html

Ray Team. (n.d.). BayesOpt example. Retrieved June 25, 2024, from https://docs.ray.io/en/latest/tune/examples/includes/bayesopt_example.html

Ray Team. (n.d.). Tune search algorithms (tune.search). Ray Documentation. Retrieved June 29, 2024, from https://docs.ray.io/en/latest/tune/api/suggestion.html

Sanh, V., Debut, L., Chaumond, J., & Wolf, T. (2019). DistilBERT, a distilled version of BERT: Smaller, faster, cheaper and lighter. ArXiv. https://doi.org/10.48550/arXiv.1910.01108

Saumyab271. (2024, May 27). Stemming vs Lemmatization in NLP: Must-Know Differences. Analytics Vidhya. Retrieved June 29, 2024, from https://www.analyticsvidhya.com/blog/2022/06/stemming-vs-lemmatization-in-nlp-must-know-differences/

Shalevska, E., & Ma, Y. (2024). The digital laureate: Examining AI-generated poetry. RATE Issues, 31. https://doi.org/10.69475/RATEI.2024.1.1

Sheng, E., & Uthus, D. (2020). Investigating societal biases in a poetry composition system. arXiv. Retrieved June 15, 2024, from https://arxiv.org/abs/2011.02686

Sheng, E., & Uthus, D. (2020). Poem sentiment [Data set]. Hugging Face. Retrieved June 10, 2024, from https://huggingface.co/datasets/google-research-datasets/poem_sentiment

Sharma, A., & Dey, S. (2012). A comparative study of feature selection and machine learning techniques for sentiment analysis. In Proceedings of the 2012 ACM Research in Applied Computation Symposium (RACS '12), 1–7. Association for Computing Machinery. https://doi.org/10.1145/2401603.2401605

Sharma, S. (2021, June 30). Sentiment Analysis Using the SentiWordNet Lexicon. Medium. Retrieved June 25, 2024, from https://srish6.medium.com/sentiment-analysis-using-the-sentiwordnet-lexicon-1a3d8d856a10

Sunnyville. (2020, September 26). Fine-tuning DistilBERT for multi-class text classification using transformers and TensorFlow. Retrieved June 25, 2024, from https://www.sunnyville.ai/fine-tuning-distilbert-multi-class-text-classification-using-transformers-and-tensorflow/

Tan, C.-M., Wang, Y.-F., & Lee, C.-D. (2002). The use of bigrams to enhance text categorization. Information Processing & Management, 38(4), 529-546. https://doi.org/10.1016/S0306-4573(01)00045-0orrectly.

Tunstall, L., von Werra, L., & Wolf, T. (2022). Natural language processing with transformers (Revised color ed.). O'Reilly Media.

VanderPlas, J. (2016). Python data science handbook: Essential tools for working with data. O'Reilly Media, Inc.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2023). Attention Is All You Need. arXiv. https://arxiv.org/abs/1706.03762

Vryniotis, V. (2013, November 20). Machine Learning Tutorial: The Max Entropy Text Classifier. Datumbox Machine Learning Blog. Retrieved June 19, 2024, from https://blog.datumbox.com/machine-learning-tutorial-the-max-entropy-text-classifier/.

Wang, J., Yang, Y., & Xia, B. (2019). A simplified Cohen's Kappa for use in binary classification data annotation tasks. IEEE Access, 7, 164386-164397. https://doi.org/10.1109/ACCESS.2019.2953104

Wang, S., & Manning, C. (2012). Baselines and bigrams: Simple, good sentiment and topic classification. In H. Li, C.-Y. Lin, M. Osborne, G. G. Lee, & J. C. Park (Eds.), Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers) (pp. 90-94). Association for Computational Linguistics. https://aclanthology.org/P12-2018

Widodo, S., Brawijaya, H., & Samudi, S. (2022). Stratified K-fold cross validation optimization on machine learning for prediction. Sinkron: Jurnal Dan Penelitian Teknik Informatika, 6(4), 2407-2414. https://doi.org/10.33395/sinkron.v7i4.11792

Yusof, N. N., Mohamed, A., & Abdul-Rahman, S. (2015). Reviewing classification approaches in sentiment analysis. In M. W. Berry, A. Mohamed, & B. W. Yap (Eds.), Soft Computing in Data Science (pp. 43-53). Springer Singapore. https://doi.org/10.1007/978-981-287-936-3_5}