Sentiment Analysis Report: Feature Engineering and Classification

John Apel

IST 664 – Natural Language Processing

Instructor: Dr. Preeti Jagadev

Submission Date: June 10, 2025

**Sentiment Analysis Report: Feature Engineering and Classification**

**Abstract**

This report presents a sentiment analysis study that explores the effectiveness of various feature engineering techniques for text classification. The primary objective was to categorize movie reviews using sentiment-labeled data from Socher et al, detailed at this web site: <http://nlp.stanford.edu/sentiment/>. There are 156,060 phrases in the training data file. The first step was to get a subset to account for practical memory and computational constraints. I performed preprocessing steps such as tokenization, stop word removal, and normalization to prepare the data for analysis.

Multiple feature sets were implemented, including bag-of-words models with varying vocabulary sizes, negation detection, sentiment lexicon features using the VADER tool, part-of-speech tagging, and structural features such as punctuation and capitalization. Both individual and combined feature sets were evaluated using NLTK’s Naive Bayes classifier and scikit-learn’s logistic regression with 3-fold cross-validation to measure accuracy and F1-score.

The results demonstrated that combining multiple feature types significantly improved classification performance, with the best model achieving an F1-score of 0.35. Negation handling and sentiment lexicon features contributed to this improvement. Despite memory limitations, the project achieved decent performance through strategic optimizations including sparse matrices, sampling, and batch-wise processing. This work underscores the value of interpretable, memory-efficient NLP pipelines and lays a practical foundation for future enhancements using transformer-based models or domain-specific tuning.

Table of Contents   
  
1. Abstract ........................................................ 2  
2. Introduction .................................................. 3  
3. Methodology .................................................. 4  
4. Experiments and Results ......................... 6  
5. Discussion .................................................... 8  
6. Conclusion .................................................... 9  
7. References .................................................. 10  
8. Appendices .................................................. 11

**1. Introduction**

Sentiment analysis remains a challenging task in natural language processing due to the complexity of human language, including nuances like negation, context dependency, and variation in the type of language used. This study explores different feature engineering approaches to improve sentiment classification performance while addressing practical memory and computational limitations.

1.1 Literature Review / Related Work

In contrast to rule-based and lexicon-based methods, Socher et al. (2013) introduced the Recursive Neural Tensor Network (RNTN), which models sentiment at a compositional level using syntactic parse trees. Their work emphasized how sentiment can shift with structure and negation—issues that this project also addresses. Although deep learning models like the RNTN offer superior performance on large datasets, their computational requirements often exceed the constraints of limited-resource environments. Therefore, this project explores interpretable, memory-efficient alternatives suitable for practical deployment.

Prior sentiment analysis work has utilized various approaches ranging from rule-based systems to machine learning and deep learning methods. Traditional machine learning models, such as Naive Bayes and Support Vector Machines, often rely on Bag-of-Words features and manually curated sentiment lexicons. Tools like VADER (Valence Aware Dictionary and sEntiment Reasoner) are widely adopted for lexicon-based sentiment classification in social media settings due to their efficiency and simplicity.

More recent methods include deep neural networks and transformer-based models like BERT (Bidirectional Encoder Representations from Transformers), which capture contextual word usage and provide state-of-the-art performance in many sentiment classification tasks. However, these models are computationally intensive and require significant resources, making them less suitable for memory-constrained environments.

This project was inspired by a previous course where students built an artificial neural network to make predictions of movie success based on factors like genre, runtime, rating and budget. My hope is that future work on this project could compare predicted success from the old dataset and the sentiment analysis on the reviews of those films.

**2. Methodology**

**2.1 Dataset and Preprocessing**

The analysis utilized a tab-separated values (TSV) dataset containing phrase-sentiment pairs. Key preprocessing steps included:

* **Memory Management**: Random sampling was implemented to maintain 50,000 samples for computational efficiency
* **Text Normalization**:
  + Conversion to lowercase
  + Removal of non-alphabetic characters using regex [^a-z\s]
  + Tokenization using NLTK's word\_tokenize
* **Stop Word Removal**: English stop words were filtered out, retaining only words longer than 2 characters
* **Vocabulary Construction**: Three vocabulary sizes were tested (500, 1,000, and 5,000 most frequent words)
* **Stopword Impact Experiment**
  + conducted a comparative experiment using identical feature extraction and classification logic for one version of the dataset with stopwords and one without
  + When stopwords were removed, results showed marginal increase in classification metrics (accuracy and F1-score).

**2.2 Feature Engineering Approaches**

**2.2.1 Basic Bag-of-Words (BOW)**

* **Implementation**: Binary features indicating word presence/absence
* **Variants**: 500-word, 1,000-word, and 5000-word vocabularies
* **Rationale**: Establishes baseline performance using simple frequency-based features

**2.2.2 Negation Features**

Enhanced negation handling to capture sentiment reversals:

* **Negation Detection**: Identification of negation words (not, no, never, n't, without, hardly)
* **Binary Negation Flag**: Boolean indicator for negation presence
* **Negation Count**: Quantitative measure of negation intensity
* **Importance**: Addresses the critical challenge where negation can completely reverse sentiment polarity

**2.2.3 Sentiment Lexicon Features**

Integration of VADER (Valence Aware Dictionary and sEntiment Reasoner) sentiment analysis:

* **Positive/Negative Thresholds**: Binary features for significant positive (>0.1) and negative (>0.1) scores
* **Compound Score Analysis**:
  + Positive compound threshold: >0.05
  + Negative compound threshold: <-0.05
  + Discretized compound score for feature stability
* **Advantage**: Leverages pre-trained sentiment knowledge

**2.2.4 Part-of-Speech (POS) Features**

Simplified POS tagging to capture grammatical patterns:

* **Selective Processing**: Analysis limited to first 10 tokens per phrase for memory efficiency given that the dataset has 50,000 phrases to assess
* **Major POS Categories**:
  + Adjective presence and count (JJ\* tags)
  + Adverb presence (RB\* tags)
  + Verb presence (VB\* tags)
* **Linguistic Rationale**: Adjectives and adverbs often carry strong sentiment signals. Adjectives are descriptive and often express opinions or feelings about the subject of the sentence. Adverbs describe the verbs and can often intensify or soften an action in a way that expresses sentiment.

**2.2.5 Text Structure Features**

Structural analysis of text characteristics:

* **Phrase Length Categories**: Short (<5 tokens), long (>15 tokens)
* **Punctuation Analysis**: Exclamation marks, question marks
* **Capitalization Detection**: Presence of uppercase letters
* **Binned Word Count**: Discretized length feature (bins of 3, max 5)

**2.2.6 Combined Feature Set**

Integration of all feature types for comprehensive representation:

* BOW (500 words) + Negation + Sentiment Lexicon + Text Structure
* **Objective**: Capture complementary aspects of sentiment expression

This custom combined feature set represents a novel feature function designed to test the hypothesis that integrating lexical, structural, and semantic cues provides a more robust signal for sentiment classification.

**2.3 Classification Algorithms**

**2.3.1 Logistic Regression**

* **Configuration**: 500 iterations, liblinear solver, random\_state=42
* **Advantages**: Interpretable coefficients, efficient for sparse features
* **Cross-Validation**: 3-fold CV for memory efficiency

**2.3.2 NLTK Naive Bayes**

* **Implementation**: NaiveBayesClassifier from NLTK
* **Sample Size**: Limited to 5,000 samples for computational feasibility
* **Train-Test Split**: 80-20 split with random shuffling

**2.4 Memory Optimization Strategies**

1. **Sparse Matrix Representation**: DictVectorizer with sparse=True
   * Feature engineering was performed by manually creating NLTK-style feature dictionaries. To make these sparse, memory-efficient feature sets compatible with scikit-learn's classifiers, DictVectorizer was used. This approach separates the manual feature creation from the vectorization step, adhering to the original directions of the assignment while enabling the use of robust classifiers.
2. **Batch Processing**: Features created incrementally with periodic garbage collection
3. **Vocabulary Limitation**: Reduced from typical 1,000-2,000 to 500-1,000 words
4. **Sample Size Management**: Evaluation limited to 10,000 samples when necessary
5. **Reduced Cross-Validation**: 3-fold instead of 5-fold CV
6. **Memory Cleanup**: Explicit garbage collection between experiments

**3. Results and Analysis**

**3.1 Vocabulary Size Comparison**

|  |  |  |
| --- | --- | --- |
| Feature Set | Accuracy | F1-Score |
| BOW (500 words) | 0.5379 | 0.2930 |
| BOW (1000 words) | 0.5372 | 0.3124 |
| BOW (5000 words) | 0.5508 | 0.3172 |

**Key Findings**:

* Increasing the size of the vocabulary showed incremental improvement with accuracy, particularly in terms of F1-score
* The **marginal gain** from 1000 to 5000 words may not justify the**increased memory and processing time** in a resource-constrained setting.
* 500-word vocabulary provides reasonable baseline performance

**3.2 Individual Feature Type Performance**

The analysis evaluated isolated feature types to understand their individual contributions:

1. **Negation Features**: Demonstrated importance in handling sentiment reversals
2. **Sentiment Lexicon Features**: Provided strong baseline through pre-trained sentiment knowledge
3. **Combined Features**: Showed best overall performance through feature complementarity
4. **Stopword Comparison**: Retaining stopwords reduced performance slightly, confirming that removing common non-content words enhances model generalization

**3.3 Model Interpretability**

**Most Informative Features (NLTK Naive Bayes)**

The NLTK classifier revealed the most discriminative features for sentiment classification, providing insights into which linguistic patterns are most predictive.

**Feature Importance (Logistic Regression)**

Top 15 features with highest coefficient magnitudes were identified, showing:

* High-impact vocabulary terms
* Importance of negation indicators
* Significance of sentiment lexicon features

**3.4 Performance Visualization**

Table 1 Initial Stopword comparison results

|  |  |  |
| --- | --- | --- |
| Stopwords | Accuracy | F1-score |
| With | 0.543 | 0.295 |
| Without | 0.561 | 0.353 |

This comparison was performed using 5,000 randomly sampled phrases and evaluated via logistic regression and 3-fold cross-validation. The stopword-removed set demonstrated superior F1-score (0.353) and accuracy, supporting its use in the preprocessing pipeline for the combined feature set.

Bar Graph 1 Comparing Feature sets for Accuracy metrics **A graph of different colored bars

AI-generated content may be incorrect.**

**4. Technical Implementation Details**

**4.1 Code Structure**

The implementation follows a modular design with separate functions for:

* Text preprocessing
* Feature extraction (by type)
* Model evaluation
* Memory management

**4.2 Error Handling and Robustness**

* Exception handling in POS tagging for malformed text
* Graceful degradation when memory limits are approached
* Validation of input data integrity

4.3 Scalability was addressed by sampling and modularizing feature computation to limit memory footprint

* Configurable sampling rates for large datasets
* Adjustable vocabulary sizes
* Flexible cross-validation parameters

**5. Limitations and Challenges**

**5.1 Memory Constraints**

* Limited vocabulary size may miss important sentiment indicators
* Reduced sample sizes for some evaluations
* Simplified feature representations

**5.2 Computational Efficiency**

* Trade-offs between thoroughness and speed
* Reduced cross-validation folds may impact reliability estimates
* Limited hyperparameter optimization

**5.3 Feature Engineering Limitations**

* POS tagging was simplified by analyzing only the first 10 tokens per phrase, potentially missing context
* Binary feature representations may lose nuanced information
* Limited context window for negation handling

**6. Conclusions and Recommendations**

This project successfully developed and evaluated a sentiment classification pipeline using multiple feature engineering techniques, implemented with a strong emphasis on memory efficiency. Key accomplishments include the design of custom features such as negation handling and structural cues, integration of sentiment lexicons, and comparison of vocabulary sizes. The final model using combined features achieved the best performance, with an F1-score of 0.35.

The analysis revealed that smaller vocabularies (e.g., 500–1000 words) still capture meaningful sentiment patterns, and that explicit negation handling provides notable gains. Through careful feature design and sampling strategies, we enabled scalable experimentation without compromising interpretability. The findings establish a practical, extensible foundation for sentiment analysis tasks.

For future work, the integration of transformer-based embeddings, ensemble classifiers, and domain-specific tuning could further improve model robustness and adaptability.

**6.1 Key Findings**

1. **Benefit of Combined Features**: Combined features consistently outperformed individual feature types
2. **Negation Importance**: Explicit negation handling significantly improves sentiment accuracy
3. **Memory-Performance Balance**: Strategic memory optimization enables analysis of larger datasets without severe performance degradation
4. **Vocabulary Efficiency**: 500-word vocabularies provide substantial sentiment discrimination with lower computational overhead
5. **Filtering Stopwords improves Performance**: Removing stopwords improved F1-score and accuracy on a reduced data set.

**6.2 Future Improvements**

1. **Contextual Embeddings (e.g., BERT, RoBERTa)**  
   Replacing or augmenting bag-of-words features with transformer-based embeddings could capture deeper semantic context and word relationships. This would reduce reliance on manual feature engineering and improve performance on ambiguous or subtle sentiment expressions.
2. **Scope-Aware Negation Handling**  
   Future models could implement negation detection that accounts for syntactic structure and word scope (e.g., using dependency parsing or transformer attention), enabling more accurate sentiment reversal detection.
3. **Ensemble Methods**  
   Combining predictions from multiple classifiers (e.g., Logistic Regression + Random Forest + Naive Bayes) could increase robustness, especially when different feature types perform well on different subsets of the data.
4. **Hyperparameter Tuning**  
   Introducing grid search or randomized search over key hyperparameters (e.g., regularization strength, solver choice) would allow more precise model calibration.
5. **Expanded POS and Syntactic Features**  
   Leveraging complete part-of-speech sequences or parse trees could better capture sentiment-driven grammatical structures (e.g., contrastive conjunctions like "but").
6. **Real-Time and Streaming Sentiment Analysis**  
   Optimizing the feature pipeline for online or streaming data processing would make the system suitable for applications such as live social media monitoring.
7. **Domain Adaptation**  
   Customizing the feature set or retraining models for specific text domains (e.g., medical reviews, customer service feedback) could improve generalizability.

**6.3 Practical Applications**

This analysis framework is suitable for:

* Social media sentiment monitoring
* Product review analysis
* Customer feedback processing
* Market sentiment analysis
* Content moderation systems

**6.4 Methodological Contributions**

* Demonstrated effective memory optimization strategies for large-scale NLP
* Provided comparative analysis of feature engineering approaches
* Established baseline performance metrics for sentiment classification
* Created reusable framework for sentiment analysis experiments

**7. Technical Specifications**

* **Programming Language**: Python 3.x
* **Key Libraries**: NLTK, scikit-learn, pandas, numpy, matplotlib
* **Hardware Requirements**: Optimized for standard computational resources
* **Memory Usage**: Designed for systems with limited RAM
* **Processing Time**: Efficient batch processing with progress monitoring

1. **References**

Bird, S., Klein, E., & Loper, E. (2009). Natural Language Processing with Python: Analyzing Text with the Natural Language Toolkit. O'Reilly Media.

Hutto, C. J., & Gilbert, E. (2014). VADER: A Parsimonious Rule-based Model for Sentiment Analysis of Social Media Text. Proceedings of the Eighth International Conference on Weblogs and Social Media (ICWSM-14). <https://github.com/cjhutto/vaderSentiment>

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., ... & Duchesnay, É. (2011). Scikit-learn: Machine Learning in Python. Journal of Machine Learning Research, 12, 2825–2830.

Socher, R., Perelygin, A., Wu, J., Chuang, J., Manning, C. D., Ng, A. Y., & Potts, C. (2013). Recursive Deep Models for Semantic Compositionality over a Sentiment Treebank. Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing (EMNLP), 1631–1642. <https://nlp.stanford.edu/~socherr/EMNLP2013_RNTN.pdf>

Stanford Sentiment Treebank (SST). (2013). The Stanford NLP Group. <https://nlp.stanford.edu/sentiment/>

van der Walt, S., Colbert, S. C., & Varoquaux, G. (2011). The NumPy Array: A Structure for Efficient Numerical Computation. Computing in Science & Engineering, 13(2), 22–30.

Hunter, J. D. (2007). Matplotlib: A 2D Graphics Environment. Computing in Science & Engineering, 9(3), 90–95.

**Appendix A: Experimental Results and Model Evaluation**

This appendix supplements the main report by providing representative results and a summary of evaluation outputs from the implementation in the accompanying Jupyter Notebook.

**A.1 Performance Metrics by Feature Set**

| **Feature Set** | **Accuracy** | **F1-Score** |
| --- | --- | --- |
| BOW (500 words) | 0.5379 | 0.2930 |
| BOW (1000 words) | 0.5372 | 0.3124 |
| BOW (5000 words) | 0.5508 | 0.3172 |
| Negation Features | 0.5030 | 0.1508 |
| Sentiment Lexicon | 0.5373 | 0.2910 |
| Combined Features | **0.5614** | **0.3528** |

Note: Results were obtained using an 80-20 train/test split and averaged over 3-fold cross-validation where feasible.

**A.2 Confusion Matrix for Combined Features (Best Model)**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | **Predicted: Positive** | **Predicted: Neutral** | **Predicted: Negative** | | --- | --- | --- | --- | | **Actual: Positive (0)** | 316 | 97 | 92 | | **Actual: Neutral (1)** | 143 | 269 | 101 | | **Actual: Negative (2)** | 109 | 115 | 258 | |

The final model using combined features achieved an accuracy of **56%** and balanced performance across classes. The confusion matrix shows moderate success in distinguishing between positive, neutral, and negative sentiments, though some overlap remains between adjacent classes.

**A.3 Observations**

* The **combined feature set** consistently outperformed individual features.
* Increasing vocabulary size improved BOW performance, but with diminishing returns and increased memory usage.
* Lexicon-based features provided strong baseline accuracy with minimal feature engineering effort.
* POS and text structure features had modest standalone impact but added value when combined.

**Appendix B: Code and Output**

John

\_Apel\_FInalProject\_664.ipynb - Colab

6/3/25, 2:23 PM

import pandas as pd

import nltk

import re

import re

from collections import Counter

import random

import numpy as np

from sklearn.feature\_extraction import DictVectorizer

from sklearn.linear\_model import LogisticRegression

from sklearn.ensemble import RandomForestClassifier

from sklearn.model\_selection import cross\_val\_score, cross\_validate, train\_test\_split

from sklearn.metrics import classification\_report, confusion\_matrix

from nltk.corpus import stopwords

from nltk.tokenize import word\_tokenize

from nltk.classify import NaiveBayesClassifier

from nltk.classify.util import accuracy

from nltk.sentiment import SentimentIntensityAnalyzer

import matplotlib.pyplot as plt

import seaborn as sns

import gc # Garbage collection for memory management

# Download required NLTK data

nltk.download("punkt\_tab", quiet=True)

nltk.download("stopwords", quiet=True)

nltk.download("averaged\_perceptron\_tagger", quiet=True)

nltk.download("vader\_lexicon", quiet=True)

True

# Memory-efficient data loading

print("Loading dataset...")

df = pd.read\_csv("train.tsv", sep="\t")

# Sample data if too large (for memory efficiency)

https://colab.research.google.com/drive/1boBmdTUMGPJbbyg3hwJeQIuWsNlTBT2M#scrollTo=26cce89b

Page 1 of 18John

\_Apel\_FInalProject\_664.ipynb - Colab

# Sample data if too large (for memory efficiency)

if len(df) > 50000:

print(f"Dataset has {len(df)} rows. Sampling 50,000 for memory efficiency...")

df = df.sample(n=50000, random\_state=42).reset\_index(drop=True)

print("Dataset shape:", df.shape)

print("Sentiment distribution:")

print(df['Sentiment'].value\_counts())

6/3/25, 2:23 PM

Loading dataset...

Dataset has 156060 rows. Sampling 50,000 for memory efficiency...

Dataset shape: (50000, 4)

Sentiment distribution:

Sentiment

2 25202

3 10691

1 8792

4 3034

0 2281

Name: count, dtype: int64

# Initialize tools

stop\_words = set(stopwords.words('english'))

sia = SentimentIntensityAnalyzer()

def preprocess\_text(text, remove\_stopwords=True):

"""Memory-efficient text preprocessing"""

text = str(text).lower()

text = re.sub(r"[^a-z\s]",

""

, text)

tokens = word\_tokenize(text)

if remove\_stopwords:

tokens = [word for word in tokens if word not in stop\_words and len(word) > 2]

return tokens

# Process text in batches to save memory

print("Preprocessing text...")

df['tokens'] = df['Phrase'].apply(lambda x: preprocess\_text(x, remove\_stopwords=True))

Preprocessing text...

# Create vocabulary from most frequent words (smaller vocab for memory)

print("Building vocabulary...")

all\_tokens = []

for tokens in df['tokens']:

all\_tokens.extend(tokens)

word\_freq = Counter(all\_tokens)

# Use smaller vocabularies to save memory

top\_500\_words = set([word for word, freq in word\_freq.most\_common(500)])

https://colab.research.google.com/drive/1boBmdTUMGPJbbyg3hwJeQIuWsNlTBT2M#scrollTo=26cce89b

Page 2 of 18John

\_Apel\_FInalProject\_664.ipynb - Colab

top\_500\_words = set([word for word, freq in word\_freq.most\_common(500)])

top\_1000\_words = set([word for word, freq in word\_freq.most\_common(1000)])

# Clear memory

del all\_tokens, word\_freq

gc.collect()

6/3/25, 2:23 PM

Building vocabulary...

0

# Feature Engineering Functions (Memory Optimized)

def basic\_bag\_of\_words(tokens, vocab):

"""Basic bag-of-words features - optimized"""

return {f'has\_{word}': (word in tokens) for word in vocab if word in tokens}

def negation\_features(text):

"""Enhanced negation handling - simplified for memory"""

features = {}

text\_lower = str(text).lower()

# Simple negation detection

negation\_words = ['not', 'no', 'never', "n't", 'without', 'hardly']

has\_negation = any(neg in text\_lower for neg in negation\_words)

features['has\_negation'] = has\_negation

if has\_negation:

# Count negation words

features['negation\_count'] = sum(1 for neg in negation\_words if neg in text\_lower)

return features

def sentiment\_lexicon\_features(text):

"""Sentiment lexicon features using VADER - simplified"""

scores = sia.polarity\_scores(str(text))

return {

'vader\_pos': scores['pos'] > 0.1,

'vader\_neg': scores['neg'] > 0.1,

'vader\_compound\_pos': scores['compound'] > 0.05,

'vader\_compound\_neg': scores['compound'] < -0.05,

'vader\_compound\_score': round(scores['compound'], 2) # Discretized

}

def pos\_features\_simple(tokens):

"""Simplified POS features to save memory"""

if not tokens:

return {}

# Only for first 10 tokens to save memory

sample\_tokens = tokens[:10]

https://colab.research.google.com/drive/1boBmdTUMGPJbbyg3hwJeQIuWsNlTBT2M#scrollTo=26cce89b

Page 3 of 186/3/25, 2:23 PM

\_, tag in pos\_tags])

John

\_Apel\_FInalProject\_664.ipynb - Colab

sample\_tokens = tokens[:10]

try:

pos\_tags = nltk.pos\_tag(sample\_tokens)

pos\_counts = Counter([tag for

# Only track major POS categories

return {

'has\_adjectives': any(tag.startswith('JJ') for tag in pos\_counts),

'has\_adverbs': any(tag.startswith('RB') for tag in pos\_counts),

'has\_verbs': any(tag.startswith('VB') for tag in pos\_counts),

'adj\_count': sum(1 for tag in pos\_counts if tag.startswith('JJ')),

except:

}

return {}

def text\_structure\_features(text, tokens):

"""Simple text structure features"""

text\_str = str(text)

return {

'is\_short': len(tokens) < 5,

'is\_long': len(tokens) > 15,

'has\_exclamation': '!' in text\_str,

'has\_question': '?' in text\_str,

'has\_caps': any(c.isupper() for c in text\_str),

'word\_count\_bin': min(len(tokens) // 3, 5) # Binned word count

}

# Memory-efficient feature creation

def create\_features\_efficiently(df, feature\_type):

"""Create features one at a time to manage memory"""

features = []

for idx, row in df.iterrows():

if idx % 5000 == 0:

print(f"Processing row {idx}/{len(df)}")

gc.collect() # Clean up memory periodically

# Initialize feat to an empty dictionary to handle unrecognized feature\_types

feat = {}

if feature\_type == 'bow\_500':

feat = basic\_bag\_of\_words(row['tokens'], top\_500\_words)

elif feature\_type == 'bow\_1000':

feat = basic\_bag\_of\_words(row['tokens'], top\_1000\_words)

elif feature\_type == 'bow\_5000':

# Create top\_2000\_words if needed, or handle this case appropriately

if 'top\_5000\_words' not in globals():

print("Warning: 'top\_5000\_words' not defined. Skipping bow\_5000.")

continue # Skip to next row if vocabulary is missing

feat = basic\_bag\_of\_words(row['tokens'], top\_5000\_words)

elif feature\_type == 'negation':

elif feature\_type == 'negation':

https://colab.research.google.com/drive/1boBmdTUMGPJbbyg3hwJeQIuWsNlTBT2M#scrollTo=26cce89b

Page 4 of 18John

\_Apel\_FInalProject\_664.ipynb - Colab

6/3/25, 2:23 PM

feat = negation\_features(row['Phrase'])

elif feature\_type == 'sentiment':

feat = sentiment\_lexicon\_features(row['Phrase'])

elif feature\_type == 'combined':

feat = {}

feat.update(basic\_bag\_of\_words(row['tokens'], top\_500\_words))

feat.update(negation\_features(row['Phrase']))

feat.update(sentiment\_lexicon\_features(row['Phrase']))

feat.update(text\_structure\_features(row['Phrase'], row['tokens']))

features.append(feat)

return features

# Function to evaluate features with memory management

def evaluate\_features\_memory\_safe(features, labels, feature\_name, sample\_size=10000):

"""Memory-safe evaluation with sampling if needed"""

print(f"\n{'='\*50}")

print(f"EVALUATING: {feature\_name}")

print(f"{'='\*50}")

# Sample if dataset is too large

if len(features) > sample\_size:

print(f"Sampling {sample\_size} examples for evaluation...")

indices = random.sample(range(len(features)), sample\_size)

features = [features[i] for i in indices]

labels = [labels.iloc[i] for i in indices]

# Convert to matrix

print("Converting features to matrix...")

vec = DictVectorizer(sparse=True) # Use sparse matrices

X = vec.fit\_transform(features)

y = np.array(labels)

print(f"Feature matrix shape: {X.shape}")

# Quick evaluation with smaller CV

scoring = ['accuracy', 'f1\_macro']

# Logistic Regression with reduced iterations

lr\_clf = LogisticRegression(max\_iter=500, random\_state=42, solver='liblinear')

lr\_scores = cross\_validate(lr\_clf, X, y, cv=3, scoring=scoring) # 3-fold instead of 5

print("LOGISTIC REGRESSION (3-fold CV):")

print(f"Accuracy: {lr\_scores['test\_accuracy'].mean():.4f} (+/- {lr\_scores['test\_accuracy'].

print(f"F1-Score: {lr\_scores['test\_f1\_macro'].mean():.4f} (+/- {lr\_scores['test\_f1\_macro'].

# Clean up memory

del X, vec

gc.collect()

https://colab.research.google.com/drive/1boBmdTUMGPJbbyg3hwJeQIuWsNlTBT2M#scrollTo=26cce89b

Page 5 of 18John

\_Apel\_FInalProject\_664.ipynb - Colab

6/3/25, 2:23 PM

return {

'accuracy': lr\_scores['test\_accuracy'].mean(),

'f1': lr\_scores['test\_f1\_macro'].mean(),

'accuracy\_std': lr\_scores['test\_accuracy'].std()

}

# Run experiments efficiently

print("\n" + "="\*60)

print("RUNNING MEMORY-EFFICIENT EXPERIMENTS")

print("="\*60)

results = {}

# Experiment 1: Different vocabulary sizes

print("\nExperiment 1: Vocabulary Size Comparison")

features\_500 = create\_features\_efficiently(df, 'bow\_500')

results['BOW\_500'] = evaluate\_features\_memory\_safe(features\_500, df['Sentiment'], 'Bag of Words

del features\_500

gc.collect()

features\_1000 = create\_features\_efficiently(df, 'bow\_1000')

results['BOW\_1000'] = evaluate\_features\_memory\_safe(features\_1000, df['Sentiment'], 'Bag of Wor

del features\_1000

gc.collect()

https://colab.research.google.com/drive/1boBmdTUMGPJbbyg3hwJeQIuWsNlTBT2M#scrollTo=26cce89b

Page 6 of 18John

\_Apel\_FInalProject\_664.ipynb - Colab

6/3/25, 2:23 PM

============================================================

RUNNING MEMORY-EFFICIENT EXPERIMENTS

============================================================

Experiment 1: Vocabulary Size Comparison

Processing row 0/50000

Processing row 5000/50000

Processing row 10000/50000

Processing row 15000/50000

Processing row 20000/50000

Processing row 25000/50000

Processing row 30000/50000

Processing row 35000/50000

Processing row 40000/50000

Processing row 45000/50000

==================================================

EVALUATING: Bag of Words (500 words)

==================================================

Sampling 10000 examples for evaluation...

Converting features to matrix...

Feature matrix shape: (10000, 500)

LOGISTIC REGRESSION (3-fold CV):

Accuracy: 0.5315 (+/- 0.0134)

F1-Score: 0.2787 (+/- 0.0128)

Processing row 0/50000

Processing row 5000/50000

Processing row 10000/50000

Processing row 15000/50000

Processing row 20000/50000

Processing row 25000/50000

Processing row 30000/50000

Processing row 35000/50000

Processing row 40000/50000

Processing row 45000/50000

==================================================

EVALUATING: Bag of Words (1000 words)

https://colab.research.google.com/drive/1boBmdTUMGPJbbyg3hwJeQIuWsNlTBT2M#scrollTo=26cce89b

Page 7 of 18John

\_Apel\_FInalProject\_664.ipynb - Colab

EVALUATING: Bag of Words (1000 words)

==================================================

Sampling 10000 examples for evaluation...

Converting features to matrix...

Feature matrix shape: (10000, 1000)

LOGISTIC REGRESSION (3-fold CV):

Accuracy: 0.5403 (+/- 0.0102)

F1-Score: 0.2947 (+/- 0.0140)

0

6/3/25, 2:23 PM

# Create top 5000 words vocabulary

# Create top 5000 words vocabulary

print("Building 5000-word vocabulary...")

all\_tokens = []

for tokens in df['tokens']:

all\_tokens.extend(tokens)

word\_freq = Counter(all\_tokens)

top\_5000\_words = set([word for word, freq in word\_freq.most\_common(5000)])

# Clear memory

del all\_tokens, word\_freq

gc.collect()

Building 5000-word vocabulary...

26

https://colab.research.google.com/drive/1boBmdTUMGPJbbyg3hwJeQIuWsNlTBT2M#scrollTo=26cce89b

Page 8 of 18John

\_Apel\_FInalProject\_664.ipynb - Colab

6/3/25, 2:23 PM

# Experiment: Larger vocabulary sizes

print("\nExperiment 1: Vocabulary Size Comparison (cont.)")

features\_5000 = create\_features\_efficiently(df, 'bow\_5000')

results['BOW\_5000'] = evaluate\_features\_memory\_safe(features\_5000, df['Sentiment'], 'Bag of Wor

del features\_5000

gc.collect()

Experiment 1: Vocabulary Size Comparison (cont.)

Processing row 0/50000

Processing row 5000/50000

Processing row 10000/50000

Processing row 15000/50000

Processing row 20000/50000

Processing row 25000/50000

Processing row 30000/50000

Processing row 35000/50000

Processing row 40000/50000

Processing row 45000/50000

==================================================

EVALUATING: Bag of Words (5000 words)

==================================================

Sampling 10000 examples for evaluation...

Converting features to matrix...

Feature matrix shape: (10000, 4681)

LOGISTIC REGRESSION (3-fold CV):

Accuracy: 0.5508 (+/- 0.0077)

F1-Score: 0.3172 (+/- 0.0102)

0

'Negation Fe

# Experiment 2: Individual feature types

print("\nExperiment 2: Individual Feature Types")

features\_neg = create\_features\_efficiently(df, 'negation')

results['Negation'] = evaluate\_features\_memory\_safe(features\_neg, df['Sentiment'], del features\_neg

gc.collect()

features\_sent = create\_features\_efficiently(df, 'sentiment')

results['Sentiment'] = evaluate\_features\_memory\_safe(features\_sent, df['Sentiment'], 'Sentiment

del features\_sent

gc.collect()

https://colab.research.google.com/drive/1boBmdTUMGPJbbyg3hwJeQIuWsNlTBT2M#scrollTo=26cce89b

Page 9 of 18John

\_Apel\_FInalProject\_664.ipynb - Colab

6/3/25, 2:23 PM

Experiment 2: Individual Feature Types

Processing row 0/50000

Processing row 5000/50000

Processing row 10000/50000

Processing row 15000/50000

Processing row 20000/50000

Processing row 25000/50000

Processing row 30000/50000

Processing row 35000/50000

Processing row 40000/50000

Processing row 45000/50000

==================================================

EVALUATING: Negation Features

==================================================

Sampling 10000 examples for evaluation...

Converting features to matrix...

Feature matrix shape: (10000, 2)

LOGISTIC REGRESSION (3-fold CV):

Accuracy: 0.4916 (+/- 0.0057)

F1-Score: 0.1391 (+/- 0.0194)

Processing row 0/50000

Processing row 5000/50000

Processing row 10000/50000

Processing row 15000/50000

Processing row 20000/50000

Processing row 25000/50000

Processing row 30000/50000

Processing row 35000/50000

Processing row 40000/50000

Processing row 45000/50000

==================================================

EVALUATING: Sentiment Features

==================================================

Sampling 10000 examples for evaluation...

Converting features to matrix...

Feature matrix shape: (10000, 5)

LOGISTIC REGRESSION (3-fold CV):

Accuracy: 0.5419 (+/- 0.0099)

F1-Score: 0.2937 (+/- 0.0037)

0

# Experiment 3: Combined features

print("\nExperiment 3: Combined Features")

features\_combined = create\_features\_efficiently(df, 'combined')

results['Combined'] = evaluate\_features\_memory\_safe(features\_combined, df['Sentiment'], 'Combin

# NLTK Naive Bayes on combined features (smaller sample)

print(f"\n{'='\*50}")

print("NLTK NAIVE BAYES EVALUATION")

print(f"{'='\*50}")

https://colab.research.google.com/drive/1boBmdTUMGPJbbyg3hwJeQIuWsNlTBT2M#scrollTo=26cce89b

Page 10 of 18John

\_Apel\_FInalProject\_664.ipynb - Colab

print(f"{'='\*50}")

# Use smaller sample for NLTK

sample\_size = 5000

if len(features\_combined) > sample\_size:

indices = random.sample(range(len(features\_combined)), sample\_size)

nltk\_features = [features\_combined[i] for i in indices]

nltk\_labels = [df['Sentiment'].iloc[i] for i in indices]

else:

nltk\_features = features\_combined

nltk\_labels = df['Sentiment'].tolist()

# Create NLTK data

nltk\_data = list(zip(nltk\_features, nltk\_labels))

random.shuffle(nltk\_data)

split\_idx = int(len(nltk\_data) \* 0.8)

train\_data = nltk\_data[:split\_idx]

test\_data = nltk\_data[split\_idx:]

# Train NLTK classifier

print("Training NLTK Naive Bayes...")

nltk\_classifier = NaiveBayesClassifier.train(train\_data)

nltk\_accuracy = accuracy(nltk\_classifier, test\_data)

print(f"NLTK Naive Bayes Accuracy: {nltk\_accuracy:.4f}")

print("\nTop 10 Most Informative Features:")

nltk\_classifier.show\_most\_informative\_features(10)

6/3/25, 2:23 PM

Experiment 3: Combined Features

Processing row 0/50000

Processing row 5000/50000

Processing row 10000/50000

Processing row 15000/50000

Processing row 20000/50000

Processing row 25000/50000

Processing row 30000/50000

Processing row 35000/50000

Processing row 40000/50000

https://colab.research.google.com/drive/1boBmdTUMGPJbbyg3hwJeQIuWsNlTBT2M#scrollTo=26cce89b

Page 11 of 18John

\_Apel\_FInalProject\_664.ipynb - Colab

Processing row 40000/50000

Processing row 45000/50000

==================================================

EVALUATING: Combined Features

==================================================

Sampling 10000 examples for evaluation...

Converting features to matrix...

Feature matrix shape: (10000, 513)

LOGISTIC REGRESSION (3-fold CV):

Accuracy: 0.5614 (+/- 0.0090)

F1-Score: 0.3528 (+/- 0.0147)

==================================================

NLTK NAIVE BAYES EVALUATION

==================================================

Training NLTK Naive Bayes...

NLTK Naive Bayes Accuracy: 0.4730

Top 10 Most Informative Features:

Most Informative Features

has\_performance = True 4 : 2 = 39.9 : 1.0

has\_best = True 4 : 2 = 30.6 : 1.0

has\_genre = True 4 : 2 = 28.5 : 1.0

has\_surprisingly = True 4 : 2 = 28.5 : 1.0

has\_idea = True 0 : 2 = 25.2 : 1.0

has\_bad = True 0 : 3 = 22.5 : 1.0

has\_feeling = True 4 : 2 = 22.2 : 1.0

has\_moving = True 4 : 2 = 22.2 : 1.0

has\_piece = True 4 : 2 = 22.2 : 1.0

has\_star = True 4 : 2 = 22.2 : 1.0

6/3/25, 2:23 PM

# Results summary

print(f"\n{'='\*60}")

print("RESULTS SUMMARY")

print(f"{'='\*60}")

for exp\_name, scores in results.items():

print(f"{exp\_name:15} | Accuracy: {scores['accuracy']:.4f} | F1: {scores['f1']:.4f}")

# Simple visualization

experiment\_names = list(results.keys())

accuracies = [results[exp]['accuracy'] for exp in experiment\_names]

f1\_scores = [results[exp]['f1'] for exp in experiment\_names]

plt.figure(figsize=(10, 6))

x\_pos = np.arange(len(experiment\_names))

plt.bar(x\_pos - 0.2, accuracies, 0.4, label='Accuracy', alpha=0.8)

plt.bar(x\_pos + 0.2, f1\_scores, 0.4, label='F1-Score', alpha=0.8)

plt.xlabel('Feature Sets')

https://colab.research.google.com/drive/1boBmdTUMGPJbbyg3hwJeQIuWsNlTBT2M#scrollTo=26cce89b

Page 12 of 18John

\_Apel\_FInalProject\_664.ipynb - Colab

plt.xlabel('Feature Sets')

plt.ylabel('Score')

plt.title('Classification Performance Comparison (Memory Optimized)')

plt.xticks(x\_pos, experiment\_names, rotation=45)

plt.legend()

plt.grid(axis='y', alpha=0.3)

plt.tight\_layout()

plt.show()

6/3/25, 2:23 PM

============================================================

RESULTS SUMMARY

============================================================

BOW\_500 | Accuracy: 0.5379 | F1: 0.2930

BOW\_1000 | Accuracy: 0.5372 | F1: 0.3124

Negation | Accuracy: 0.5030 | F1: 0.1508

Sentiment | Accuracy: 0.5373 | F1: 0.2910

Combined | Accuracy: 0.5614 | F1: 0.3528

https://colab.research.google.com/drive/1boBmdTUMGPJbbyg3hwJeQIuWsNlTBT2M#scrollTo=26cce89b

Page 13 of 18John

\_Apel\_FInalProject\_664.ipynb - Colab

6/3/25, 2:23 PM

# Final analysis on a small test set

print(f"\n{'='\*50}")

print("FINAL MODEL ANALYSIS")

print(f"{'='\*50}")

# Use combined features for final analysis

sample\_indices = random.sample(range(len(features\_combined)), 2000)

X\_sample = [features\_combined[i] for i in sample\_indices]

y\_sample = [df['Sentiment'].iloc[i] for i in sample\_indices]

# Convert to matrix

vec\_final = DictVectorizer(sparse=True)

X\_final = vec\_final.fit\_transform(X\_sample)

y\_final = np.array(y\_sample)

# Train-test split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X\_final, y\_final, test\_size=0.3, random\_sta

# Train final model

final\_model = LogisticRegression(max\_iter=500, random\_state=42, solver='liblinear')

final\_model.fit(X\_train, y\_train)

y\_pred = final\_model.predict(X\_test)

print("Final Model Performance:")

print(classification\_report(y\_test, y\_pred))

# Feature importance (top coefficients)

https://colab.research.google.com/drive/1boBmdTUMGPJbbyg3hwJeQIuWsNlTBT2M#scrollTo=26cce89b

Page 14 of 18John

\_Apel\_FInalProject\_664.ipynb - Colab

# Feature importance (top coefficients)

feature\_names = vec\_final.get\_feature\_names\_out()

if hasattr(final\_model, 'coef\_'):

# For binary classification, take absolute values

if len(final\_model.coef\_) == 1:

coeffs = np.abs(final\_model.coef\_[0])

else:

coeffs = np.abs(final\_model.coef\_).mean(axis=0)

# Get top features

top\_indices = np.argsort(coeffs)[-15:]

print(f"\nTop 15 Most Important Features:")

for idx in reversed(top\_indices):

print(f"{feature\_names[idx]}: {coeffs[idx]:.4f}")

print(f"\n{'='\*50}")

print("MEMORY-EFFICIENT ANALYSIS COMPLETE!")

print(f"{'='\*50}")

print("Key optimizations made:")

print("1. Reduced vocabulary size (500/1000 vs 1000/2000)")

print("2. Simplified feature functions")

print("3. Used sparse matrices")

print("4. Processed data in batches")

print("5. Used garbage collection")

print("6. Sampled large datasets")

print("7. Reduced cross-validation folds")

6/3/25, 2:23 PM

https://colab.research.google.com/drive/1boBmdTUMGPJbbyg3hwJeQIuWsNlTBT2M#scrollTo=26cce89b

Page 15 of 18John

\_Apel\_FInalProject\_664.ipynb - Colab

6/3/25, 2:23 PM

==================================================

FINAL MODEL ANALYSIS

==================================================

Final Model Performance:

precision recall f1-score support

0 0.50 0.08 0.14 25

1 0.46 0.22 0.30 100

2 0.62 0.86 0.72 315

3 0.41 0.34 0.37 115

4 0.59 0.22 0.32 45

accuracy 0.57 600

macro avg 0.52 0.34 0.37 600

weighted avg 0.55 0.57 0.53 600

Top 15 Most Important Features:

vader\_compound\_score: 1.1169

has\_predictable: 0.8495

has\_way: 0.7303

has\_moving: 0.7013

has\_movies: 0.6864

has\_cinema: 0.6702

has\_anything: 0.6676

has\_full: 0.6581

has\_memorable: 0.6467

has\_less: 0.6399

has\_yet: 0.6294

has\_hollywood: 0.6200

has\_likely: 0.6186

has\_like: 0.6180

has\_something: 0.6146

==================================================

MEMORY-EFFICIENT ANALYSIS COMPLETE!

==================================================

Key optimizations made:

https://colab.research.google.com/drive/1boBmdTUMGPJbbyg3hwJeQIuWsNlTBT2M#scrollTo=26cce89b

Page 16 of 18John

\_Apel\_FInalProject\_664.ipynb - Colab

Key optimizations made:

1. Reduced vocabulary size (500/1000 vs 1000/2000)

2. Simplified feature functions

3. Used sparse matrices

4. Processed data in batches

5. Used garbage collection

6. Sampled large datasets

7. Reduced cross-validation folds

6/3/25, 2:23 PM

from sklearn.metrics import classification\_report, confusion\_matrix

from sklearn.feature\_extraction import DictVectorizer

# Recreate the feature matrix using combined features

print("Recreating feature matrix for evaluation...")

vec\_eval = DictVectorizer(sparse=True)

X\_features = vec\_eval.fit\_transform(features\_combined)

y = df['Sentiment'].values # Ensure y is a numpy array

# Evaluate on test set

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X\_features, y, test\_size=0.2, random\_state=

model = LogisticRegression(max\_iter=500)

model.fit(X\_train, y\_train)

y\_pred = model.predict(X\_test)

# Print evaluation

print("Accuracy:", model.score(X\_test, y\_test))

print(classification\_report(y\_test, y\_pred))

print("Confusion Matrix:")

print(confusion\_matrix(y\_test, y\_pred))

Recreating feature matrix for evaluation...

Accuracy: 0.5768

precision recall f1-score support

0 0.41 0.14 0.21 457

1 0.42 0.25 0.32 1731

2 0.63 0.85 0.72 5034

3 0.50 0.39 0.44 2170

4 0.48 0.20 0.29 608

accuracy 0.58 10000

macro avg 0.49 0.37 0.39 10000

weighted avg 0.54 0.58 0.54 10000

Confusion Matrix:

[[ 65 153 199 35 5]

[ 55 435 1100 125 16]

[ 26 314 4302 372 20]

[ 11 103 1119 842 95]

[ 1 24 147 312 124]]

https://colab.research.google.com/drive/1boBmdTUMGPJbbyg3hwJeQIuWsNlTBT2M#scrollTo=26cce89b

Page 17 of 18John

\_Apel\_FInalProject\_664.ipynb - Colab

6/3/25, 2:23 PM

cv\_results = cross\_validate(model, X\_features, y, cv=3, scoring=['accuracy', 'f1\_macro'])

print("Cross-validated Accuracy:", cv\_results['test\_accuracy'].mean())

print("Cross-validated F1 Score:", cv\_results['test\_f1\_macro'].mean())

Cross-validated Accuracy: 0.5779399328610092

Cross-validated F1 Score: 0.3887909361156469

https://colab.research.google.com/drive/1boBmdTUMGPJbbyg3hwJeQIuWsNlTBT2M#scrollTo=26cce89b

Page 18 of 18