Classification of Kaggle Movie Reviews

By

S M Nawazish K

Table of Contents

1. Introduction
2. Dataset Description
3. Implementation
    - 3.1 Data Loading
    - 3.2 Data Preprocessing
        - Step 1: Lemmatization, Tokenization, and Negation Handling
        - Step 2: Remove Duplicates
        - Step 3: Preprocess and Store Text
    - 3.3 Feature Engineering
        - Step 1: TF-IDF Features
        - Step 2: Sentiment Lexicon Features
        - Step 3: POS Tagging Features
        - Step 4: Combine Features
    - 3.4 Addressing Class Imbalance
    - 3.5 Model Training and Evaluation
4. Observations

1. Introduction

The goal of this project is to classify movie reviews into sentiment categories (0-4) based on their textual content. By using the Kaggle movie reviews dataset, we preprocess text, engineer advanced features, and apply multiple machine learning models to improve classification accuracy.

Sentiment Categories

- 0: Negative
- 1: Strong Negative
- 2: Neutral
- 3: Positive
- 4: Strong Positive

Our approach integrates preprocessing, feature engineering, and advanced machine learning models while addressing class imbalances using oversampling techniques.

2. Dataset Description

Training Data

- Size: 156,060 entries labeled with sentiment values (0-4).
- Columns:
    - Phrase: The review text.
    - PhraseId: Unique identifier for each phrase.
    - SentenceId: Identifier for sentences containing the phrase.
    - Sentiment: Target labels.

Test Data

- Size: 66,292 entries.
- No sentiment labels provided.

Sentiment Lexicon

- A subjectivity lexicon containing words annotated with positive or negative sentiment used to extract features.

3. Implementation

3.1 Data Loading

What We Did

train_file_path = 'D:/NLP/FINAL PROJ/FinalProjectData/kagglemoviereviews/corpus/train.tsv'

test_file_path = 'D:/NLP/FINAL PROJ/FinalProjectData/kagglemoviereviews/corpus/test.tsv'

lexicon_file_path = 'D:/NLP/FINAL PROJ/FinalProjectData/kagglemoviereviews/SentimentLexicons/subjclueslen1-HLTEMNLP05.tff'

train_data = pd.read_csv(train_file_path, sep='\t')

test_data = pd.read_csv(test_file_path, sep='\t')

Why We Did It

- Purpose: Load training and test datasets for preprocessing and analysis.

The Approach Before

- Initially, datasets were loaded without specifying the separator (sep='\t'), causing parsing issues.

3.2 Data Preprocessing

Step 1: Lemmatization, Tokenization, and Negation Handling

What We Did

lemmatizer = WordNetLemmatizer()

def preprocess_text_with_negation(text):

    if not isinstance(text, str):

        return ""

    tokens = word_tokenize(text.lower())

```
    tokens = [word for word in tokens if word.isalnum() and word not in
stopwords.words('english')]
    tokens = [lemmatizer.lemmatize(word) for word in tokens]
    # Negation handling
    for i in range(len(tokens) - 1):
        if tokens[i] in ['not', 'no', "don't", "won't"]:
            tokens[i] = tokens[i] + '_' + tokens[i + 1]
            tokens[i + 1] = ''
    tokens = [word for word in tokens if word]
    return ' '.join(tokens)
```

Why We Did It

- Lowercasing: Standardizes tokens and avoids duplication (e.g., "Movie" and "movie").
- Tokenization: Breaks text into smaller units (words).
- Stopword Removal: Removes common words (e.g., "the", "is") that do not add value.
- Lemmatization: Converts words to their base forms (e.g., "running" → "run").
- Negation Handling: Captures sentiment more effectively by appending negation terms to the next word.

The Approach Before

- Negation handling was not implemented, leading to misclassification of phrases like "not good."

Step 2: Remove Duplicates

What We Did

```
train_data = train_data.drop_duplicates(subset=['Phrase']).reset_index(drop=True)
y_train = train_data['Sentiment'].reset_index(drop=True)
```

Why We Did It

- To reduce redundancy and ensure unique data points in training.

The Approach Before

- Duplicate phrases were left in the dataset, causing redundancy.

Step 3: Preprocess and Store Text

What We Did

train_data['ProcessedText'] = train_data['Phrase'].apply(preprocess_text_with_negation)

test_data['ProcessedText'] = test_data['Phrase'].apply(preprocess_text_with_negation)

Why We Did It

- To preprocess both datasets consistently for feature extraction.

The Approach Before

- Test data was left unprocessed, leading to inconsistencies during evaluation.

3.3 Feature Engineering

Step 1: TF-IDF Features

What We Did

tfidf_vectorizer = TfidfVectorizer(max_features=1000, ngram_range=(1, 2))

X_train_tfidf = tfidf_vectorizer.fit_transform(train_data['ProcessedText']).toarray()

X_test_tfidf = tfidf_vectorizer.transform(test_data['ProcessedText']).toarray()

Why We Did It

- TF-IDF: Assigns weights to words based on their importance.
- Unigrams and Bigrams: Captures both single-word and two-word patterns.

The Approach Before

- Only unigrams with basic Bag-of-Words representation were used.

Step 2: Sentiment Lexicon Features

What We Did

```
def load_subjectivity_lexicon(path):
    pos_words, neg_words = set(), set()
    with open(path, 'r') as file:
        for line in file:
            if "priorpolarity=positive" in line:
                pos_words.add(line.split()[2].split('=')[1])
            elif "priorpolarity=negative" in line:
                neg_words.add(line.split()[2].split('=')[1])
    return pos_words, neg_words


pos_words, neg_words = load_subjectivity_lexicon(lexicon_file_path)


def sentiment_features(text):
    tokens = text.split()
    pos_count = sum(1 for word in tokens if word in pos_words)
    neg_count = sum(1 for word in tokens if word in neg_words)
    return [pos_count, neg_count]


train_data['LexiconFeatures'] = train_data['ProcessedText'].apply(sentiment_features)
test_data['LexiconFeatures'] = test_data['ProcessedText'].apply(sentiment_features)
```

Why We Did It

- To capture sentiment-specific information using positive and negative word counts.

The Approach Before

- No lexicon-based features were included.

Step 3: POS Tagging Features

What We Did

```
def pos_features(text):
    tokens = word_tokenize(text)
    pos_tags = nltk.pos_tag(tokens)
    noun_count = sum(1 for _, tag in pos_tags if tag.startswith('NN'))
    verb_count = sum(1 for _, tag in pos_tags if tag.startswith('VB'))
    adj_count = sum(1 for _, tag in pos_tags if tag.startswith('JJ'))
    return [noun_count, verb_count, adj_count]


train_data['POSFeatures'] = train_data['ProcessedText'].apply(pos_features)
test_data['POSFeatures'] = test_data['ProcessedText'].apply(pos_features)
```

Why We Did It

- To include syntactic features (e.g., nouns, verbs, adjectives) for better classification.

The Approach Before

- Syntactic features were not included, leading to a loss of valuable context.

Step 4: Combine Features

What We Did

```
X_train_combined = pd.concat([
    pd.DataFrame(X_train_tfidf),
    pd.DataFrame(train_data['LexiconFeatures'].tolist()),
    pd.DataFrame(train_data['POSFeatures'].tolist())
], axis=1).values


X_test_combined = pd.concat([
    pd.DataFrame(X_test_tfidf),
    pd.DataFrame(test_data['LexiconFeatures'].tolist()),
    pd.DataFrame(test_data['POSFeatures'].tolist())
], axis=1).values
```

Why We Did It

- To unify different features into a comprehensive representation for model training.

The Approach Before

- Only single-feature sets (e.g., TF-IDF) were used.

3.4 Addressing Class Imbalance

What We Did

smote = SMOTE(random_state=42)

X_train_smote, y_train_smote = smote.fit_resample(X_train_combined, y_train)

Why We Did It

- To balance minority classes using oversampling.

The Approach Before

- Class imbalance led to poor performance on underrepresented classes.

3.5 Model Training and Evaluation

Results

- Naive Bayes (TF-IDF Only):
    - F1 Scores: [0.442, 0.449, 0.446]
    - Mean F1 Score: 0.446
- Logistic Regression (Combined Features):
    - F1 Scores: [0.552, 0.555, 0.558]
    - Mean F1 Score: 0.555
- Random Forest (Tuned):
    - Best Parameters: {'n_estimators': 200, 'max_depth': None}

4. Observations

- Combined features outperformed single-feature models.
- SMOTE improved performance on minority classes.
- Logistic Regression demonstrated the best results, achieving an F1 score of 0.555.