Opinion Mining Project

Contents

1 Sentiment Analysis Approaches

Sentiment analysis is a crucial task in opinion mining, focusing on determining the sentiment expressed in a given text. The approaches to sentiment analysis can generally be categorized into **rule-based methods** and **machine learning-based methods**.

1. Rule-Based Approach

- Relies on predefined linguistic rules and sentiment lexicons.
- Words and phrases are assigned sentiment scores (e.g., "excellent" = +2, "terrible" = -2).
- Overall sentiment is calculated by combining these scores, often adjusted for factors like negation (e.g., "not good") and intensity (e.g., "very bad").
- Popular tools for rule-based sentiment analysis include **VADER** and **TextBlob**.

2. Machine Learning-Based Approaches

Traditional Machine Learning: In this approach, text is converted into numerical features using techniques like Bag-of-Words (BoW) or TF-IDF, and machine learning models are trained to classify sentiment. Common algorithms used include:

- Naive Bayes
- Support Vector Machines (SVM)
- Logistic Regression
- Linear Regression

This approach requires extensive feature engineering but often provides good results for structured datasets.

3. Handling Linguistic Challenges

Sentiment analysis models need to address several linguistic complexities, such as:

- **Negation:** Phrases like "not good" or "wasn't terrible" can invert sentiment. Both rule-based and machine learning models use heuristics or training data to handle negation.

- **Sarcasm:** Sarcasm often expresses the opposite of the literal meaning (e.g., “I love waiting for hours!”), making it difficult for models to detect.
- **Idiomatic Expressions:** Expressions like “kick the bucket” or “break the ice” carry meanings that differ from the literal translation, which can be challenging for both rule-based and traditional machine learning approaches.

Summary Table

Approach	**Description**
Rule-Based	Uses predefined lexicons and linguistic rules to calculate sentiment scores. Simple, interpretable, but less flexible.
Traditional Machine Learning	Converts text into numerical features (TF-IDF, BoW) and uses classifiers such as SVM, Naive Bayes, or Logistic Regression. Requires feature engineering but performs well on structured data.

2 Steps of Opinion Mining

Opinion mining, also known as sentiment analysis, involves a series of steps to extract and classify sentiments from text data. Here are the key steps involved:

1. **Data Collection:** In this step, I started by testing a small dataset generated from AI to understand the different sentiment cases and get familiar with the data.
2. **Text Preprocessing:** After collecting the dataset, I started with cleaning and preparing the text data before feeding it into any model. Here’s the exact sequence I followed:
    - **Lowercasing:** First, I converted all the text to lowercase. This helped avoid treating the same word differently just because of letter casing — for example, “Happy” and “happy” would be counted as the same.
    - **Punctuation and Symbol Removal:** Then, I removed all punctuation marks and symbols like commas, exclamation marks, hashtags, etc. This made the text cleaner and easier to process.
    - **Stopword Removal:** After that, I removed common stopwords like “is”, “the”, and “in” because they don’t carry meaningful sentiment and would just add noise to the model.

- **Emoji Removal:** Since the data included emojis, I used a Python library to strip them out. Emojis can be useful in some cases, but for my model they weren't necessary and could affect the tokenization.
- **Tokenization:** I tokenized the text — which means I broke each sentence down into individual words or tokens. This step was crucial for feeding the text into the machine learning pipeline.
- **Lemmatization:** I reduced each word to its base form using POS tagging and the WordNet lemmatizer to improve text consistency.

3 Subjectivity Detection

Before analyzing whether a sentence is positive or negative, I needed to answer a more basic question: *Is this sentence even expressing an opinion?* That's where subjectivity detection comes in.

3.1 What is Subjectivity Detection?

Subjectivity detection is the process of determining whether a sentence is **subjective** (expressing personal opinions or feelings) or **objective** (stating factual information). For example:

- **Subjective:** "I absolutely loved the plot twist."
- **Objective:** "The movie was released in 2021."

This step is crucial because analyzing factual sentences that don't express opinions would only add noise to the sentiment analysis process.

3.2 Why is it Important?

While going through data like reviews or social media posts, I noticed that not every sentence contains an opinion. Some are just plain facts — like the name of an actor or the runtime of a film. By filtering these out, I was able to:

- Focus on content that actually carries opinions.
- Reduce irrelevant information.
- Improve the overall accuracy of the sentiment classifier.

3.3 How I Did It

I used the labeled dataset provided by the `nltk.corpus.subjectivity` module. This dataset contains:

- 500 subjective sentences.
- 500 objective sentences.

I trained a **Naive Bayes Classifier** using this data. Each sentence was converted into a set of features using a bag-of-words approach — meaning that each word was treated as a binary feature (present or not in the sentence).

3.4 Examples of Classification

Here are a few real examples I tested with the classifier:

```
Input: "The cinematography was breathtaking."
→ Subjective

Input: "The film is two hours long."
→ Objective
```

4 Text Representation

After preprocessing and subjectivity detection, the next crucial step was to convert the text data into numerical features that machine learning models can understand.

4.1 Bag-of-Words (BoW) Representation

I first used the `CountVectorizer` from `scikit-learn` to apply the Bag-of-Words technique. This method counts the occurrences of each unique word (feature) across the dataset. Each sentence is then represented as a vector of word counts. For example, if the vocabulary consists of *awesome*, *bad*, and *movie*, a sentence like "awesome movie" would be represented as `[1, 0, 1]`.

Using this method, I created a feature matrix `X_bow` and saved the resulting feature vectors into a CSV file for further analysis.

4.2 TF-IDF Representation

To better capture the importance of words in the context of the whole dataset, I also applied the Term Frequency-Inverse Document Frequency (TF-IDF) method using `TfidfVectorizer`. Unlike BoW, TF-IDF weighs down very common words and highlights rarer but more informative words. For example, words like "the" or "is" receive low scores, while sentiment-laden words like "awesome" or "disappointed" receive higher weights.

This resulted in a TF-IDF feature matrix `X_tfidf`, which was converted to a DataFrame for inspection and further processing.

—

Both representations provide numerical formats suitable for feeding into classifiers and help capture the textual information in different ways — BoW focusing on raw frequency counts, and TF-IDF emphasizing relative importance.

In the end, I chose to use the Bag-of-Words (BoW) representation for training my models because it performed well on my dataset and was simpler to interpret.

5 Opinion Positivity Detection

After transforming the textual data into numerical features, the next step was to label the sentiment polarity (positive, negative, or neutral) of each sentence. This step provided supervised labels for training and evaluating machine learning models.

5.1 Using TextBlob for Polarity Detection

To perform polarity-based sentiment classification, I used the `TextBlob` library — a simple yet powerful NLP library in Python. TextBlob provides a built-in `sentiment` function that returns two key scores for any given text:

- **Polarity:** A float within the range `[-1.0, 1.0]` indicating the sentiment. A score greater than 0 indicates positive sentiment, less than 0 indicates negative sentiment, and exactly 0 indicates a neutral tone.
- **Subjectivity:** A float within `[0.0, 1.0]`, though in this case I focused only on polarity.

Using this polarity score, I defined a custom function `get_sentiment(text)` that classifies each sentence into one of three categories: `positive`, `negative`, or `neutral`.

5.2 Workflow Summary

- I read the cleaned and lemmatized dataset from `test_sample.csv`.
- I used the precomputed Bag-of-Words features stored in `bow_features.csv`.
- I applied TextBlob's polarity scoring function to each sentence in the original text column.
- The resulting sentiment labels were stored in a new column called `label`.
- I saved the updated DataFrame back to the same CSV file for consistency in future steps.

This labeling step helped provide a ground truth for training and validating the sentiment classification model, and it also allowed for basic evaluation and visualization of the dataset's sentiment distribution.

6 Sentiment Classification Using Logistic Regression

After labeling the data using TextBlob, I moved on to train a machine learning model to classify sentiments automatically. I chose **Logistic Regression** as my baseline classifier due to its simplicity, interpretability, and strong performance on linear problems.

6.1 Data Preparation

- I encoded the sentiment labels (`positive`, `negative`, `neutral`) into numerical form using `LabelEncoder`.
- I applied `TfidfVectorizer` to the cleaned text (`clean_text` column), using a large feature space of unigrams and bigrams (`ngram_range=(1, 2)`), with up to 100,000 features.
- The dataset was split into training and testing sets with a 90-10 split, maintaining class balance via stratified sampling.

6.2 Model Training and Fine-Tuning

I trained the Logistic Regression model with the following settings:

- `solver='saga'` for efficient performance with large sparse datasets.
- `max_iter=1000` to allow for sufficient convergence.

- `n_jobs=-1` to parallelize computation across CPU cores.

After training, the model was tested on the held-out test set.

6.3 Evaluation Metrics and Results

I evaluated the classifier using accuracy and the classification report (precision, recall, F1-score). The performance was as follows:

- **Accuracy:** Achieved satisfactory accuracy on the test set.
- **Class-wise performance:** The classifier performed well on `positive` and `negative` classes, with slightly lower precision on the `neutral` class, which is expected due to the subtle nature of neutral expressions.

6.4 Confidence-Based Post-Processing

To improve interpretability on custom inputs, I implemented confidence thresholding:

- For each sample, I extracted the predicted probabilities from the model.
- If the maximum confidence was between 0.45 and 0.65, I labeled the result as `neutral`, regardless of the top class.
- Otherwise, the top class was used as the final prediction.

6.5 Sample Predictions

Here are some examples from real-world sample inputs:

```
"This is the best service I've ever had!" → positive (0.98 confidence)
"Honestly, just okay." → neutral (0.54 confidence)
"Worst. Pizza. Ever." → negative (0.99 confidence)
```

6.6 Comparison with Transformer-Based Model

To benchmark my Logistic Regression model, I also tested a state-of-the-art pre-trained transformer — `cardiffnlp/twitter-roberta-base-sentiment`. This model was fine-tuned specifically on social media sentiment data.

Using Hugging Face's `transformers` library, I:

- Tokenized the text samples using RoBERTa's tokenizer.

- Ran inference through the transformer model to get softmax-normalized probabilities.
- Mapped predictions to `positive`, `neutral`, or `negative`.

6.7 Transformer Results

Example outputs:

```
Transformer: "Amazing work @brand, you nailed it!" → positive (0.95 confidence)
Transformer: "It was fine. Not great, not awful." → neutral (0.61 confidence)
Transformer: "Absolutely terrible. Never ordering again." → negative (0.98 confidence)
```

6.8 Conclusion

While the transformer-based model provided high-quality results, the Logistic Regression model also achieved competitive performance, especially when combined with fine-tuned TF-IDF features and post-processing heuristics. This demonstrated that traditional machine learning models, when carefully engineered, can still be highly effective for sentiment classification tasks.