# **NLP Project #1: Movie Review Text Classification**

Contents

[**NLP Project #1: Movie Review Text Classification** 1](#_Toc199769552)

[**Project Summary** 2](#_Toc199769553)

[**Prerequisites** 4](#_Toc199769554)

[**Loading Data** 4](#_Toc199769555)

[**Preprocessing** 5](#_Toc199769556)

[**Feature Extraction** 8](#_Toc199769557)

[**Model Training** 10](#_Toc199769558)

## **Project Summary**

**Resume format**

Developed an NLP-based movie review text classification system utilizing Python, NLTK, TF-IDF with N-grams, and Logistic Regression, achieving robust sentiment analysis.

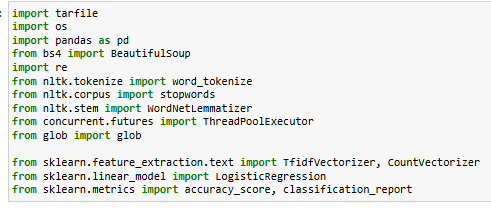
**Full Summary**

This project tackles movie review text classification, aiming to automatically determine the sentiment (positive or negative) of a given movie review. The core process involves several key NLP stages:

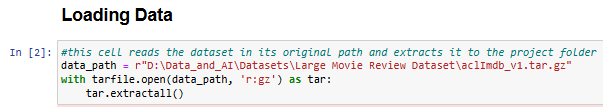
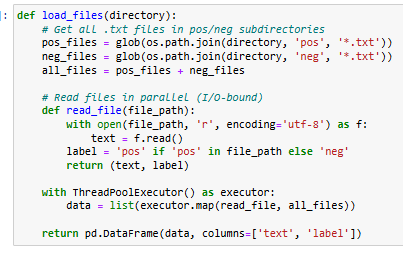
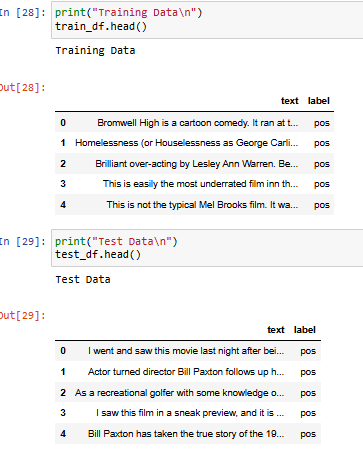
* **Prerequisites & Data Loading:** The project utilizes the NLTK (Natural Language Toolkit) Python library. Data is loaded efficiently using tarfile for extraction and glob with ThreadPoolExecutor for parallelized file reading.
* **Preprocessing:** Raw text undergoes a thorough cleaning process. This includes extracting text from HTML, removing non-alphabetic characters and extra whitespaces, tokenization (breaking text into words), removing stop words (common words like "the," "a"), and lemmatization (reducing words to their base dictionary form). While lemmatization for verbs like "ate" to "eat" requires part-of-speech (POS) tagging and isn't always critical for sentiment analysis, it's considered.
* **Feature Extraction:** To convert text into a numerical format suitable for machine learning models, two primary methods are explored:
  + Bag of Words (BoW): Represents text as an unordered collection of word counts, useful for basic classification but can create large, sparse vectors.
  + TF-IDF (Term Frequency-Inverse Document Frequency): A more sophisticated method that weighs words by their importance in a document relative to the entire corpus, downplaying common words and highlighting rare, meaningful ones. TF-IDF is generally preferred for better accuracy.
  + N-grams: The project emphasizes the importance of N-grams (sequences of N words, e.g., "not good") to capture local context, handle negations, and recognize phrases that single words might miss. Using a TfidfVectorizer with ngram\_range=(1, 3) generates unigrams, bigrams, and trigrams, significantly boosting model performance, especially in sentiment analysis, despite increasing computational cost and feature space.
* **Model Training:** 
  + Logistic Regression is employed as the classification model.
  + Hyperparameter tuning is performed using 5-fold Cross-Validation to obtain a robust estimate of the model's generalization performance. The C hyperparameter, which controls regularization strength, is specifically tuned.
  + Support Vector Machines (SVM) are also discussed as an alternative, focusing on finding an optimal hyperplane to separate classes, potentially using the "kernel trick" to handle non-linearly separable data.
* **Implementation Details:** The project notes the use of sublinear\_tf=True in TF-IDF for log scaling of term frequencies, which helps reduce the dominance of overly frequent words. It also highlights how positive coefficients in a binary logistic regression indicate positive sentiment, and in multiclass scenarios, each class has its own binary classifier with associated coefficients.

This project demonstrates a comprehensive approach to text classification, from initial data preparation to advanced feature engineering and model training, with a strong emphasis on practical considerations for sentiment analysis.

## **Prerequisites**

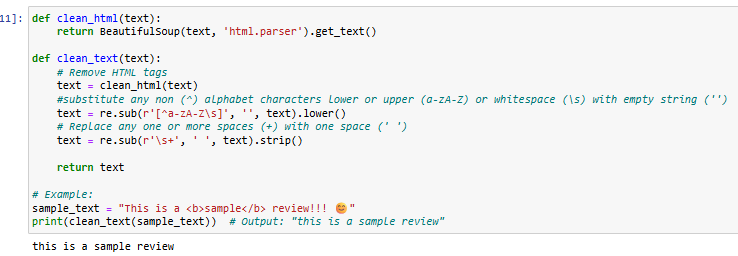
* NLTK (Natural Language Toolkit) is a python library needed for the project and some of its content needs to be downloaded first.
  + 
  + 

## **Loading Data**

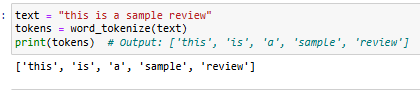
* Tarfile is used to extract the data first.  
  
* The following function loads the data quickly due to using glob, which avoids nested loops by using wild cards (matching pattern like this \*.txt means to get all files that end in .txt) as well as parallelizing file reading by using multiple threads with ThreadPoolExecutor   
  
* Data shape:  
  

## **Preprocessing**

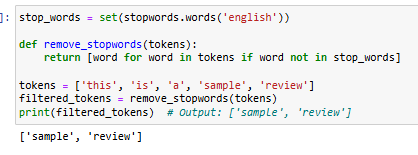
The following functions clean the text by extracting text only from the scraped html, removes any characters that aren’t alphabetical or white spaces, then it finally removes extra white spaces.



**Tokenization**:



**Removing Stop Words:**



**Lemmatization**: stripping words into their dictionary form



Q:

**Why doesn’t Lemmatization change “ate” into “eat”, and should we do it anyway?**

A:

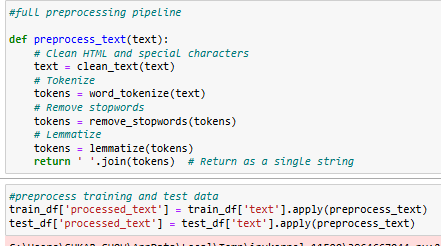
Lemmatization reduces words to their *dictionary form* (lemma), which depends on **part-of-speech (POS) tagging**.

1. **"dogs" → "dog"** works because it's just plural → singular (noun form).
2. **"ate" → "eat"** *would* work, **but only if you specify the word is a verb** (POS tag). By default, many lemmatizers assume nouns, so "ate" stays unchanged.

**Should we do it?**

* **For sentiment analysis:** Usually not worth the effort. The base form ("eat") often carries similar sentiment as past tense ("ate").
* **For other NLP tasks:** Maybe, if tense matters (e.g., question answering).

**Full preprocessing pipeline:**



## **Feature Extraction**

**Bag of Words (BoW)**

**What it is:**

* Represents text as a "bag" (unordered collection) of word counts.
* Ignores grammar/order but captures word frequency.

**How it works:**

1. Create a **vocabulary** of unique words from all documents.
2. For each document, count how many times each word appears.

**Example:**

* Documents:
  + Doc1: "I love cats"
  + Doc2: "I hate dogs"
* BoW (vocabulary order: ["I", "love", "cats", "hate", "dogs"]):
  + Doc1: [1, 1, 1, 0, 0]
  + Doc2: [1, 0, 0, 1, 1]

**Pros:**

* Simple and fast.
* Works well for basic classification.

**Cons:**

* Treats common words (e.g., "the") as equally important as rare ones.
* Creates very large, sparse vectors (memory-heavy).

**TF-IDF (Term Frequency-Inverse Document Frequency)**

**What it is:**

* Weighs words by their **importance** in a document relative to the entire corpus.
* Downweighs frequent, generic words (e.g., "and", "the").

**How it works:**

* **Term Frequency (TF):** (Number of times a word appears in a document) / (Total words in the document)
* **Inverse Document Frequency (IDF):** log(Total documents / Number of documents containing the word)
* **TF-IDF Score:** TF \* IDF

**Example:**

* Doc1: "I love cats"
* Doc2: "I hate dogs"
* Doc3: "Cats hate dogs"
* TF-IDF for "hate" in Doc2:
  + TF: 1/3 (appears once in a 3-word doc)
  + IDF: log(3/2) (appears in 2 out of 3 docs)
  + TF-IDF: (1/3) \* log(3/2) ≈ 0.06

**Pros:**

* Highlights rare, meaningful words (e.g., "awesome", "terrible").
* Better for tasks like search engines or sentiment analysis.

**Cons:**

* Slightly slower than BoW.

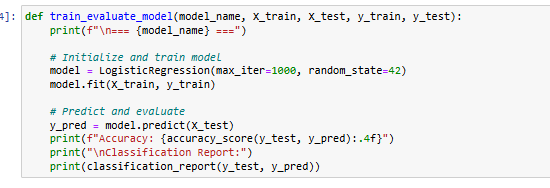
**When to Use**

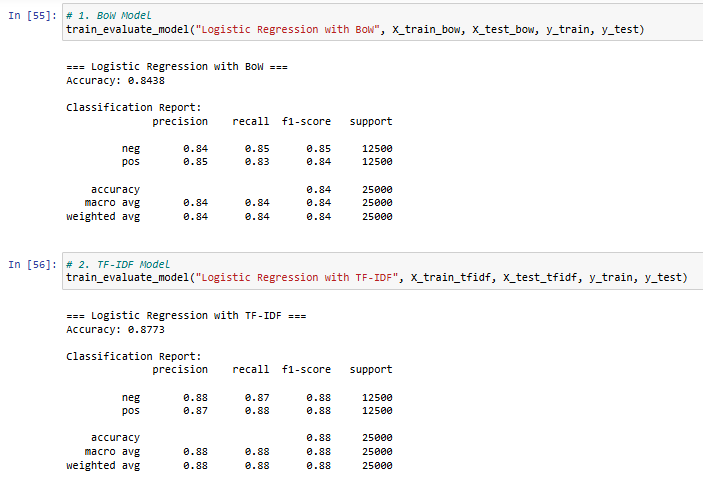
* **BoW:** Quick baseline models, small datasets.
* **TF-IDF:** Better accuracy for most NLP tasks (e.g., sentiment analysis).



## **Model Training**

**Logistic Regression**



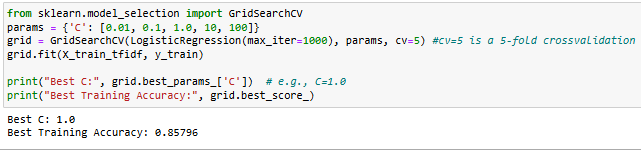


**Hyperparameter tuning**

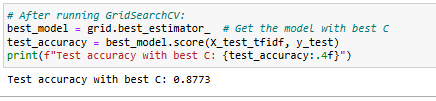
5-fold Cross Validation

The training data is split into 5 equal parts (folds). The model is trained on 4 folds and validated on the remaining fold. This process repeats 5 times, with each fold serving as the validation set once.

Performance metrics (e.g., validation accuracy) are averaged across all 5 runs to estimate generalization.

****

**What is C?**

* C is a **hyperparameter**, it's the **inverse of** **regularization** **strength**, so a smaller C means a stronger regularization. Its default value is actually 1.0 (which turns out to result in the best accuracy)
* Since the 5-fold cross validation averages the validation accuracies over the 5 iterations, it gives a more conservative estimate of accuracy than just one run for the model, that's why we end up with 85% validation accuracy as opposed to 87% we get when we evaluate the model on the test set.
* ****

**Using N-grams**

**1. What are N-grams?**

An n-gram is a contiguous sequence of n items (words, characters, or symbols) from a text. In NLP, we typically use word n-grams or character n-grams:

* Unigram (1-gram): Single words (e.g., "movie", "good").
* Bigram (2-gram): Pairs of consecutive words (e.g., "great movie", "not good").
* Trigram (3-gram): Triples of consecutive words (e.g., "best movie ever", "not very good").

**2. Why N-grams Matter**

N-grams capture local context and phrases that single words (unigrams) miss.

Example:

* Sentence: "This movie is not good."
* Unigrams: ["this", "movie", "is", "not", "good"] → Misses negation ("not good").
* Bigrams: ["this movie", "movie is", "is not", "not good"] → Captures "not good" as a phrase.
* Trigrams: ["this movie is", "movie is not", "is not good"] → Captures "is not good".

**3. How N-grams Improve Sentiment Analysis**

* Negation Handling:  
  Bigrams like "not good" or "wasn't great" preserve the sentiment reversal.
* Idioms/Phrases:  
  Trigrams like "out of this world" or "waste of time" carry stronger sentiment than individual words.
* Contextual Meaning:  
  The word "long" might be neutral alone, but "long runtime" (bigram) is negative for movie reviews.

**4. N-grams in Feature Extraction**

When you use TfidfVectorizer(ngram\_range=(1, 3)):

1. Tokenization: Split text into words.
2. Sliding Window: Extract all 1-word, 2-word, and 3-word sequences.
3. Vectorization: Convert these n-grams into TF-IDF scores.

**Example input:** "The plot was boring but the acting was good."

**N-grams Generated:**

* **Unigrams:** ["the", "plot", "was", "boring", "but", "acting", "good"]
* **Bigrams:** ["the plot", "plot was", "was boring", "boring but", ..., "acting was", "was good"]
* **Trigrams:** ["the plot was", "plot was boring", ..., "acting was good"]

**5. Trade-offs of Using N-grams**

|  |  |
| --- | --- |
| **Pros** | **Cons** |
| Captures phrases/negations | Increases feature space (can cause overfitting) |
| Better context for models | Longer n-grams are rare (sparse data) |
| Works well with TF-IDF/BoW | Higher computational/memory costs |

**6. Practical Tips for N-grams**

1. **Start Small:** Use ngram\_range=(1, 2) (unigrams + bigrams) for most tasks.
2. **Filter Rare N-grams:** Use min\_df=5 to ignore n-grams appearing in fewer than 5 documents.
3. **Combine with TF-IDF:** Prioritize n-grams with high TF-IDF scores (e.g., max\_features=15000).
4. **Avoid Long N-grams:** Trigrams/4-grams often add noise unless your dataset is huge.

**7. How N-grams Affect Model Performance**

* **Accuracy Boost:** On IMDB reviews, adding bigrams can improve accuracy by **1-3%** (e.g., 88% → 90%).
* **Example Model Impact**:
  + Logistic Regression (Unigrams): 88%
  + Logistic Regression (Unigrams + Bigrams): 89.5%
  + Linear SVM (Unigrams + Bigrams + Trigrams): 91%

**8. When to Avoid N-grams**

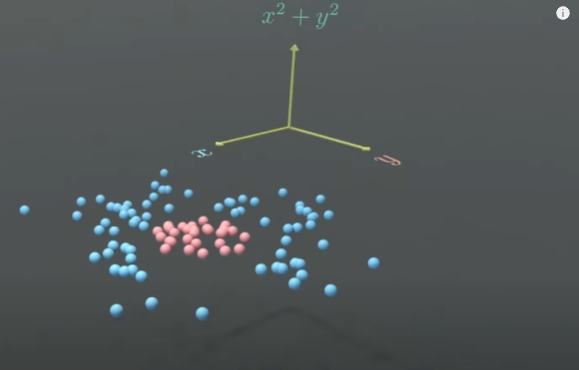
* **Short Texts:** Tweets or product titles (too few words for meaningful n-grams).
* **High-Dimensional Data:** If you can’t afford computational overhead.

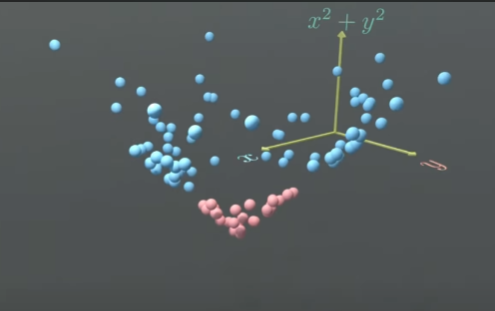
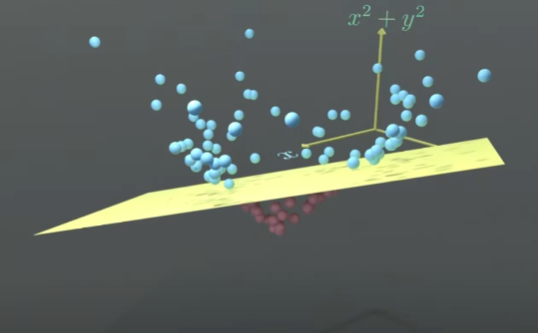
**Support Vector Machines (SVM)**

**Goal:** separate the categories by a hyperplane (a line in 2D, a plane in 3D, etc.) that maximizes the margin between it and the support vectors (the data points that touch the furthest parallel hyperplane.

|  |  |  |
| --- | --- | --- |
|  |  |  |

**Kernel trick:** if the datapoints can’t be separated by a hyperplane, we map the features to a higher dimensional space, we find the separating hyperplane in this high dimensional space, then project back to the original space.

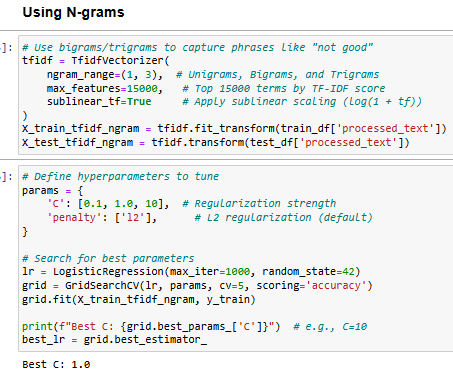
 

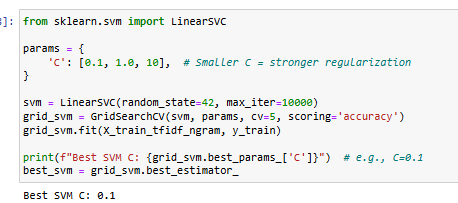
 

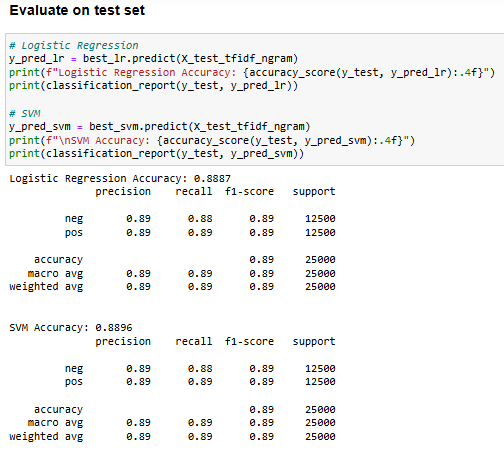
**Implementation**

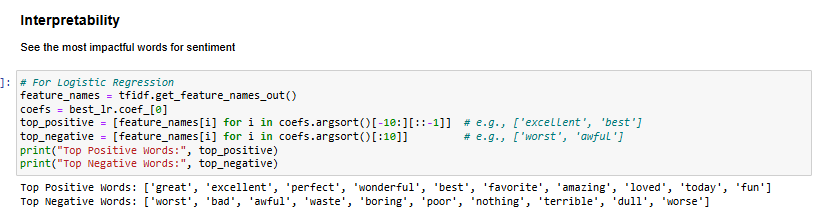
In TFIDF, using “sublinear\_tf=True” applies **log scaling** to term frequencies (TF):

* Instead of raw counts (TF), it uses log(1 + TF).
* **Why?** Reduces the dominance of very frequent words (e.g., "movie" appearing 20x in a review)









* The variable “feature\_names” contains a list all the vocabulary (words, ngrams).
* Coefs is the same size as feature\_names, and it contains weights for each word/ngram. In the case of a binary logistic regression, positive coefficients represent positive sentiment and vice versa.
* In the case of multiclass classification, there’s one binary classifier with its own coefficients associated with each class, and positive weights mean that the sentiment is in favor of that class, while all other classes are associated with negative weights.