Open in app ↗

Search Medium

Get unlimited access to all of Medium for less than $1/week. Become a member

Naive Bayes Classifier for Rotten Tomatoes Reviews

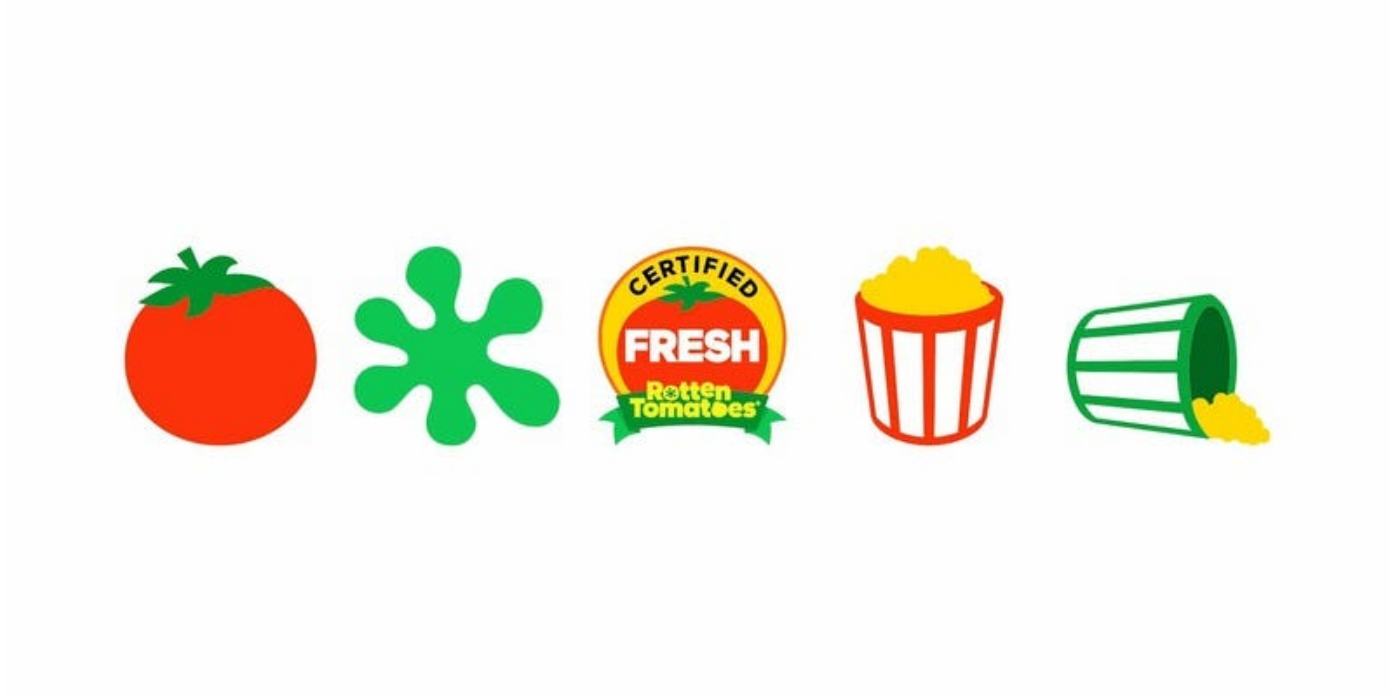
Amoshsapkota

8 min read · Apr 16

Listen Share More

Rotten Tomatoes Reviews

The Rotten Tomatoes Reviews text dataset contains a collection of movie reviews sourced from the Rotten Tomatoes website. It includes the text of the reviews as well as a label indicating whether the review was classified as “fresh” or “rotten” based on Rotten Tomatoes’ review aggregation system. The dataset comprises reviews from a broad spectrum of critics and publications, spanning various movie genres and languages. Each review in the dataset is tagged with a fresh/rotten label, which can be used for supervised learning tasks such as sentiment analysis or classification.



Objective

Our objective is to build a Naive Bayes Classifier that can accurately classify movie reviews as either “fresh” or “rotten” using the Rotten Tomatoes Reviews text dataset. We intend to construct a useful tool for sentiment analysis and classification of movie reviews using the Naive Bayes approach. Through this study, we want to show how well the Naive Bayes algorithm works and how it can be used for sentiment analysis and other tasks involving natural language processing.

Naive Bayes Classifier

The Naive Bayes Classifier is a probabilistic method that is frequently used for classification applications, particularly in text analysis and natural language processing. It derives its name from its “naive” assumption that all features in the input data are independent of each other, despite this not always being true in reality. This assumption, however, enables a simpler and more effective computation technique, making it suitable for big datasets. The algorithm uses the Bayes theorem to determine the likelihood of each feature given the class label and the likelihood of each class given the input features. The class label with the highest probability is the one predicted for the input data.

Building a Naive Bayes Classifier

Step 1: Dataset Preprocessing

The first step I took was to load the rt_reviews.csv dataset from the rotten tomatoes reviews challenge on Kaggle. Next, we split the given dataset into three subsets: training, development, and testing. We chose to split the data in a ratio of 60:20:20, with 60% of the data used for training, 20% for development, and 20% for testing.

```
#Load the dataset from kaggle
#default character encoding of utf-8 failed to decode the file, so used encoding=
df = pd.read_csv("/kaggle/input/rotten-tomatoes-reviews/rt_reviews.csv", encoding=

# Splitting the dataset into train,development,and test
train, dev, test = np.split(df.sample(frac=1, random_state=42), [int(0.6*len(df))
```

After splitting the dataset, I created a vocabulary of words present in the reviews. To reduce the dimensionality of the feature space and prevent overfitting, I removed rare words from the vocabulary that occurred less than 5 times in the entire dataset. The resulting vocabulary is called vocab_filter.

```
# Build a vocabulary as list
vocab = {}
for reviews in train['Review']:
    words = reviews.lower().split()
    for word in words:
        if word in vocab:
            vocab[word] += 1
        else:
            vocab[word] = 1
```

```
# Remove rare words from vocab if occurring is less than 5 times
vocab_filter = {k:v for k,v in vocab.items() if v>=5}
```

To further process the vocabulary, I generated a reverse index for each word. This index assigns a unique integer value to each word in the vocabulary, ranging from 0 to the number of words in the vocabulary minus 1.

```
# Generate reverse index where indices are the integers from 0 to the number of w
reverse_index = {word: i for i, word in enumerate(vocab_filter)}
```

Step 2: Probability Calculation

First, the entire number of documents (reviews) in the training set were counted in the training dataset. This step is required in order to later calculate different probabilities.

Next, the number of reviews labeled as “fresh” and the number labeled as “rotten” were calculated. This is important for calculating the prior probabilities of a reviews being fresh or rotten, which will be used in the Naive Bayes Classifier algorithm.

After that, the count of how many times each word appears in the “fresh” and “rotten” reviews was calculated. This count is used to calculate the conditional probability of each word given the class label (fresh or rotten).

Additionally, the reviews in the train data were counted to see how often the words “fresh” and “rotten” appeared. This count is crucial to the Laplace smoothing procedure because it helps to prevent zero probabilities and guarantees that words that weren’t in the training set won’t receive a zero probability from the Naive Bayes Classifier.

Lastly, Laplace smoothing was performed to adjust the probability estimates for words that appear only a few times in the training data, and prevent zero probabilities from occurring. This helps the Naive Bayes Classifier to make more accurate predictions on the test data.

```
# Calculates the number of reviews in the training set
num_of_documents = len(train)
# Calculate the number of reviews labeled as "fresh", and the number labeled as "
num_of_fresh = len(train[train['Freshness']=='fresh'])
num_of_rotten = len(train[train['Freshness']=='rotten'])
#Calculate the prior probabilities of a document fresh / rotten
pb_fresh = num_of_fresh / num_of_documents
pb_rotten = num_of_rotten / num_of_documents
```

```
#Count how many times each word appears in the "fresh" and "rotten" documents.
word_cnt_fresh = np.zeros(len(vocab_filter))
word_cnt_rotten = np.zeros(len(vocab_filter))
#Count the number of times word "fresh" repeats in reviews in train data
for reviews in train[train['Freshness']=='fresh']['Review']:
    words = reviews.lower().split()
    for word in words:
        if word in vocab_filter:
            word_cnt_fresh[reverse_index[word]] += 1
#Count the number of times word "rotten" repeats in reviews in train data
for reviews in train[train['Freshness']=='rotten']['Review']:
    words = reviews.lower().split()
    for word in words:
        if word in vocab_filter:
            word_cnt_rotten[reverse_index[word]] += 1
#Perform laplace smoothing
pb_of_word_fresh = (word_cnt_fresh + 1) / (np.sum(word_cnt_fresh) + len(vocab_fil
pb_of_word_rotten = (word_cnt_rotten + 1) / (np.sum(word_cnt_rotten) + len(vocab_
```

Step 3: Accuracy Calculation of dev dataset

Moving on to the development dataset, we determined the accuracy using the probability estimates we previously generated. We looped over each row in the development dataset and calculated the probability of it being labeled as “fresh” or “rotten” using the naive Bayes classifier. Then, we determined which class has the larger probability by comparing these probabilities. Then, we calculated the accuracy by comparing the predicted labels with the true labels and dividing the total number of correct predictions by the total number of samples. Here, accuracy came out to be 0.7966666, which means the model correctly classified 79.66 % of the reviews in the development dataset.

```
#calculates the accuracy on the development dataset (dev) using the probability es
import math
correct = 0
total = len(dev)

#calculates the probability that a review will be labeled as "fresh" or "rotten" l

for i, row in dev.iterrows():
    fresh_prob = math.log(pb_fresh)
    rotten_prob = math.log(pb_rotten)
```

```
#determine which class has the larger probability by comparing these probabilities

    words = row['Review'].lower().split()
    for word in words:
        if word in vocab_filter:
            fresh_prob += math.log(pb_of_word_fresh[reverse_index[word]])
            rotten_prob += math.log(pb_of_word_rotten[reverse_index[word]])

    if fresh_prob > rotten_prob:
        prediction = 'fresh'
    else:
        prediction = 'rotten'

    if prediction == row['Freshness']:
        correct += 1

accuracy = correct / total
print("Accuracy:" +str(accuracy))
```

```
Accuracy:0.7966666666666666
```

Step 4: Experiment

In order to improve the performance of our classifier, I performed an experiment by comparing different smoothing factors on the development (dev) dataset. I tried smoothing factors of 0, 0.5, 1, and 5.

```
# Comparing different smoothing factors on development (dev) dataset
smoothing_factors = [0, 0.5, 1, 5]

for sf in smoothing_factors:
    word_cnt_fresh = np.zeros(len(vocab_filter))
    word_cnt_rotten = np.zeros(len(vocab_filter))

    for reviews in train[train['Freshness']=='fresh']['Review']:
        words = reviews.lower().split()
        for word in words:
            if word in vocab_filter:
                word_cnt_fresh[reverse_index[word]] += 1

    for reviews in train[train['Freshness']=='rotten']['Review']:
```

```
            words = reviews.lower().split()
            for word in words:
                if word in vocab_filter:
                    word_cnt_rotten[reverse_index[word]] += 1

        pb_of_word_fresh = (word_cnt_fresh + sf) / (np.sum(word_cnt_fresh) + sf*len(v
        pb_of_word_rotten = (word_cnt_rotten + sf) / (np.sum(word_cnt_rotten) + sf*le

        correct = 0
        for i in range(len(dev)):
            fresh_prob = np.log(pb_fresh)
            rotten_prob = np.log(pb_rotten)
            review_words = dev.iloc[i]['Review'].lower().split()
            for word in review_words:
                if word in vocab_filter:
                    fresh_prob += np.log(pb_of_word_fresh[reverse_index[word]])
                    rotten_prob += np.log(pb_of_word_rotten[reverse_index[word]])
            predicted_freshness = 'fresh' if fresh_prob > rotten_prob else 'rotten'
            if predicted_freshness == dev.iloc[i]['Freshness']:
                correct += 1
        accuracy = correct / len(dev)
        print("Smoothing factor:", sf)
        print("Accuracy:", accuracy)
```

```
Smoothing factor: 0
Accuracy: 0.79315625
Smoothing factor: 0.5
Accuracy: 0.7961458333333333
Smoothing factor: 1
Accuracy: 0.7966666666666666
Smoothing factor: 5
Accuracy: 0.79471875
```

From these results, I concluded that the optimal smoothing factor for our classifier on this dataset was 1, as it resulted in the highest accuracy. I also noticed that the accuracy decreased when the smoothing factor was either increased or decreased from 1.

From these results, I concluded that the optimal smoothing factor for our classifier on this dataset was 1, as it resulted in the highest accuracy. I also noticed that the

accuracy decreased when the smoothing factor was either increased or decreased from 1.

Yet another experiment was done on development dataset to find the top 10 words that predict each class (fresh and rotten). Only strong words with a length of at least 10 were taken into account because short and repetitive words have less of an impact on determining the sentiment of the review.

```
# Derive Top 10 words that predict each class fresh and rotten
#Only powerful words with a length of at least 10 are taken into consideration bec
top_words_fresh = sorted([(word, pb_of_word_fresh[reverse_index[word]]) for word 
top_words_rotten = sorted([(word, pb_of_word_rotten[reverse_index[word]]) for word
```

```
print("Top 10 words that predict fresh reviews are:")
for word, p in top_words_fresh:
    print(f"Word: {word}, P(fresh|{word}): {p:.8f}")

print("Top 10 words that predict rotten reviews are:")
for word, p in top_words_rotten:
    print(f"Word: {word}, P(rotten|{word}): {p:.8f}")
```

```
Top 10 words that predict fresh reviews are:
Word: performance, P(fresh|performance): 0.00082815
Word: characters, P(fresh|characters): 0.00074779
Word: performances, P(fresh|performances): 0.00074567
Word: entertaining, P(fresh|entertaining): 0.00069854
Word: documentary, P(fresh|documentary): 0.00061726
Word: compelling, P(fresh|compelling): 0.00034655
Word: fascinating, P(fresh|fascinating): 0.00033356
Word: everything, P(fresh|everything): 0.00031301
Word: beautifully, P(fresh|beautifully): 0.00030818
Word: experience, P(fresh|experience): 0.00028431
Top 10 words that predict rotten reviews are:
Word: characters, P(rotten|characters): 0.00101604
Word: ultimately, P(rotten|ultimately): 0.00054700
Word: interesting, P(rotten|interesting): 0.00045887
Word: everything, P(rotten|everything): 0.00040863
Word: performance, P(rotten|performance): 0.00036518
Word: performances, P(rotten|performances): 0.00031279
Word: unfortunately,, P(rotten|unfortunately,): 0.00031279
Word: predictable, P(rotten|predictable): 0.00028691
```

```
Word: filmmakers, P(rotten|filmmakers): 0.00025640
Word: completely, P(rotten|completely): 0.00024561
```

Contribution

After conducting hyperparameter tuning on the development dataset, I found that the smoothing factor is the optimal hyperparameter to remove zero probabilities and avoid overfitting issues. Upon testing different values such as 0, 0.5, 1, and 5, the optimal value was found to be 1.When the smoothing parameter is set to 0, no smoothing is performed, and the model only uses the frequencies from the training set to predict probabilities. The influence of rare terms in the data is lessened when the smoothing parameter is 0.5, which denotes moderate smoothing. The Laplace smoothing algorithm, also known as additive smoothing, adds one to each word's count in order to prevent zero probability. When smoothing is heavy, like in the case of a smoothing value of 5, rare words are given larger probabilities. Therefore, I used a smoothing factor of 1 on the test dataset and performed predictions. The accuracy increased slightly to 0.7999, indicating that the model accurately predicted nearly 80% of the reviews in the test dataset.

```
#From hyperparameter tuning in development (dev) dataset, smoothing_factor is cons
smoothing_factor = 1
correct = 0
total = len(test)

for i, row in test.iterrows():
    fresh_prob = math.log(pb_fresh)
    rotten_prob = math.log(pb_rotten)

    words = row['Review'].lower().split()

    #index = reverse_index.get(word, -1) is  used to find the word's index in the
    for word in words:
        if word in vocab_filter:
            index = reverse_index.get(word, -1)
            if index != -1:
                fresh_prob += math.log((word_cnt_fresh[index] + smoothing_factor)
                rotten_prob += math.log((word_cnt_rotten[index] + smoothing_facto

    if fresh_prob > rotten_prob:
        prediction = 'fresh'
    else:
```

```
            prediction = 'rotten'

        if prediction == row['Freshness']:
            correct += 1

    accuracy = correct / total
    print("Final accuracy on test dataset:", accuracy)
```

```
Final accuracy on test dataset: 0.7992604166666667
```

References

Rotten Tomatoes Reviews

Can you predict if a review is rotten or fresh?

www.kaggle.com

GitHub Repository: https://github.com/AmoshSapkota/Rotten-Tomato-Review-Using-Naive-Bayes-Classifier

Edit profile

Written by Amoshsapkota

0 Followers