


Tallapelli Saicharan  2 minutes ago 10 min read

Naïve Bayes Classifier (NBC)

The goal of this blog is to learn about the Naive Bayes Classifier (NBC).

What does Naïve Bayes Classifiers?

A naive Bayes classifier is an algorithm that uses Bayes' theorem to classify objects. Naive Bayes classifiers assume strong, or naive, independence between attributes of data points. Popular uses of naive Bayes classifiers include spam filters, text analysis and medical diagnosis.

In this blog we will use text dataset about the rotten tomato review

Rotten Tomatoes Reviews | Kaggle

Given the review, our goal is predicting whether the review is rotten or fresh.

In this assignment we are not going to use any library to implement the main algorithm of NBC, but we will be using following: pandas is used to read and manipulate the dataset, NumPy is used for numerical operations, and the counter function from collections is used to count the frequency of each word in the reviews.

a) Merge the dataset into one. And divide the dataset as train, development and test.

Let us start by loading the data.

```
filename = "rt_reviews.csv"
df = pd.read_csv("rt_reviews.csv", encoding='ISO-8859-1')
df.head()
```

The dataset is loaded using the **read_csv** function from Pandas, it reads a csv file with reviews and freshness labels and stores them in a pandas DataFrame and the first five rows of the dataset are displayed using the **head()** method to verify that the dataset was loaded correctly.

If we want to take some part of the dataset then we can use `df=df.sample(num)` where num=the amount of data we want to consider.

```
df_shuffled = df.sample(frac=1, random_state=42).reset_index(drop=True)
```

The dataframe is shuffled using the `sample()` method with `frac=1` to shuffle the entire dataset and `random_state=42` to ensure that the same shuffling is applied each time the code is run. The `reset_index()` method is then used to reset the index of the dataframe.

This shuffling step is important in order to randomly partition the dataset into training and testing sets later on, which helps to prevent overfitting and evaluate the performance of the model on unseen data.

```
train_size = int(0.8 * len(df))
dev_size = int(0.1 * len(df))
test_size = len(df) - train_size - dev_size
# Splitting the datasets
train_df = df_shuffled[:train_size]
```

```
dev_df = df_shuffled[train_size : train_size + dev_size]
test_df = df_shuffled[train_size + dev_size :]
```

The dataset is then split into training, development, and test sets using the ratios 0.8:0.1:0.1, respectively. The **train_size**, **dev_size**, and **test_size** variables are calculated based on the length of the dataset, and then the **train_df**, **dev_df**, and **test_df** dataframes are created using the shuffled dataset and the calculated sizes.

b) Build a vocabulary as list and a reverse index as the key value might be handy.

```
word_counts = Counter()
for review in train_df['Review']:
    tokens = review.lower().split()
    word_counts.update(tokens)

# Building vocabulary and reverse indices
min_occurrences = 5
vocabulary = [word for word, count in word_counts.items() if count >=
min_occurrences]
reverse_index = {word: idx for idx, word in enumerate(vocabulary)}
```

The **Counter()** function from the collections package is used to count the frequency of each word in the reviews in the training set. The reviews are first converted to lowercase and split into individual words using the **lower()** and **split()** methods, respectively. The **update()** method is then used to add the counts of each word to the **word_counts** Counter object. The **vocabulary** list is then created by filtering the words that occur at least **min_occurrences** times in the training set. The **reverse_index** dictionary is created to map each word to its index in the **vocabulary**.

c) Calculate the probabilities: Probability of the occurrence and Conditional probability based on the sentiment.

```
# Tokenizing and creating counters
word_counts = Counter()
sentiment_word_counts = {'fresh': Counter(), 'rotten': Counter()}
for _, row in train_df.iterrows():
    sentiment, review = row['Freshness'], row['Review']
    tokens = set(review.lower().split())
    word_counts.update(tokens)
    sentiment_word_counts[sentiment].update(tokens)

# Calculating the probabilities
num_documents = len(train_df)
num_fresh_documents = sum(train_df['Freshness'] == 'fresh')
```

```

num_rotten_documents = sum(train_df['Freshness'] == 'rotten')

p_the = word_counts['the'] / num_documents
p_the_given_fresh = sentiment_word_counts['fresh']['the'] /
num_fresh_documents
p_the_given_rotten = sentiment_word_counts['rotten']['the'] /
num_rotten_documents

print(f"P(the) = {p_the}")
print(f"P(the|fresh) = {p_the_given_fresh}")
print(f"P(the|rotten) = {p_the_given_rotten}")

```

In the code above, the training data is tokenized and two counters are created - one for the total number of occurrences of each word and one for the number of occurrences of each word in either the "fresh" or "rotten" sentiment class. The probabilities are then calculated using these counters. First, the total number of documents, as well as the number of documents in each sentiment class, are counted. Then, the probability of the word "the" occurring overall ($P(\text{the})$) is calculated as the total number of occurrences of "the" divided by the total number of documents. The probability of "the" occurring given a fresh sentiment ($P(\text{the}|\text{fresh})$) is calculated as the number of times "the" occurs in fresh reviews divided by the total number of fresh reviews. The probability of "the" occurring given a rotten sentiment ($P(\text{the}|\text{rotten})$) is calculated similarly for the rotten sentiment class.

```

def calculate_conditional_probabilities(vocabulary, sentiment_word_counts,
num_fresh_documents, num_rotten_documents, smoothing=1):
    fresh_probs = {}
    rotten_probs = {}
    vocab_size = len(vocabulary)

    for word in vocabulary:
        fresh_count = sentiment_word_counts['fresh'][word]
        rotten_count = sentiment_word_counts['rotten'][word]

        fresh_prob = (fresh_count + smoothing) / (num_fresh_documents +
vocab_size * smoothing)
        rotten_prob = (rotten_count + smoothing) / (num_rotten_documents +
vocab_size * smoothing)

        fresh_probs[word] = fresh_prob
        rotten_probs[word] = rotten_prob

```

```

    return fresh_probs, rotten_probs

def naive_bayes_classifier(review, vocabulary, fresh_probs, rotten_probs):
    tokens = set(review.lower().split())
    tokens = [token for token in tokens if token in vocabulary]

    fresh_prob = np.prod([fresh_probs[token] for token in tokens])
    rotten_prob = np.prod([rotten_probs[token] for token in tokens])

    return 'fresh' if fresh_prob > rotten_prob else 'rotten'

```

These are two additional functions added to the main code to implement the Naive Bayes Classifier.

calculate_conditional_probabilities takes the vocabulary, the sentiment_word_counts (as computed earlier), the number of fresh documents, the number of rotten documents, and a smoothing value (defaulted to 1) as inputs. It returns two dictionaries, one for fresh probabilities and one for rotten probabilities, where each dictionary contains the probabilities for each word in the vocabulary given a fresh or a rotten sentiment. The probabilities are calculated using the Laplace smoothing technique.

naive_bayes_classifier takes a review, the vocabulary, the fresh_probs, and the rotten_probs dictionaries as inputs. It tokenizes the review, removes the tokens not in the vocabulary, and computes the probability of the review being fresh or rotten using the Naive Bayes formula, which is the product of the conditional probabilities of each word in the review given a fresh or a rotten sentiment. It then returns the sentiment with the highest probability, either 'fresh' or 'rotten'

```

# Calculating the conditional probabilities
fresh_probs, rotten_probs = calculate_conditional_probabilities(vocabulary,
    sentiment_word_counts, num_fresh_documents, num_rotten_documents, smoothing=0)

# Classifying reviews in the development dataset and calculate accuracy
correct_predictions = 0
for _, row in dev_df.iterrows():
    sentiment, review = row['Freshness'], row['Review']
    predicted_sentiment = naive_bayes_classifier(review, vocabulary, fresh_probs,
    rotten_probs)
    if predicted_sentiment == sentiment:
        correct_predictions += 1

accuracy = correct_predictions / len(dev_df)
print(f"Accuracy: {accuracy}")

```

After calculating the probabilities of the words given the two classes (fresh and rotten), the next step is to classify the reviews in the development dataset using the Naive Bayes Classifier. For each review in the development dataset, the function **naive_bayes_classifier** is called, which takes the review, vocabulary, fresh_probs, and rotten_probs as inputs and returns the predicted sentiment of the review.

If the predicted sentiment is the same as the actual sentiment of the review, the count of correct predictions is increased by one. After all the reviews have been classified, the accuracy of the classifier is calculated by dividing the count of correct predictions by the total number of reviews in the development dataset.

The accuracy of the classifier is a measure of how well the classifier is performing on the development dataset, which can be used to tune the model and improve its performance. We are getting around 80.3 % as our accuracy on the development dataset.

e) Experiments

i) Compare the effects of smoothing:

```
# Calculate the conditional probabilities with and without Laplace smoothing
fresh_probs_smoothed, rotten_probs_smoothed =
calculate_conditional_probabilities(vocabulary, sentiment_word_counts,
num_fresh_documents, num_rotten_documents, smoothing=1)
fresh_probs_unsmoothed, rotten_probs_unsmoothed =
calculate_conditional_probabilities(vocabulary, sentiment_word_counts,
num_fresh_documents, num_rotten_documents, smoothing=0)

# Classify reviews in the development dataset and calculate accuracy for both
cases
correct_predictions_smoothed = 0
correct_predictions_unsmoothed = 0

for _, row in dev_df.iterrows():
    sentiment, review = row['Freshness'], row['Review']

    predicted_sentiment_smoothed = naive_bayes_classifier(review, vocabulary,
fresh_probs_smoothed, rotten_probs_smoothed)
    if predicted_sentiment_smoothed == sentiment:
        correct_predictions_smoothed += 1

    predicted_sentiment_unsmoothed = naive_bayes_classifier(review, vocabulary,
fresh_probs_unsmoothed, rotten_probs_unsmoothed)
    if predicted_sentiment_unsmoothed == sentiment:
        correct_predictions_unsmoothed += 1

accuracy_smoothed = correct_predictions_smoothed / len(dev_df)
accuracy_unsmoothed = correct_predictions_unsmoothed / len(dev_df)
```

```
print(f"Accuracy with Laplace smoothing: {accuracy_smoothed}")
print(f"Accuracy without smoothing: {accuracy_unsmoothed}")
```

```
... Accuracy with Laplace smoothing: 0.8052291666666667
    Accuracy without smoothing: 0.8030208333333333
```

First, the code calls the **calculate_conditional_probabilities** function twice, once with a smoothing value of 1 and once with a smoothing value of 0. This calculates the conditional probabilities for each word in the vocabulary given the freshness label, with and without smoothing. Then, the code iterates over the development dataset and uses the **naive_bayes_classifier** function to classify each review, once using the smoothed probabilities and once using the unsmoothed probabilities. The number of correct predictions is incremented for each case. Finally, the accuracy for both cases is calculated and printed. The accuracy with Laplace smoothing is printed first, followed by the accuracy without smoothing. This allows for a comparison of the two different methods of calculating conditional probabilities and their impact on the accuracy of the Naive Bayes classifier. So here, accuracy with Laplace smoothing is 80.5% and without smoothing is 80.3% i.e with laplace smoothing the accuracy of NBC is increasing.

ii) Comparing and calculating best alpha values:

```
def grid_search_alpha(vocabulary, sentiment_word_counts, num_fresh_documents,
num_rotten_documents, dev_df, alphas):
    best_alpha = None
    best_accuracy = 0

    for alpha in alphas:
        # For each alpha calculating the conditional probability
        fresh_probs_alpha, rotten_probs_alpha =
        calculate_conditional_probabilities(vocabulary, sentiment_word_counts,
num_fresh_documents, num_rotten_documents, smoothing=alpha)

        # Count to store correct predictions
        correct_predictions_alpha = 0
        # Making prediction
        for _, row in dev_df.iterrows():
            sentiment, review = row['Freshness'], row['Review']
            predicted_sentiment_alpha = naive_bayes_classifier(review,
vocabulary, fresh_probs_alpha, rotten_probs_alpha)
            if predicted_sentiment_alpha == sentiment:
                correct_predictions_alpha += 1

        # Calculating the accuracy
        accuracy_alpha = correct_predictions_alpha / len(dev_df)
```

```

    # Updating best alpha, based on the accuracy
    if accuracy_alpha > best_accuracy:
        best_alpha = alpha
        best_accuracy = accuracy_alpha

    return best_alpha, best_accuracy

# Defining different alpha values to experiment with
alphas = [0, 0.1, 0.5, 1, 2, 5, 10]

# Performing analysis
best_alpha, best_accuracy = grid_search_alpha(vocabulary, sentiment_word_counts,
num_fresh_documents, num_rotten_documents, dev_df, alphas)

print(f"Best alpha value: {best_alpha}")
print(f"Best accuracy: {best_accuracy}")

```

```

... Best alpha value: 0.5
    Best accuracy: 0.8056666666666666

```

This code defines a function **grid_search_alpha** that performs a grid search for the best value of the Laplace smoothing parameter **alpha** for the Naive Bayes classifier. The function takes in the vocabulary, sentiment word counts, number of fresh documents, number of rotten documents, development dataset, and a list of **alpha** values to experiment with.

For each **alpha** value in the list, the function calculates the conditional probability of each word given the sentiment, using the **calculate_conditional_probabilities** function. It then classifies each review in the development dataset using the **naive_bayes_classifier** function with the **fresh_probs_alpha** and **rotten_probs_alpha** values calculated using the current **alpha** value. The function counts the number of correct predictions and calculates the accuracy for each **alpha** value.

The function then updates the **best_alpha** and **best_accuracy** variables based on the accuracy of each **alpha** value. The function returns the **best_alpha** and **best_accuracy** values.

Finally, the code defines a list of **alpha** values to experiment with, calls the **grid_search_alpha** function with these values, and prints out the best **alpha** value and corresponding best accuracy.

iii) Deriving top 10 words that predicts each class: $P[\text{class}|\text{word}]$

```

def calculate_class_given_word_probabilities(vocabulary, fresh_probs,
rotten_probs, num_fresh_documents, num_rotten_documents):
    fresh_given_word_probs = {}
    rotten_given_word_probs = {}

    for word in vocabulary:
        fresh_prob = fresh_probs[word]

```

```

rotten_prob = rotten_probs[word]

prob_word = fresh_prob * num_fresh_documents + rotten_prob *
num_rotten_documents

fresh_given_word_prob = (fresh_prob * num_fresh_documents) / prob_word
rotten_given_word_prob = (rotten_prob * num_rotten_documents) / prob_word

fresh_given_word_probs[word] = fresh_given_word_prob
rotten_given_word_probs[word] = rotten_given_word_prob

return fresh_given_word_probs, rotten_given_word_probs

# Calculating P[class | word] for each word
fresh_given_word_probs, rotten_given_word_probs =
calculate_class_given_word_probabilities(vocabulary, fresh_probs_smoothed,
rotten_probs_smoothed, num_fresh_documents, num_rotten_documents)

# Sorting the words by P[class | word] and get the top 10 words for each class
top_fresh_words = sorted(fresh_given_word_probs, key=fresh_given_word_probs.get,
reverse=True)[:10]
top_rotten_words = sorted(rotten_given_word_probs,
key=rotten_given_word_probs.get, reverse=True)[:10]

print("Top 10 words that predict 'fresh':")
print(top_fresh_words)

print("\nTop 10 words that predict 'rotten':")
print(top_rotten_words)

```

```

... Top 10 words that predict 'fresh':
['heartbreakingly', 'razor-sharp', 'beautifully.', 'flawless,', 'gem.', 'skilful', 'joyous,', 'cannily', 'rewarded.', 'petzold']

Top 10 words that predict 'rotten':
['charmless', 'unfunny', 'limp,', 'mirthless', 'lifeless.', 'flavorless', 'tediously', 'charmless', 'yawn.', 'uninteresting,']
+ Code + Markdown

```

This code defines a function **calculate_class_given_word_probabilities** that takes in the vocabulary, the conditional probabilities of each word given the freshness and rottenness classes (**fresh_probs** and **rotten_probs**), and the number of fresh and rotten documents, and calculates the probability of each class given each word (**P[class | word]**).

For each word in the vocabulary, the function calculates the probability of the word (**prob_word**) as the weighted

sum of the fresh and rotten probabilities of the word, where the weight for each probability is the number of documents of that class. The function then calculates the probability of the fresh class given the word (**fresh_given_word_prob**) as the product of the fresh probability of the word and the number of fresh documents, divided by the probability of the word, and similarly for the rotten class. The function returns dictionaries of the fresh and rotten probabilities given each word.

The code then calls this function with the smoothed conditional probabilities and the number of fresh and rotten documents, and obtains the top 10 words that predict "fresh" and "rotten" by sorting the words by their respective probabilities using the **sorted** function and the **get** method to get the probability of each word. Finally, it prints out the top 10 words that predict "fresh" and "rotten" using the **print** function.

```
# Calculating the conditional probabilities using the best alpha value
fresh_probs_best_alpha, rotten_probs_best_alpha =
calculate_conditional_probabilities(vocabulary, sentiment_word_counts,
num_fresh_documents, num_rotten_documents, smoothing=best_alpha)

# to store correct predictions
correct_predictions_test = 0
for _, row in test_df.iterrows():
    sentiment, review = row['Freshness'], row['Review']
    predicted_sentiment_test = naive_bayes_classifier(review, vocabulary,
fresh_probs_best_alpha, rotten_probs_best_alpha)
    if predicted_sentiment_test == sentiment:
        correct_predictions_test += 1

accuracy_test = correct_predictions_test / len(test_df)
print(f"Final accuracy: {accuracy_test}")
```

Final accuracy: 0.8032708333333334

This code calculates the final accuracy of the Naive Bayes classifier on the test dataset using the best alpha value found from the development dataset.

First, the code calculates the conditional probabilities of each word given the sentiment using the best alpha value. Then, the code initializes a counter to store the number of correct predictions and iterates over each review in the test dataset. For each review, the code uses the Naive Bayes classifier function to predict the sentiment and checks whether the predicted sentiment matches the true sentiment. If it does, the code increments the counter. After iterating over all the reviews, the code calculates the accuracy by dividing the number of correct predictions by the total number of reviews in the test dataset. Finally, prints the accuracy which is 80.3%.

My Contributions:

1. By following the referenced articles , I have understood the naive bayes classifier in practical.
2. Faced many challenges as we are implementing NBC without any modules but after going through few articles and understanding them helped me solve them.

3. From article[8] , got knowledge of smoothing and Laplace smoothing and how to implement them.
4. Also experimented with different alpha values such as $\alpha=[0,0.1,0.5,1,2,5,10]$ and got best alpha as 0.5 with accuracy of 80.3 %
5. Understood how NBC is used to classifies the data in practical scenario data.
6. Helped in understanding conditional probability concepts more perfectly.

Challenges:

1. During the testing and developing it was really hard to wait 20-30 mins for the model to be trained on 480,000 samples, so during the testing of model I used a sub sample of 24,000 values to test my code.
2. One of the issue was creating the inverted indices and counting the frequency of the words, after tokening [5] this article showed me how to use the counter function from collections library, and also provided a nice implementation of Naïve Bayes.
3. Another issue was to find the correct formula to calculate the probabilities,[6] this article helped me to calculate the posterior and prior, it also showed me how to use them in my naïve bayes implementation.
4. I was thinking about using a more complex tokenizing approach but it was quite hard then expected so, I decided to use simple split function along with lowercase.

References:

1. <https://www.techopedia.com/definition/32335/naive-bayes>
2. <https://note.nkmk.me/en/python-pandas-random-sort-shuffle/>
3. <https://www.cs.rhodes.edu/~kirlinp/courses/ai/f18/projects/proj3/naive-bayes-log-probs.pdf>
4. http://aritter.github.io/courses/5525_slides/probability_nb.pdf
5. <https://bsantraigi.github.io/2019/07/13/text-classification-using-naive-bayes-method.html>
6. <https://machinelearningmastery.com/classification-as-conditional-probability-and-the-naive-bayes-algorithm/>
7. <https://www.analyticsvidhya.com/blog/2022/03/building-naive-bayes-classifier-from-scratch-to-perform-sentiment-analysis/>
8. <https://towardsdatascience.com/introduction-to-na%C3%AFve-bayes-classifier-fa59e3e24aaf>
9. <https://akshithasingareddy.wixsite.com/2022/post/text-classification>

Github:

<https://github.com/saicharan1922/Naive-Bayes-Classifier-NBC-.git>