Document Classification with Naive Bayes ¶

Introduction

In this lesson, you'll investigate another implementation of the Bayesian framework in order to classify YouTube videos into the appropriate topic. The dataset you'll be investigating again comes from Kaggle. For further information, you can check out the original dataset here: https://www.kaggle.com/extralime/math-lectures (https://www.kaggle.com/extralime/math-lectures) .

Objectives

You will be able to:

- Implement document classification using Naive Bayes
- Explain how to code a bag of words representation
- Explain why it is necessary to use Laplacian smoothing correction

Bayes theorem for document classification

A common example of using Bayes' theorem to classify documents is a spam filtering algorithm. You'll be exploring this application in the upcoming lab. To do this, you examine the question "given this word (in the document) what is the probability that it is spam versus not spam?" For example, perhaps you get a lot of "special offer" spam. In that case, the words "special" and "offer" may increase the probability that a given message is spam.

Recall Bayes theorem:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

Applied to a document, one common implementation of Bayes' theorem is to use a bag of words representation. A bag of words representation takes a text document and converts it into a word frequency representation. For example, in a bag of words representation, the message:

> "Thomas Bayes was born in the early 1700s, although his exact date of birth is unknown. As a Presbyterian in England, he took an unconventional approach to education for his day since Oxford and Cambridge were tied to the Church of England."

Would look like this:

Processing math: 100%

In [1]:
```
doc = "Thomas Bayes was born in the early 1700s, although his exact date of birth is unknown. As a Presbyterian in England, he tc
bag = {}
for word in doc.split():
    # Get the previous entry, or 0 if not yet documented; add 1
    bag[word] = bag.get(word, 0) + 1
bag
```

Out[1]:
```
{'Thomas': 1,
 'Bayes': 1,
 'was': 1,
 'born': 1,
 'in': 2,
 'the': 2,
 'early': 1,
 '1700s,': 1,
 'although': 1,
 'his': 2,
 'exact': 1,
 'date': 1,
 'of': 2,
 'birth': 1,
 'is': 1,
 'unknown.': 1,
 'As': 1,
 'a': 1,
 'Presbyterian': 1,
 'England,': 1,
 'he': 1,
 'took': 1,
 'an': 1,
 'unconventional': 1,
 'approach': 1,
 'to': 2,
 'education': 1,
 'for': 1,
 'day': 1,
 'since': 1,
 'Oxford': 1,
 'and': 1,
 'Cambridge': 1,
 'were': 1,
 'tied': 1,
 'Church': 1,
 'England.': 1}
```

Additional preprocessing techniques can also be applied to the document before applying a bag of words representation, many of which you'll explore later when further investigating Natural Language Processing (NLP) techniques.

Once you've converted the document into a bag of words representation, you can then implement Bayes' theorem. Returning to the case of 'Spam' and 'Not Spam', you would have:

$$P(\text{Spam} \mid \text{Word}) = \frac{P(\text{Word} \mid \text{Spam})P(\text{Spam})}{P(\text{Word})}$$

Using the bag of words representation, you can then define $P(\text{Word} \mid \text{Spam})$ as

$$P(\text{Word} \mid \text{Spam}) = \frac{\text{Word Frequency in Document}}{\text{Word Frequency Across All Spam Documents}}$$

However, this formulation has a problem: what if you encounter a word in the test set that was not present in the training set? This new word would have a frequency of zero! This would commit two grave sins. First, there would be a division by zero error. Secondly, the numerator would also be zero; if you were to simply modify the denominator, having a term with zero probability would cause the probability for the entire document to also be zero when you subsequently multiplied the conditional probabilities in Multinomial Bayes. To effectively counteract these issues, Laplacian smoothing is often used giving:

$$P(\text{Word} \mid \text{Spam}) = \frac{\text{Word Frequency in Document} + 1}{\text{Word Frequency Across All Spam Documents} + \text{Number of Words in Corpus Vocabulary}}$$

Now, to implement this in Python!

Processing math: 100%

Load the dataset

In [2]:
```
import pandas as pd
df = pd.read_csv('raw_text.csv')
df.head()
```

Out[2]:

	text	label
0	The following content is\nprovided under a Cre...	Calculus
1	In this sequence of segments,\nwe review some ...	Probability
2	The following content is\nprovided under a Cre...	CS
3	The following\ncontent is provided under a Cre...	Algorithms
4	The following\ncontent is provided under a Cre...	Algorithms

In [3]:
```
df['label'].value_counts()
```

Out[3]:
```
Linear Algebra     152
Probability        124
CS                 104
Diff. Eq.           93
Algorithms          81
Statistics          79
Calculus            70
Data Structures     62
AI                  48
Math for Eng.       28
NLP                 19
Name: label, dtype: int64
```

Simple two-class case

To simplify the problem, you can start by subsetting to two specific classes:

In [4]:
```
df2 = df[df['label'].isin(['Algorithms', 'Statistics'])]
df2['label'].value_counts()
```

Out[4]:
```
Algorithms    81
Statistics    79
Name: label, dtype: int64
```

In [5]:
```
p_classes = dict(df2['label'].value_counts(normalize=True))
p_classes
```

Out[5]:
```
{'Algorithms': 0.50625, 'Statistics': 0.49375}
```

In [6]:
```
df2.iloc[0]
```

Out[6]:
```
text     The following\ncontent is provided under a Cre...
label                                           Algorithms
Name: 3, dtype: object
```

Train-test split

In [7]:
```
from sklearn.model_selection import train_test_split
X = df2['text']
y = df2['label']
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=17)
train_df = pd.concat([X_train, y_train], axis=1)
test_df = pd.concat([X_test, y_test], axis=1)
```

Processing math: 100%

Create the word frequency dictionary for each class

In [8]:
```
# Will be a nested dictionary of class_i : {word1:freq, word2:freq..., wordn:freq},.... class_m : {}
class_word_freq = {}
classes = train_df['label'].unique()
for class_ in classes:
    temp_df = train_df[train_df.label == class_]
    bag = {}
    for row in temp_df.index:
        doc = temp_df['text'][row]
        for word in doc.split():
            bag[word] = bag.get(word, 0) + 1
    class_word_freq[class_] = bag
```

Count the total corpus words

In [9]:
```
vocabulary = set()
for text in train_df['text']:
    for word in text.split():
        vocabulary.add(word)
V = len(vocabulary)
V
```

Out[9]:
```
23977
```

Create a bag of words function

In [10]:
```
def bag_it(doc):
    bag = {}
    for word in doc.split():
        bag[word] = bag.get(word, 0) + 1
    return bag
```

Implement Naive Bayes

In [11]:
```
import numpy as np
def classify_doc(doc, class_word_freq, p_classes, V, return_posteriors=False):
    bag = bag_it(doc)
    classes = []
    posteriors = []
    for class_ in class_word_freq.keys():
        p = p_classes[class_]
        for word in bag.keys():
            num = bag[word]+1
            denom = class_word_freq[class_].get(word, 0) + V
            p *= (num/denom)
        classes.append(class_)
        posteriors.append(p)
    if return_posteriors:
        print(posteriors)
    return classes[np.argmax(posteriors)]
```

In [12]:
```
classify_doc(train_df.iloc[0]['text'], class_word_freq, p_classes, V, return_posteriors=True)
```

```
[0.0, 0.0]
```

Out[12]:
```
'Algorithms'
```

Avoid underflow

As you can see from the output above, repeatedly multiplying small probabilities can lead to underflow; rounding to zero due to numerical approximation limitations. As such, a common alternative is to add the logarithms of the probabilities as opposed to multiplying the raw probabilities themselves. If this is alien to you, it might be worth reviewing some algebra rules of exponents and logarithms briefly:

$e^x \cdot e^y = e^{x+y}$

$log_e(e) = 1$

$e^{log(x)} = x$

With that, here's an updated version of the function using log probabilities to avoid underflow:

Processing math: 100%

In [13]:
```
def classify_doc(doc, class_word_freq, p_classes, V, return_posteriors=False):
    bag = bag_it(doc)
    classes = []
    posteriors = []
    for class_ in class_word_freq.keys():
        p = np.log(p_classes[class_])
        for word in bag.keys():
            num = bag[word]+1
            denom = class_word_freq[class_].get(word, 0) + V
            p += np.log(num/denom)
        classes.append(class_)
        posteriors.append(p)
    if return_posteriors:
        print(posteriors)
    return classes[np.argmax(posteriors)]
```

In [14]:
```
classify_doc(train_df.iloc[0]['text'], class_word_freq, p_classes, V, return_posteriors=True)
```

```
[-5578.267536771343, -5577.213285866603]
```

Out[14]:
```
'Statistics'
```

In [15]:
```
classify_doc(train_df.iloc[10]['text'], class_word_freq, p_classes, V, return_posteriors=True)
```

```
[-2572.1544445158343, -2571.311308656896]
```

Out[15]:
```
'Statistics'
```

In [16]:
```
classify_doc(train_df.iloc[12]['text'], class_word_freq, p_classes, V, return_posteriors=True)
```

```
[-4602.602622507951, -4601.755644621728]
```

Out[16]:
```
'Statistics'
```

In [17]:
```
y_hat_train = X_train.map(lambda x: classify_doc(x, class_word_freq, p_classes, V))
residuals = y_train == y_hat_train
residuals.value_counts(normalize=True)
```

Out[17]:
```
False    0.508333
True     0.491667
dtype: float64
```

As you can see, this algorithm leaves a lot to be desired. A measly 49% accuracy is nothing to write home about. (In fact, it's slightly worse than random guessing!) In practice, substantial additional preprocessing including removing stop words and using stemming or lemmatisation would be required. Even then, Naive Bayes might still not be the optimal algorithm. Nonetheless, it is a worthwhile exercise and a comprehendible algorithm.

Summary

In this lesson, you got to see another application of Bayes' theorem as a means to do some rough documentation classification.

Processing math: 100%