**Probability and Classification + Bayes Optimal Decisions**

* **Supervised Learning Algorithm**

Textual data dominates our world from the tweets you read to the timeless writings of Seneca. And while we’re consuming images (looking at you Instagram) and videos at increasing rates, you still read Google search results multiple times per daily.

One frequently recurring problem with text data is [Sentiment analysis](https://en.wikipedia.org/wiki/Sentiment_analysis) (classification). Imagine you’re a big sugar + water beverage company. You want to know what people think of your products. You’ll search for texts with some tags, logos or names. You can then use Sentiment analysis to figure out if the opinions are positive or negative. Of course, you’ll send the negative ones to your highly underpaid support center in India to sort things out.

Here, we’ll build a generic text classifier that puts movie review texts into one of two categories - negative or positive sentiment. We’re going to have a brief look at the Bayes theorem and relax its requirements using the Naive assumption.

**Dealing with Text**

Computers don’t understand text data, though they do well with numbers. [Natural Language Processing (NLP)](https://en.wikipedia.org/wiki/Natural_language_processing) offers a set of approaches to solve text-related problems and represent text as numbers. While NLP is a vast field, we’ll use some simple preprocessing techniques and [Bag of Words](https://en.wikipedia.org/wiki/Bag-of-words_model) model.

**The Data**

Our data comes from a Kaggle challenge - [“Bag of Words Meets Bags of Popcorn”](https://www.kaggle.com/c/word2vec-nlp-tutorial).

We have 25,000 movie reviews from IMDB labeled as positive or negative. You might know that IMDB ratings are in the 0-10 range. An additional preprocessing step, done by the dataset authors, converts the rating to binary sentiment (<5 - negative ). Of course, a single movie can have multiple reviews, but no more than 30.

**Reading the reviews**

Let’s load the training and test data in Pandas data frames:

train = pd.read\_csv("imdb\_review\_train.tsv", delimiter="\t")

test = pd.read\_csv("imdb\_review\_test.tsv", delimiter="\t")

**Exploration**

Let’s get a feel for our data. Here are the first 5 rows of the training data:

| **id** | **sentiment** | **review** |
| --- | --- | --- |
| 5814\_8 | 1 | With all this stuff going down at the moment w… |
| 2381\_9 | 1 | \The Classic War of the Worlds” by Timothy Hi… |
| 7759\_3 | 0 | The film starts with a manager (Nicholas Bell)… |
| 3630\_4 | 0 | It must be assumed that those who praised this… |
| 9495\_8 | 1 | Superbly trashy and wondrously unpretentious 8… |

We’re going to focus on the sentiment and review columns. The id column is combining the movie id with a unique number of a review. This might be a piece of important information in real-world scenarios, but we’re going to keep it simple.



Both positive and negative sentiments have an equal presence. No need for additional gimmicks to fix that!

**Cleaning**

Real-world text data is really messy. It can contain excessive punctuation, HTML tags (including that br), multiple spaces, etc. We’ll try to remove/normalize most of it.

Most of the cleaning we’ll do using [Regular Expressions](https://en.wikipedia.org/wiki/Regular_expression), but we’ll use two libraries to handle

HTML tags (surprisingly hard to remove) and removing common (stop) words:

def clean(self, text):

no\_html = BeautifulSoup(text).get\_text()

clean = re.sub("[^a-z\s]+", " ", no\_html, flags=re.IGNORECASE)

return re.sub("(\s+)", " ", clean)

First, we use [BeautifulSoup](https://www.crummy.com/software/BeautifulSoup/bs4/doc/" \t "_blank) to remove HTML tags from our text. Second, we remove anything that is not a letter or space (note the ignoring of uppercase characters). Finally, we replace excessive spacing with a single one.

**Tokenization**

Now that our reviews are “clean”, we can further prepare them for our Bag of Words model. Let’s them to lowercase letters and split them into individual words. This process is known as [tokenization](https://en.wikipedia.org/wiki/Lexical_analysis#Tokenization):

def tokenize(self, text):

clean = self.clean(text).lower()

stopwords\_en = stopwords.words("english")

return [w for w in re.split("\W+", clean) if not w in stopwords\_en]

The last step of our pre-processing is to remove [stop words](https://en.wikipedia.org/wiki/Stop_words) using those defined in the [NLTK](https://www.nltk.org/) library. Stop words are usually frequently occurring words that might not significantly affect the meaning of the text. Some examples in English are: “is”, “the”, “and”.

An additional benefit of removing stop words is speeding up our models since we’re removing the amount of train/test data. However, in real-world scenarios, you should think about whether removing stop words can be justified.

We’ll place our clean and tokenize function in a class called *Tokenizer*.

**Naive Bayes**

*Naive Bayes* models are probabilistic classifiers that use the [Bayes theorem](https://en.wikipedia.org/wiki/Bayes%27_theorem) and make a strong assumption that the features of the data are independent. For our case, this means that each word is independent of others.

Intuitively, this might sound like a dumb idea. You know that (even from reading) the prev word(s) influence the current and next ones. However, the assumption simplifies the math and [works really well in practice](https://www.cc.gatech.edu/~isbell/reading/papers/Rish.pdf)!

The Bayes theorem is defined as:

P(A|B) = \frac{P(B|A)P(A)}{P(B)}*P*(*A*∣*B*)=*P*(*B*)*P*(*B*∣*A*)*P*(*A*)​

where A*A* and B*B* are some events and P(.)*P*(.) is a probability.

This equation gives us the conditional probability of event A*A* occurring given B*B* has happened. In order to find this, we need to calculate the probability of B*B* happening given A*A* has happened and multiply that by the probability of A*A* (known as *Prior*) happening. All of this is divided by the probability of B*B* happening on its own.

The naive assumption allows us to reformulate the Bayes theorem for our example as:

P(Sentiment|w\_1,\dots,w\_n)=\frac{P(Sentiment) \displaystyle \prod\_{i=1}^{n}{P(w\_i|Sentiment)}}{P(w\_1,\dots,w\_n)}*P*(*Sentiment*∣*w*1​,…,*wn*​)=*P*(*w*1​,…,*wn*​)*P*(*Sentiment*)*i*=1∏*n*​*P*(*wi*​∣*Sentiment*)​

We don’t really care about probabilities. We only want to know whether a given text has a positive or negative sentiment. We can skip the denominator entirely since it just scales the numerator:

P(Sentiment|w\_1,\dots,w\_n) \propto P(Sentiment) \prod\_{i=1}^{n}{P(w\_i|Sentiment)}*P*(*Sentiment*∣*w*1​,…,*wn*​)∝*P*(*Sentiment*)*i*=1∏*n*​*P*(*wi*​∣*Sentiment*)

To choose the sentiment, we’ll compare the scores for each sentiment and pick the one with a higher score.

**Implementing Multinomial Naive Bayes**

As you might’ve guessed by now, we’re classifying text into one of two groups/categories - positive and negative sentiment.

Multinomial Naive Bayes allows us to represent the features of the model as frequencies of their occurrences (how often some word is present in our review). In other words, it tells us that the probability distributions we’re using are [multinomial](https://en.wikipedia.org/wiki/Multinomial_distribution).

**Note on numerical stability**

Our model relies on multiplying many probabilities. Those might become increasingly small and our computer might round them down to zero. To combat this, we’re going to use log probability by taking log of each side in our equation:

\log{P(Sentiment|w\_1,\dots,w\_n)} = \log{P(Sentiment)} + \log{\prod\_{i=1}^{n}{P(w\_i|Sentiment)}}log*P*(*Sentiment*∣*w*1​,…,*wn*​)=log*P*(*Sentiment*)+log*i*=1∏*n*​*P*(*wi*​∣*Sentiment*)

Note that we can still use the highest score of our model to predict the sentiment since log is [monotonic](https://en.wikipedia.org/wiki/Monotonic_function).

**Building our model**

Finally, time to implement our model in Python. Let’s start by defining some variables and group the data by class, so our training code is a bit tidier:

def fit(self, X, y):

self.n\_class\_items = {}

self.log\_class\_priors = {}

self.word\_counts = {}

self.vocab = set()

n = len(X)

grouped\_data = self.group\_by\_class(X, y)

We’re going to implement a generic text classifier that doesn’t assume the number of classes. You can use it for news category prediction, sentiment analysis, email spam detection, etc.

For each class, we’ll find the number of examples in it and the log probability (prior). We’ll also record the number of occurrences of each word and create a vocabulary - set of all words we’ve seen in the training data:

for c, data in grouped\_data.items():

self.n\_class\_items[c] = len(data)

self.log\_class\_priors[c] = math.log(self.n\_class\_items[c] / n)

self.word\_counts[c] = defaultdict(lambda: 0)

for text in data:

counts = Counter(self.tokenizer.tokenize(text))

for word, count in counts.items():

if word not in self.vocab:

self.vocab.add(word)

self.word\_counts[c][word] += count

Note that we use our *Tokenizer* and the built-in class [Counter](https://docs.python.org/3/library/collections.html#collections.Counter) to convert a review to a bag of words.

In case you’re interested, here’s how group\_by\_class works:

def group\_by\_class(self, X, y):

data = dict()

for c in self.classes:

data[c] = X[np.where(y == c)]

return data

**Making predictions**

In order to predict sentiment from text data, we’ll use our class priors and vocabulary:

def predict(self, X):

result = []

for text in X:

class\_scores = {c: self.log\_class\_priors[c] for c in self.classes}

words = set(self.tokenizer.tokenize(text))

for word in words:

if word not in self.vocab: continue

for c in self.classes:

log\_w\_given\_c = self.laplace\_smoothing(word, c)

class\_scores[c] += log\_w\_given\_c

result.append(max(class\_scores, key=class\_scores.get))

return result

For each review, we use the log priors for positive and negative sentiment and tokenize the text. For each word, we check if it is in the vocabulary and skip it if it is not. Then we calculate the log probability of this word for each class. We add the class scores and pick the class with a max score as our prediction.

**Laplace smoothing**

Note that log(0)*log*(0) is *undefined* (and no, we’re not using JavaScript here). It is entirely possible for a word in our vocabulary to be present in one class but not another - the probability of finding this word in that class will be 0! We can use [Laplace smoothing](https://en.wikipedia.org/wiki/Additive_smoothing) to fix this problem. We’ll simply add 1 to the numerator but also add the size of our vocabulary to the denominator:

def laplace\_smoothing(self, word, text\_class):

num = self.word\_counts[text\_class][word] + 1

denom = self.n\_class\_items[text\_class] + len(self.vocab)

return math.log(num / denom)

**Predicting sentiment**

Now that your model can be trained and make predictions, let’s use it to predict sentiment from movie reviews.

**Data preparation**

As discussed previously, we’ll use only the review and sentiment columns from the training data. Let split it for training and testing:

X = train['review'].values

y = train['sentiment'].values

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=RANDOM\_SEED)

**Evaluation**

We’ll pack our fit and predict functions into a class called MultinomialNavieBayes Let’s use it:

MNB = MultinomialNaiveBayes(

classes=np.unique(y),

tokenizer=Tokenizer()

).fit(X\_train, y\_train)

Our classifier takes a list of possible classes and a Tokenizer as parameters. Also, the API is quite nice (thanks scikit-learn!)

y\_hat = MNB.predict(X\_test)

accuracy\_score(y\_test, y\_hat)

0.8556

This looks nice. We got an accuracy of ~86% on the test set.

Here is the classification report:

precision recall f1-score support

0 0.89 0.81 0.85 2481

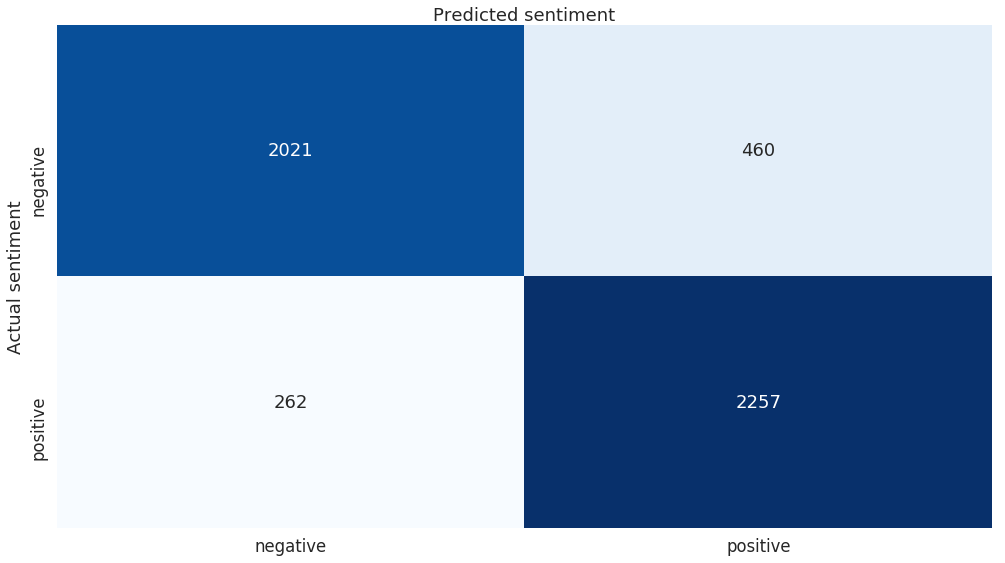
1 0.83 0.90 0.86 2519

accuracy 0.86 5000

macro avg 0.86 0.86 0.86 5000

weighted avg 0.86 0.86 0.86 5000

And the confusion matrix:



Overall, our classifier does pretty well for himself. Submit the predictions to Kaggle and find out what place you’ll get on the leaderboard.

**Conclusion**

Well done! You just built a Multinomial Naive Bayes classifier that does pretty well on sentiment prediction. You also learned about Bayes theorem, text processing, and Laplace smoothing!