

Module 18: Natural Language Processing (NLP)

Quick Reference Guide

Learning Outcomes:

1. Identify the similarities and differences between text and numerical data.
2. Convert text data to numerical data using NLTK.
3. Tokenize blocks of text.
4. Decide what features to extract from a text for a particular classification purpose.
5. Implement feature extraction using NLTK.
6. Use Bag-of-Words and TF-IDF algorithms to convert text data to numerical representation.
7. Classify text data using Bag-of-Words, TF-IDF, and logistic regression algorithms.
8. Compare the results of classification on text data using several models.
9. Apply the naïve Bayes classifier.

Introduction

Natural Language Processing (NLP) aims to make human language accessible to computers. So far in this course, you have worked with models whose inputs were numbers. They were either real valued measurements, such as the lengths and widths of iris flower petals. Or they were categories that could be easily encoded as numbers using, for example, the one-hot encoding scheme. Language is different.

Language as an input

First, the order of words in a sentence changes its meaning: For example, the meaning of "I had my car cleaned," is different from "I had cleaned my car" even though both sentences contain exactly the same words.

Second, a word's meaning depends on its position in the sentence. For example, the word bark in the sentence, "That dog loves to bark" has a different meaning than in, "The car damaged the bark of that tree." This does not happen with numbers. A two is always a two.

Third, human communication is often unclear. Consider, for example, that humans don't always say exactly what they mean. Moreover, communication styles tend to be highly individualized. Mistakes such as typos further complicate the process. As a result, NLP is one of the most challenging and interesting fields of machine learning.

NLP applications

NLP has many applications with major impacts on everyday life. These include:

- Search engines use NLPs to interpret questions
- NLP identifies spam emails
- Deep NLP has improved computer translation
- NLP is used for dialogue systems and chatbots
- NLP can produce summaries of lengthy texts
- NLP can categorize texts in different ways, which include spam filtering and sentiment analysis
- NLP systems can recognize human speech

Deep NLP

Just like many other areas of machine learning, NLP has been transformed by deep neural networks. And this has given rise to the new area of deep NLP. Deep NLP is an advanced topic that will not be covered in this course. However, the concepts here will help with further study.

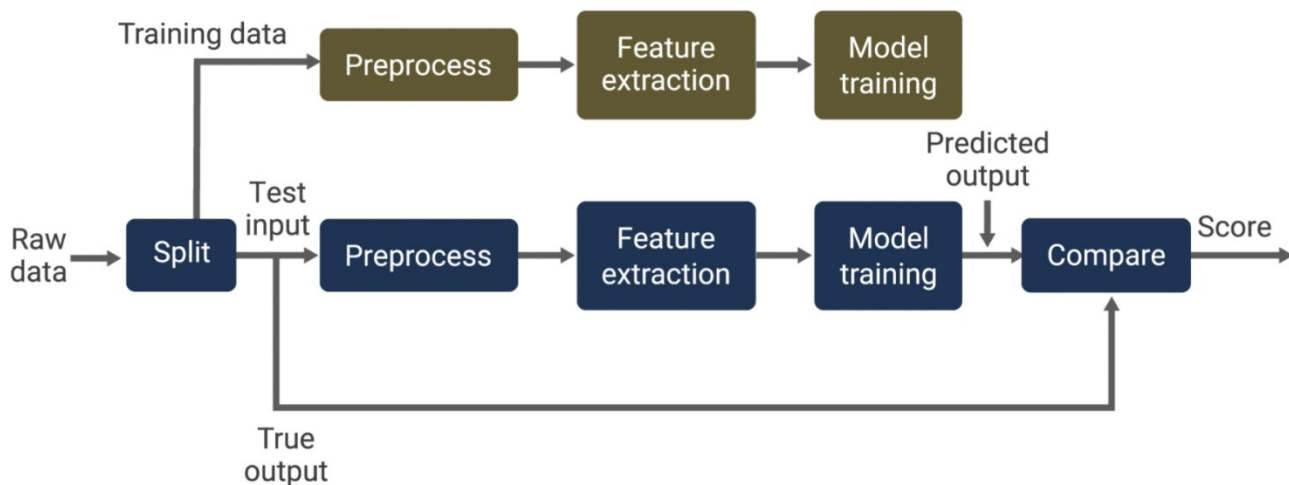
Workflow: The similarities between ML and NPL

In a number-based ML problem, you are given a raw dataset and typically the solution is structured as follows:

1. You set aside a **test set**, which is unused until the test phase
2. The remainder is the **training data**, which is used to build the model
3. **Preprocessing** corrects flaws in the data
4. You can also build **features** for the model
5. You use these inputs in a model training procedure, which minimizes the **loss function**

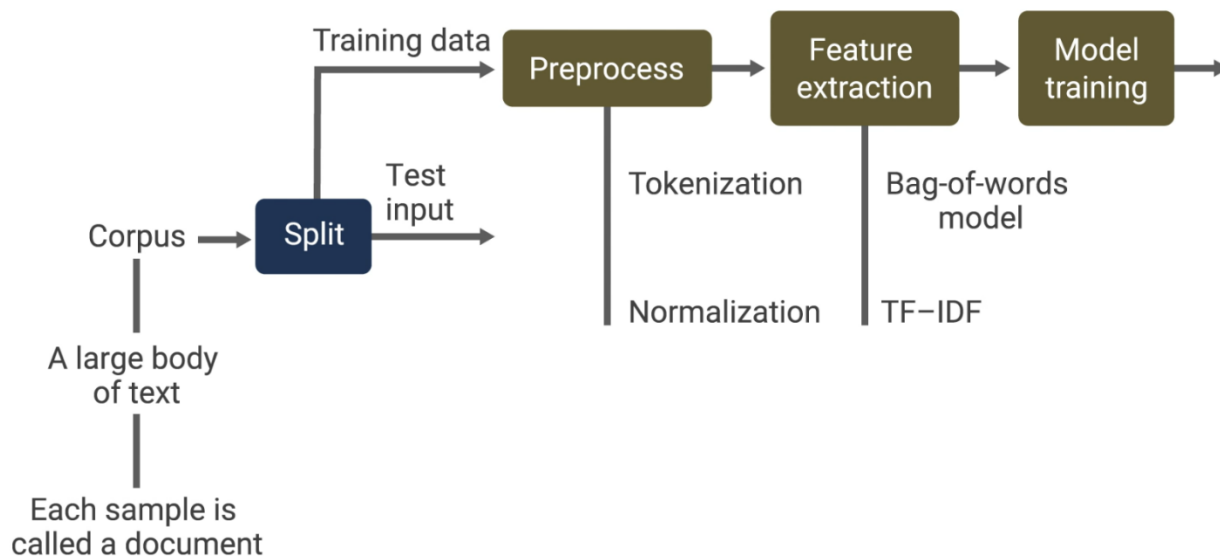
This procedure can be performed for different hyperparameters or model types. The performance of these competing designs can be compared using a separate validation dataset, or with cross-validation. Once the training process has been established, it can be applied to the test data.

ML WORKFLOW



The test input is fed through a pipeline to obtain the predicted outputs, which are compared to the true outputs to produce a final score for the model. Most of this workflow is identical in NLP, but there are a few important differences.

ML WORKFLOW FOR NLP



The raw dataset is usually called a corpus and comprises a large body of text, perhaps the IMDB database of movie reviews or all of Wikipedia, for example. Each sample in the corpus is called a document. A document can be a single review, or a Wikipedia article, or a tweet. Just as before, the corpus is split into training and testing datasets.

The main differences reside in the **preprocessing** and **feature extraction** steps. Feature extraction in NLP is about converting the text-based representation of the document into a numerical representation that is amenable to machine learning models. Approaches to feature extraction include the **bag-of-words model** and **TF-IDF**. The preprocessing steps include text **tokenization** and **text normalization**. Text tokenization splits the text into separate grammatical units, or tokens. Normalization reduces the tokens to a core set that captures the important information in the document. The rest of the workflow, including how you test and score the model, is similar to the standard machine learning problem. The most popular Python library for doing these NLP-specific preprocessing and feature extraction tasks is the **Natural Language toolkit**, or **NLTK**.

NLTK Token

Preprocessing steps are applied to text documents to prepare them for use in machine learning models. The Natural Language toolkit (NLTK) can be used to do this. NLTK is an open-source set of Python modules that:

- Provides datasets and algorithms used in NLP
- Is simple and accessible

<https://www.nltk.org/>

| | |
|---|---|
| <p>NLTK</p> <p>Search</p> <p>NLTK Documentation</p> <p>API Reference</p> <p>Example Usage</p> <p>Module Index</p> <p>Wiki</p> <p>FAQ</p> <p>Open Issues</p> <p>NLTK on GitHub</p> <p>Installation</p> <p>Installing NLTK</p> <p>Installing NLTK Data</p> | <p>Documentation</p> <p>Natural Language Toolkit</p> <p>NLTK is a leading platform for building Python programs to work with human language data. It provides easy-to-use interfaces to over 50 corpora and lexical resources such as WordNet, along with a suite of text processing libraries for classification, tokenization, stemming, tagging, parsing, and semantic reasoning, wrappers for industrial-strength NLP libraries, and an active discussion forum.</p> <p>Thanks to a hands-on guide introducing programming fundamentals alongside topics in computational linguistics, plus comprehensive API documentation, NLTK is suitable for linguists, engineers, students, educators, researchers, and industry users alike. NLTK is available for Windows, Mac OS X, and Linux. Best of all, NLTK is a free, open source, community-driven project.</p> <p>NLTK has been called “a wonderful tool for teaching, and working in, computational linguistics using Python,” and “an amazing library to play with natural language.”</p> <p>Natural Language Processing with Python provides a practical introduction to programming for language processing. Written by the creators of NLTK, it guides the reader through the fundamentals of writing Python</p> |
|---|---|

From the NLTK webpage, you can click on installing NLTK to reach the installation instructions. These provide installation steps depending on whether you are on a Mac/Unix machine or a Windows machine.

Since you already have Python and NumPy installed, all you will have to do is **pip install nltk**.

Or if you prefer, you can use Anaconda. With Anaconda installed on your machine, you can create an environment by entering this string into the command line: **conda create -n nlp -c conda-forge numpy pandas jupyter matplotlib scikit-learn nltk**. Activate your environment with **conda activate nlp**. To import NLTK into your Python work environment, use **import nltk**.

NLTK works with very large bodies of text. It offers a download manager that you can use to retrieve specific datasets. To launch the downloads manager, type **nltk.download()** into a Jupyter Notebook. This opens a window with all the datasets, or corpora, that nltk provides.



For example, to obtain the Twitter dataset, select `twitter_samples` from the list, and then click on Download.

Tokenization

In NLP, tokens are single units of text. They can be sentences or words. Consider, for example, the following review of the 2002 sci-fi action thriller, *Minority Report*.

"I enjoyed 'Minority Report.' Tom Cruise didn't disappoint, and Steven Spielberg is at the top of his game. The movie was long but it wasn't boring. GREAT MOVIE!"

To build a model with the goal of identifying whether or not the review is positive, start by splitting the text into individual words using the **word_tokenize()** method. Note that 'words' for this purpose include

punctuation marks, numbers, URLs, et cetera. In the movie review example, the sample text includes 36 so-called 'words.'

There are many different tokenizers to choose from, each with its own set of rules. Furthermore, tokenizers are language specific. Some languages are much harder to tokenize.

Once a document is tokenized, the next step is to combine and filter those tokens in ways that reveal identifiable structures in the text. This is called **normalization**.

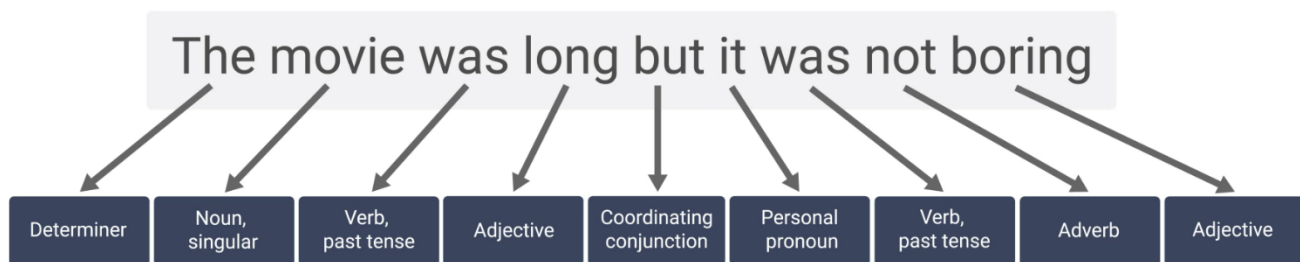
Normalization

Text normalization operations use tokens as inputs and generate tokens as outputs. This is different from **feature extraction**, which produces numbers from tokens. Normalization operations improve the quality of tokens.

Autocorrecting spelling errors can be difficult, since there may be multiple options. For example, the misspelled word *fask* may refer to *task*, *flask*, *ask*, *mask*, et cetera. To improve the performance of the model, you might **remove special characters** if they provide little information. You can also **remove stop words**, such as *am*, *is*, *are*, et cetera.

Parts of speech tagging

Identifying parts of speech can help in identifying the grammatical structure of sentences. This is used, for example, in speech recognition apps or in automatic writing assistance to identify the grammatical structure of sentences.



Each word in this sentence has a standard part of speech designation. NLTK's **pos_tag()** method can be used to obtain the parts of speech in a list of words.

```
words_pos = nltk.pos_tag(words)
words_pos
```

```
[('I', 'PRP'),
 ('enjoyed', 'VBP'),
 ('Minority', 'JJ'),
 ('Report', 'NNP'),
 ('"', 'POS'),
 (',', ','),
 ('Tom', 'NNP'),
 ('Cruise', 'NNP'),
 ('did', 'VBD'),
 ('n't', 'RB'),
 ('disappoint', 'VB'),
 (',', ','),
 ('and', 'CC'),
 ('Steven', 'NNP'),
 ('Spielberg', 'NNP'),
 ('is', 'VBZ'),
 ('at', 'IN'),
```

```
(('the', 'DT'),
 ('top', 'NN'),
 ('of', 'IN'),
 ('his', 'PRP$'),
 ('game', 'NN'),
 (',', ','),
 ('The', 'DT'),
 ('movie', 'NN'),
 ('was', 'VBD'),
 ('long', 'RB'),
 ('but', 'CC'),
 ('it', 'PRP'),
 ('was', 'VBD'),
 ('n't', 'RB'),
 ('boring', 'JJ'),
 (',', ','),
 ('GREAT', 'JJ'),
 ('MOVIE', 'NN'),
 ('!', '!')]
```

You can find the meaning of each of the parts of speech tags with the **upenn_tagset()** method.

Named entity recognition

Once the parts of speech have been identified, the next step is to identify named entities. These are noun phrases that denote particular kinds of things. Some of the most common named entity types are summarized in the table.

| Named entity type | An example |
|-------------------|--------------------|
| Organization | The United Nations |
| Person | Ringo Starr |
| Date | Fresno, California |
| Time | 4:42 p.m. |

You can use NLTK's **ne_chunk()** method to pick out named entities. These named entity tags can be used to, for example, remove the names of people from a data sample. You can also remove all the stop words. A list of English stop words can be loaded using the `stopwords` method from the `nltk.corpus` module. In this way, the text is distilled without losing the core message.

Stemming and lemmatization

Stemming and lemmatization replace groups of words with their root forms. For example, it is not useful to have four separate tokens for joy, joyful, joyfully, and joyous. Instead, you can replace all of these with the root form, joy. **Stemming** is a rule-based operation that produces root forms called stems by making a series of substring replacements. There are many stemming systems to choose from. **PorterStemmer** is the basic one

provided in NLTK. Stemmers can produce non-words. The model does not need to understand the meaning of the words.

Lemmatization is similar to stemming, but the outputs are always real words. This is a much more difficult task and it requires a significant knowledge of the language on the part of the lemmatizer. NLTK provides an interface to the **WordNet** lemmatizer. WordNet is a vast lexical database for the English language. To use it, you call the `lemmatize` method on the **WordNetLemmatizer** object. The lemmatizer does not necessarily reduce the number of tokens, but it does maintain grammatical coherence. Lemmatization can be useful in document summarization, where some measure of grammatical coherence should be preserved.

Stemmed

```
' '.join([stemmer.stem(w) for w in words_nonames_lower_nostop])
"enjoy 'minor report ' . n't disappoint , top game . movi long n't bore . great movi !"
```

Lemmatized

```
' '.join([lemma.lemmatize(w) for w in words_nonames_lower_nostop])
"enjoyed 'minority report ' . n't disappoint , top game . movie long n't boring . great movie !"
```

In this example, the stemmed version of the movie review is shorter and probably better for the task of sentiment analysis than the lemmatized one.

Bag-of-words and TF-IDF

Text processed using tokenization and normalization convert documents into tokens that capture the essence of the text. Next, you will translate those tokens into numbers. This step is called **feature extraction** or feature representation. **Bag-of-words** is the simplest method for feature extraction. The **TF-IDF** approach is more complex. **Full-word vectorization** is another,

very advanced method that is beyond the scope of this course. The approach you choose, will depend on the application, data availability, and model performance.

In the bag-of-words method, you create a feature for every distinct token in the data. For each document, you enter the number of occurrences of each token in that document. In this way you can process, for example, three movie reviews and boil them down to their essential tokens. You can create a **feature matrix** by entering the number of times that each feature appears in a document, in a separate feature column. Along with the output of positive and negative sentiment, this table constitutes your training dataset for a logistic regression model.

| | | features | | | | output |
|-----------|---------------------------------------|----------|--------------|--------|---------|-----------|
| documents | Tokens | 'enjoy' | 'disappoint' | 'bore' | 'great' | sentiment |
| | ['enjoy','disappoint','bore','great'] | 1 | 1 | 1 | 1 | 1 |
| | ['bore','disappoint','bore','bore'] | 0 | 1 | 3 | 0 | 0 |
| | ['great','great','enjoy'] | 1 | 0 | 0 | 2 | 1 |

One obvious drawback to this approach is that it discards the ordering of words. So, for example, the sentence, "I had my car cleaned" becomes indistinguishable from "I had cleaned my car." But that might not be a problem in cases where **you** are not interested in who did the cleaning. Another problem with the bag-of-words approach is that it does not track how informative words are. This is especially important when **you** are

dealing with large documents. The TF-IDF approach attempts to correct this by quantifying the usefulness of each token.

| Tokens | 'enjoy' | 'disappoint' | 'bore' | 'great' |
|---------------------------------------|---------|--------------|--------|---------|
| ['enjoy','disappoint','bore','great'] | 0.25 | 0.25 | 0.25 | 0.25 |
| ['bore','disappoint','bore','bore'] | 0 | 0.25 | 0.75 | 0 |
| ['great','great','enjoy'] | 0.33 | 0 | 0 | 0.66 |

Here, each entry in the feature table is not an integer count, but a real valued TF-IDF score. These scores are computed as:

$$\mathbf{tfidf}(t, d) = \mathbf{tf}(t, d) \times \mathbf{idf}(t)$$

The term frequency of a term, t , in a document, d , is computed as:

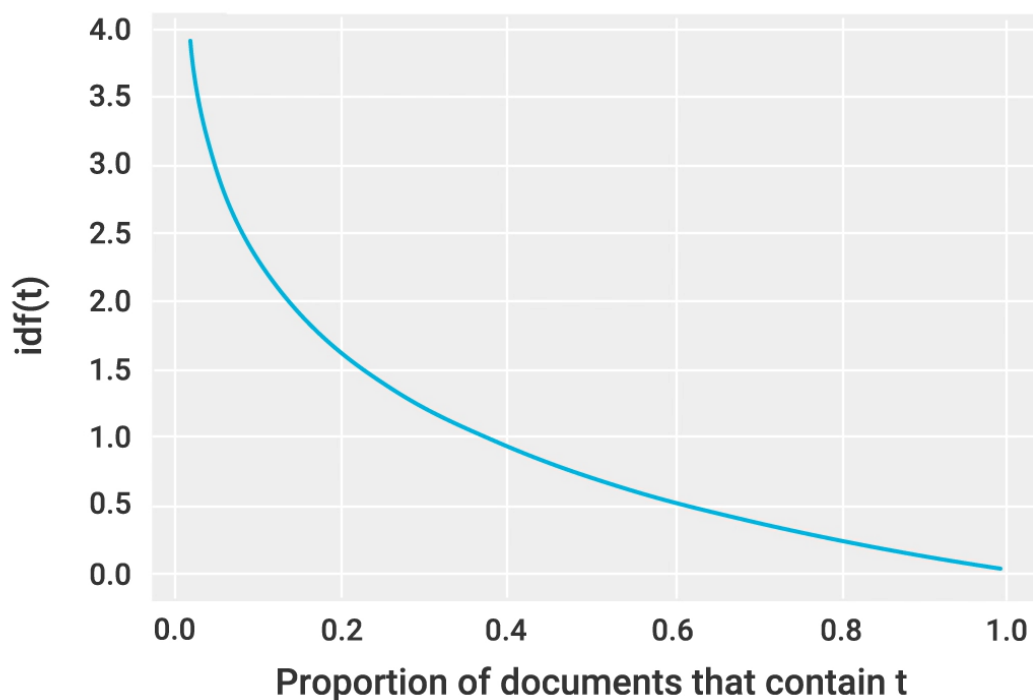
$$\mathbf{tf}(t, d) = \frac{\text{number of times that } t \text{ occurs in } d}{\text{number of words in } d}$$

The numerator here is the bag-of-words counts. The term frequency is a normalized version of bag-of-words. Just like bag-of-words, tf scores are meant to give larger weight to terms that appear often in the document.

The inverse document frequency of a term, t , quantifies the importance of that term as:

$$\mathbf{idf}(t) = -\log \frac{\text{number of documents that contain } t}{\text{total number of documents}}$$

Here is a plot of the idf as a function of the proportion of documents containing the term.



Notice that the idf is always positive. The idf amplifies very rare words whose document frequency approaches zero and it will completely annihilate words that are present in every document.

In the example, each of the terms appear in two of the three documents. Since they all have the same idf score of negative log of two thirds, the TF-IDF representation is essentially the same as bag-of-words in this case. The data is now ready to be used to train a classification model. Having covered the two simplest methods for translating a set of word tokens into numbers, bag-of-words and TF-IDF, you are encouraged to look into more advanced methods too, such as Google's Word2vec.

Naive Bayes

By now, you have been introduced to a number of classifiers that include K-nearest neighbors, logistic regression, decision trees, and support vector

machines. Naive Bayes is another classifier worth exploring. Recall the setup for the multi-class classification problem:

There is a dataset of N input–output pairs.

- The inputs x are feature vectors of length M
- The outputs y are integers of a class between 1 and K

The goal is to construct a classifier $h(x)$ that returns a class y for every input x .

As you may know, one of the models for solving this problem is multinomial logistic regression. The tuning parameters for this model are $K - 1$ vectors β , each with $M + 1$ entries. The model is evaluated by first computing an estimate of the conditional probability for each of the K classes. This is done using the formula shown here for \hat{P} of class κ and input x .

$$h_P(x) = \arg_K \max \hat{P}_\beta(Y = K | X = x)$$

The output class is then chosen to be the one with the largest conditional probability. The β vectors are trained by minimizing the cross-entropy loss over a training dataset.

Naive Bayes is not too dissimilar from multinomial logistic regression in that it also selects the class with the largest estimated conditional probability. However, it goes about finding those estimates in a different way. First, Naive Bayes uses features conditioned on classes by applying Bayes's rule. Second, it assumes that features X are independent of each other. In multilayer perceptron (MLP), positive reviews look like independent draws from a particular distribution of words. While negative reviews look

like independent draws from a different distribution of words. This works well in combination with a bag-of-words approach.

Developing the Naive Bayes model

Recall that the classifier is going to select the class that is most likely given the input. Applying Bayes rules, you get the probability of κ given X . The denominator in this formula does not depend on κ . And so, it scales all classes equally and does not alter the maximum. Hence, you can eliminate it to simplify the formula as shown.

$$P(X = x|Y = k) = P(x|k)$$

You can focus on the first of these two terms, the probability of x given κ . You can then expand out all of the m features in the notation. By the definition of the conditional probability, this equates to:

$$\begin{aligned} P(x, k) &= P(x_1, x_2, x_3, \dots, x_m | k) \\ &= P(x_1, x_2, x_3, \dots, x_{m-1} | x_m, k) P(x_m | k) \\ &= P(x_1, x_2, x_3, \dots, x_{m-2} | x_{m-1} x_m, k) P(x_{m-1} | x_m, k) P(x_m | k) \\ &= P(x_1 | x_2 \dots x_m, k) P(x_2 | x_3 \dots x_m, k) \dots P(x_m | k) \end{aligned}$$

You can continue to apply this definition repeatedly, each time extracting another feature from the first term and adding a term to the product. And you apply the assumption that the features are independent within each class κ . This means that the probability of any feature, given any number of other features and κ , equals:

$$P(x_i | x_j, k) = P(x_i | k)$$

Therefore:

$$P(x|k) = P(k) \prod_{i=1}^M p(x_i|k)$$

Notice that, without the assumption, you would have to estimate the full joint probability of all of the m features. If each of the x_i 's could take one of five values, you would need to estimate 5^M values. With the assumption, the problem is simplified by estimating M separate one-dimensional distributions. This means only $5M$ parameters need to be estimated instead of 5^M . This is a massive reduction in parameters.

Now that you have a nice formula for the conditional probability, consider its maximum. You can use the logarithm to convert the product into a sum.

$$\begin{aligned} h(x) &= \arg_k \max P(k) \prod_{i=1}^M P(x_i|k) \\ &= \arg_k \max \log P(k) + \sum_{i=1}^M \log P(x_i|k) \end{aligned}$$

If the features are bag-of-word-type counts, the $P(x_i = 4 \text{ occurrences})$, which equals $P(x_i = 1 \text{ occurrence})^4$. That is:

$$P(x_i|k) = P_{k_i}^{x_i}$$

Plug this relationship it into your model, to produce a linear function of parameters β , which are the logarithms of probability estimates.

$$\begin{aligned}
 h(x) &= \arg_k \max \log P_k + \sum_{i=1}^M \log P_{k_i}^{x_i} \\
 &= \arg_k \max \log P_k + \sum_{i=1}^M x_i \log P_{k_i} \\
 &= \arg_k \max \beta_{k_0} + \beta_k^T x
 \end{aligned}$$

Now, you need to estimate the values of the parameters from the training data. The probability of seeing a word i in a document of class k can be estimated as:

$$\hat{p}_{k_i} = \frac{N_{i_k}}{\sum_{i=1}^M N_{i_k}}$$

Similarly, the marginal probability of seeing a document of class k is estimated as:

$$\hat{p}_k = \frac{\sum_{i=1}^M N_{i_k}}{N} = \frac{\sum_{i=1}^M N_{i_k}}{\sum_{k=1}^K \sum_{i=1}^M N_{i_k}}$$

And so, your model is complete.

$$h(x) = \arg_k \max \log \left(\frac{\sum_{i=1}^M N_{i_k}}{N} \right) + \sum_{i=1}^M x_i \log \left(\frac{N_{i_k}}{\sum_{i=1}^M N_{i_k}} \right)$$

What happens if there is a word that is present in the training data in some classes but not in others? In that case:

$$N_{i_k} = 0$$

But since

$$\log\left(\frac{0}{N}\right) = \log(0) = -\infty$$

those classes will never be chosen. To correct this, you can add a positive number, α , to the numerator of the probability estimate. When you do this, you also have to add an α_M to the denominator, so the probabilities continue adding to one. This technique is called **Laplace smoothing**.

$$\log\left(\frac{\sum_{i=1}^M N_{i_k+\alpha}}{N + \alpha_M}\right)$$

Naive Bayes can be applied to any classification problem where the assumption of conditional independence of the features is appropriate. It simplifies complex text classification problems.

Training Evaluation

So far you have learned many of the basic concepts of NLP. Now, you will build a sentiment analysis model for Twitter data. Tweets are ideal for testing NLP because they are:

- Compact
- Emotionally unambiguous

Your goal is to sort tweets into positive and negative classes. You can begin by importing NLTK and using `nltk.download('twitter_samples')` to fetch the dataset. Next, use the **twitter_samples method** to load a set of positive and negatively labeled tweets into memory. You have a total of

10,000 tweets to work with—5,000 are positive and 5,000 are negative. Here's what your dataset looks like in a pandas table.

| | Tweet | Label |
|------|--|-------|
| 0 | #FollowFriday @France_Inte @PKuchly57 @Milipol... | 1 |
| 1 | @Lamb2ja Hey James! How odd :/ Please call our... | 1 |
| 2 | @DespiteOfficial we had a listen last night :)... | 1 |
| 3 | @97sides CONGRATS :) | 1 |
| 4 | yeaaaaah yippppy!!! My acct verified rqst has... | 1 |
| ... | ... | ... |
| 9995 | I wana change my avi but uSanele :(| 0 |
| 9996 | MY PUPPY BROKE HER FOOT :(| 0 |
| 9997 | where's all the jaebum baby pictures :((| 0 |
| 9998 | But but Mr Ahmed Maslan cooks too :(https://t... | 0 |
| 9999 | @eawoman As a Hull supporter I am expecting a... | 0 |

The middle column is the raw text of the tweet. And the rightmost column is the sentiment, 1 one for positive and 0 for negative.

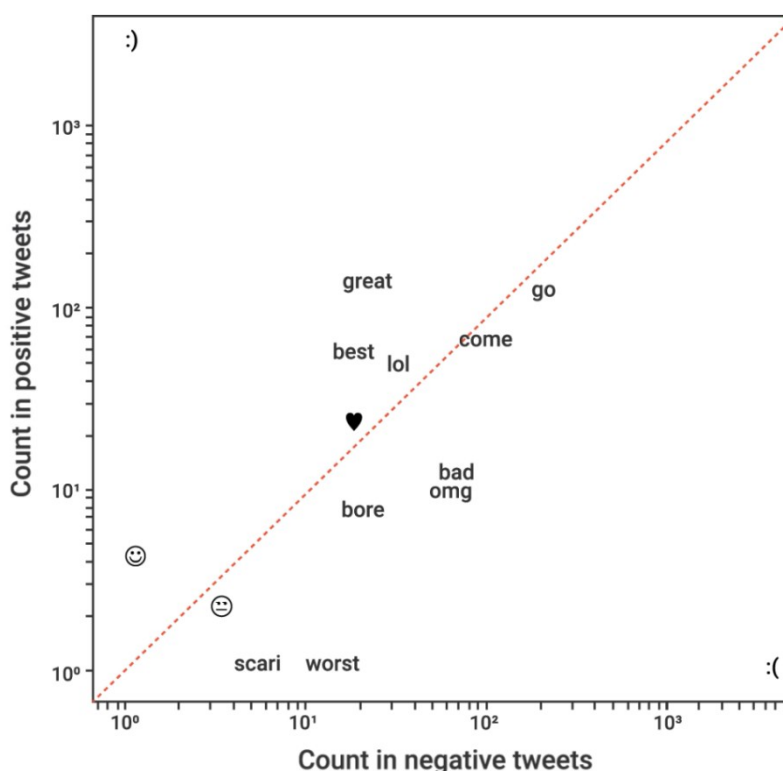
Normalizing the tweets

The first step in the process of converting raw text into usable numerical features is to separate out the test data using scikit-learn's **train_test_split**

method. Reserve one quarter of the data for testing and three quarters — or 7,500 tweets — for training.

Next, you can tokenize each tweet. NLTK provides a tokenizer designed especially for tweets, called **TweetTokenizer**. You remove the stop words and apply PorterStemmer to each tweet. Then, collect your preprocessing steps into a single function to process the test data. This method applies the tokenization, stemming, and stop word removal steps. You use it to create the process training data matrix, **X_train_pp**.

This scatterplot shows the number of times that a word appeared in a negative tweet on the x-axis and in a positive tweet on the y-axis. In general, you can expect to find negative words below the diagonal in this plot and positive words above the diagonal. Near the diagonal, you can expect to find words that are relatively neutral.




Building features

Having normalized the tweets, your next step is to build features. You can begin with a bag-of-words approach. To do this, first create a set with all of the unique words, or tokens, in the corpus.

```
unique_words = set()
for tweet in X_train_pp:
    unique_words.update(tweet)
len(unique_words)
```

In total, there are 10,225 unique words. And thus, your bag-of-words model will have 10,225 features. The bag-of-words matrix has one row for every tweet and a column for each unique word. You can easily populate this matrix by iterating through each word in each tweet of the training data.

And this is what it looks like as a pandas DataFrame.

| | whyi | jotzh | simpson | 2emt | https://t.co/cl0yoptuag | #messengerforaday | danger | librari | http://t.co/ccaoyyagj7 | http://t.co/hnbinh35 | ... | trash | lmbo | associ | http://t.co/dbwmkur3hk |  | Switch |
|------|------|-------|---------|------|-------------------------|-------------------|--------|---------|------------------------|----------------------|-----|-------|------|--------|------------------------|---|--------|
| 0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 1 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 2 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 3 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 4 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| 7495 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 7496 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 7497 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 7498 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 7499 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |

The total number of entries is the number of features, 10,225, times the number of tweets, 7,500. That's over 76 million entries. But the number of non-zero entries cannot exceed the total number of words in all the tweets

in the training set, which is only 61,487. That means that, at most, 0.08% of the entries in this matrix can be non-zero. Such sparse matrices are typical of NLP applications. Note that SciPy offers a special sparse matrix class for working with them more efficiently.

Training a classification model

Once you have cast your training data in the sparse bag-of-words representation, you are ready to use it to train a classification model. You can first try the simplest approach: The Naive Bayes model. In contrast to most other models, the full implementation of Naive Bayes does not require you to solve an optimization problem or even invert a matrix. You simply add up counts and divide.

For example, to determine the word count for each term i in positive and negative tweets you would use this line:

```
N_pos_i = bow_matrix_train[ind_pos, :].sum(axis=0)
```

And **alpha** is the Laplace smoothing parameter, which you set to one.

```
alpha = 1
```

You can use code to implement the formulas that you derived for the Naive Bayes model or use scikit-learn's implementation of Naive Bayes.

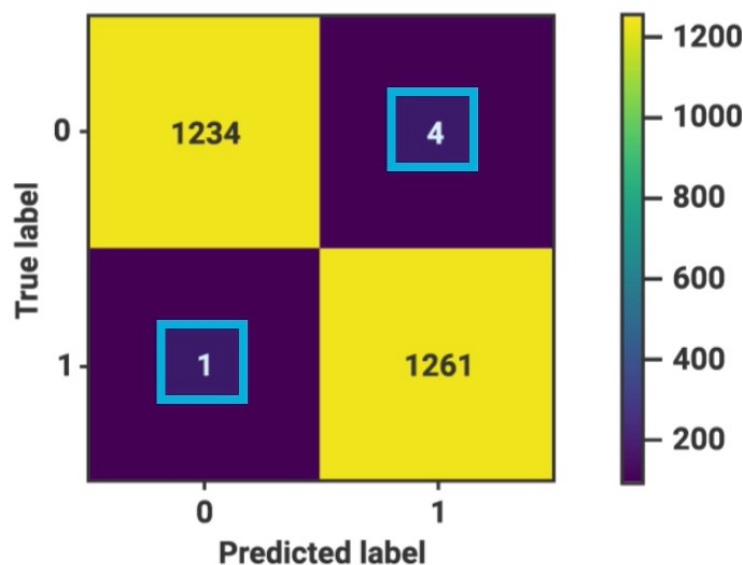
Testing the model

Your next step is to test the model using the test dataset. First, you put the test data through the same preprocessing and feature extraction steps as you did the training data.

Then you evaluate the accuracy of the model using the **score** method:

```
naive_bayes.score(bow_matrix_test, y_test)
```

And this is the result.



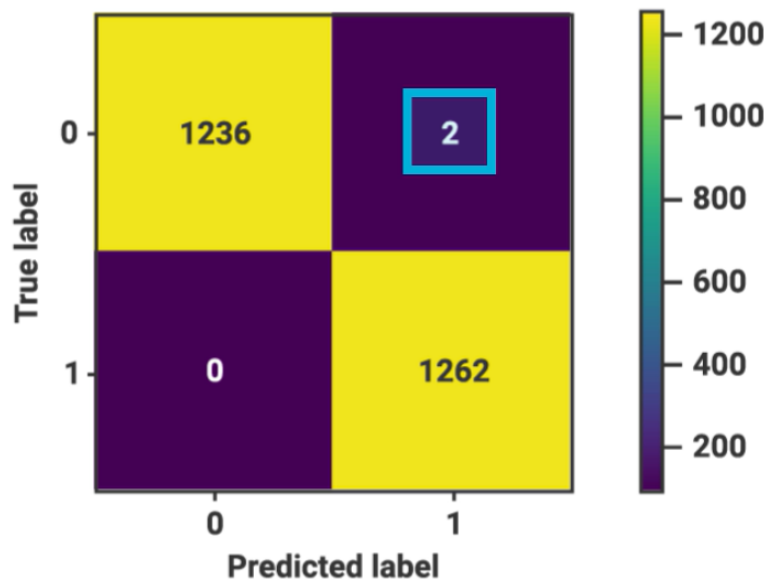
You may be surprised by the accuracy that this simple model achieves on classifying tweets. On the 2,500 test samples, it made only five mistakes.

Logistic regression

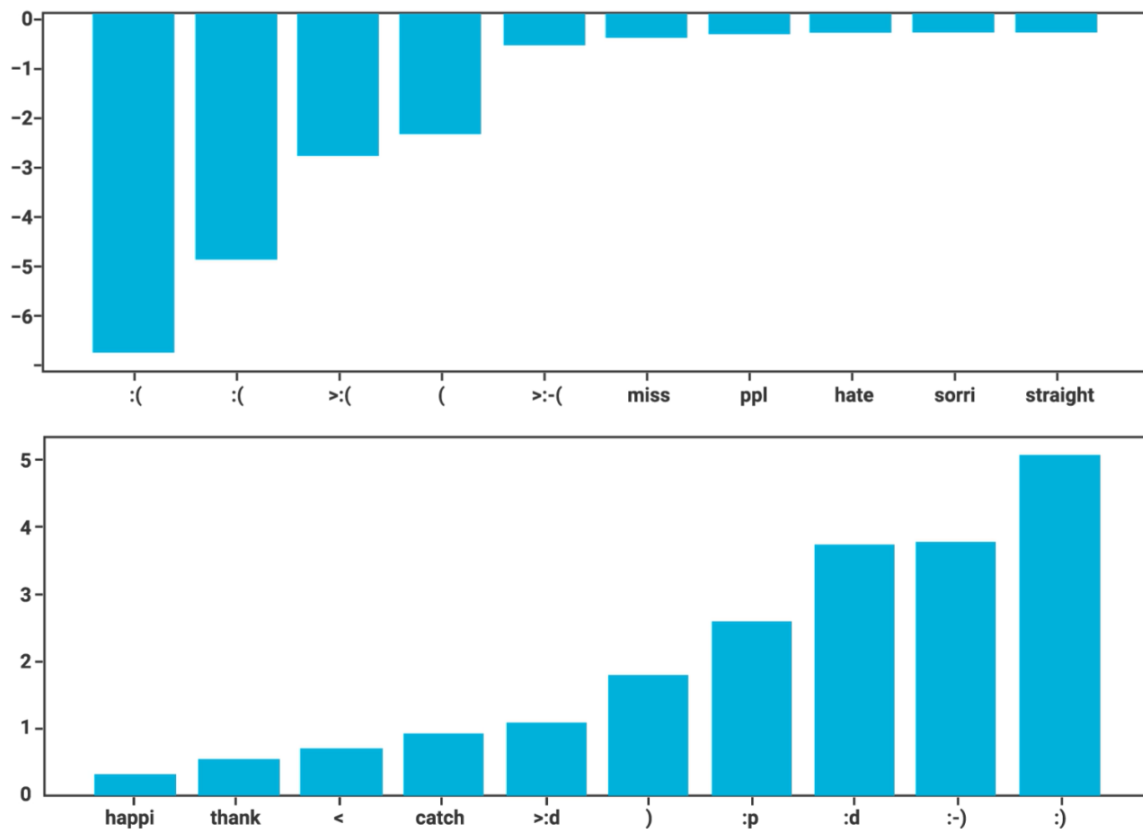
Just as before, you can train and score a logistic regression.

```
from sklearn.linear_model import LogisticRegression
lr_full = LogisticRegression().fit(bow_matrix_train,y_train)

lr_full.score(bow_matrix_test, y_test)
```

The model made only two mistakes on the test data—99,9% accuracy.



These two plots show the ten most negative and ten most positive coefficients of the logistic regression model.

Notice that the most significant feature for identifying negative and positive tweets, turn out to be the frowning and smiling emoticons, respectively. And these two appear in almost all of the tweets. Hence, the model can apply a simple rule: If the tweet has a smiling face, it is positive. If it has a frowning face, it is negative. And if this is true, you should be able to achieve high accuracy using only these two features.

Reduced training and testing datasets

Logistic regression and Naive Bayes actually improved after eliminating all but two features. They now achieve a 99.9% accuracy, each making only one mistake on the test dataset. This is because the dimension of the problem has been radically reduced and the density of non-zero entries in the feature matrix has increased.

NLP is currently one of the fastest-moving areas of artificial intelligence with cutting-edge research in machine translation, text generation, and conversational AI.