Module 18: Natural Language Processing (NLP)

Video Transcripts

Video 1: Introduction

Hi, and welcome to the module on natural language processing or NLP. NLP is a large topic whose central aim is to make human language accessible to computers. So far in this course, we've 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.

First, the order in which you say words usually has some bearing on its meaning. For example, if you say, "I had my car cleaned," it means something different than if you say, "I had cleaned my car." Even though both sentences contain exactly the same words.

Second, the words themselves may take on different meanings depending on their position in the sentence. 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, and this may surprise you. We humans are not always clear and completely precise in our communication. We don't always say exactly what we mean. And our styles of communication are highly individualized. And sometimes we make typos. NLP has to deal with all of this. And as a consequence, it is one of the most challenging and interesting fields of machine learning.

NLP has many applications with major impacts on our everyday lives. Internet search engines such as Google Search, use NLP to interpret our queries. NLP is also used to keep our inboxes free of spam. Machine translation is a difficult problem that has seen significant advances of late with the use of deep NLP. We use NLP to create dialogue systems and chatbots. Text summarization is an interesting application where you provide the machine with the large text, a book, for example. And it produces a short bullet point summary of the main ideas in the text. Text categorization is the application of classification techniques to text. Spam filtering is one example. Sentiment analysis is another, where we decide whether a tweet or a movie review is positive or negative.

Speech recognition systems such as Alexa, Siri, and Hey Google, operate on audio inputs as opposed to text-based inputs. And the list goes on. In this module, we will learn about the basic concepts of NLP and apply them to a simple text classification task. 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 we will not be able to cover in this course. However, the concepts that you learn here will help you to learn more about deep NLP if you choose to do so.

Let's start by understanding the similarities and differences between our standard machine learning workflow and the NLP workflow. In a number-based ML problem, we are given a raw dataset and our first task is to set aside a portion of that data, which we call the test dataset. This portion is put away and not touched until the test phase. The remainder is the raw training data, which we use to build our model. This data can have imperfections such as missing values or useless columns. And we correct these in the preprocessing stage. Here, it is often a good idea to normalize the data, meaning that we center its mean to zero and set its standard deviation to one.

Next, we might build features for the model by taking nonlinear combinations of the inputs. We do this, for example, in linear regression by creating polynomial or other nonlinear combinations. Finally, we feed these inputs to a model training procedure that chooses the parameters of the model such that a loss function is minimized. This model-building procedure can be repeated for different hyperparameters of the model, such as the regularization weight, or the number of features, or even for completely different model types, such as logistic regression versus SVM, versus decision trees.

We can compare the performance of these competing designs using a separate validation dataset, or with cross-validation. Once we are satisfied with the training process, we can take the final pipeline consisting of the preprocessing steps, the features, and the trained model, and apply it to the test data. The test input is fed through this 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.

The raw dataset is usually called a corpus and it is some large body of text, perhaps the IMDB database of movie reviews or all of Wikipedia. Each sample in the corpus is called a document. A document can be a single review, or a Wikipedia article, or a tweet. We split the corpus into training and testing datasets just as before. 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. We will learn more about these in the upcoming videos. The preprocessing steps include text tokenization and text normalization. Text tokenization is the process of splitting the text into separate grammatical units, or tokens. Normalization includes a host of operations on those tokens that help to reduce them to a core set that captures the important information in the document.

The rest of the workflow, including how we 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. In the next video, we will install NLTK and begin using it to perform various texts processing tasks. I'll see you then.

Video 2: NLTK Token

Hi. In this video, we will learn about some of the useful preprocessing steps that we can apply to text documents to prepare them for usage in machine learning models. To do this, we will use the Natural Language toolkit, or

NLTK. NLTK is an open-source set of Python modules that provide access to datasets and algorithms used in NLP. It is similar to scikit-learn in that it is a simple and accessible entry point to a complex topic. 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, as I do, you can use Anaconda. With Anaconda installed on your machine, you can create an environment by entering this string into the command line. And then 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, which would be inconvenient to store on your computer. Instead, it offers a download manager that you can use to retrieve specific datasets. To launch the downloads...downloads manager, type nltk.download() into a Jupyter Notebook. This opens a window with all of 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. Now, recall that the NLP workflow involves preprocessing and feature extraction. In this video, we will describe the different types of preprocessing operations, beginning with tokenization. A token in NLP is a single unit of text. What constitutes a unit depends on your goal. With NLTK, one can tokenize text into sentences or into word tokens. Here we have a review of the 2002 sci-fi action thriller, Minority Report. It reads, "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!" Let's imagine that our goal is to build a model that can identify this review as being positive, which indeed it is. We begin by splitting the text into individual words using the word_tokenize() method. 'Words' in this context include punctuation marks, numbers, names, URLs, et cetera. Here we see that the sample text contains a total of 36 words. Looking closely at the output, we can observe that the tokenizer has made some interesting decisions.

For example, it included the opening quotation mark with the word Minority, while separating the closing quotation mark from the word Report. Also, it split the word wasn't into "was" and "n't." There are many different tokenizers to choose from, each with its own set of rules. Furthermore, tokenizers are language specific. Written English is relatively easy to tokenize by splitting over spaces. Other languages with composite words or more complicated demarcation rules, such as German and Chinese, can be considerably more difficult 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, and it's the topic of our next video. I'll see you then.

Video 3: Normalization

Text normalization refers to operations that take tokens as input and produce tokens as output. This is different from feature extraction and NLP, which take tokens and produce numbers. There are many normalization operations that you can apply to your text. And we will not cover all of them here, but they all serve the purpose of improving the quality of your tokens in some way.

For example, say you're interested in categorizing news articles by topic. Then you might want to convert uppercase words such as HOME RUN to lowercase, since both of these provide equal evidence that the document is sports related. Autocorrecting spelling errors is also possible. However, it is a little tricky, since there may be multiple options. For example, the misspelled word 'fask' may refer to task, flask, ask, mask, et cetera.

It is common to remove characters or words that provide little information. Special characters such as dollar signs, ampersands, hashtags, and @ symbols should in some cases be removed. So-called stop words, such as am, is, are, et cetera, are so common as to be relatively uninformative. They tend to be scattered uniformly throughout the text. And so, removing them may actually improve the performance of the model.

Let's look more closely at the remaining items. Tagging parts of speech, named entity recognition, stemming, and lemmatization. It is sometimes useful to identify the part of speech of each word in a document. 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—"The movie was long but it was not boring"—has a standard part of speech designation. The word "the" is a determiner, "movie" is a singular noun. "Was" is a verb in the past tense, and so on. NLTK's pos_tag() method can be used to obtain the parts of speech in a list of words.

Here we see the parts of speech of all of the words in our movie review. Notice that the names—Tom and Cruise, and Steven and Spielberg—had been correctly identified as proper nouns or NNPs. But so is the word Report in Minority Report. This is probably because it was capitalized. You can find the meaning of each of the parts of speech tags in...with the upenn_tagset() method. Here we see that PRP$ indicates a possessive pronoun.

Once we have identified the parts of speech, the next step to understanding a document is often to identify named entities. Named entities are noun phrases that indicate particular items in the world. This table shows some of the most common named entity types, such as organization, person, location, date, and time. This somewhat complicated code picks out the named entities in our review. Notice that the code is based on NLTK's ne_chunk() method.

This code correctly identifies Tom Cruise and Steven Spielberg as 'PERSON' named entities. But strangely, it also thinks that the word Report, Great, and Movie are organizations. Again, it seems the capitalization has thrown it off. We can use these named entity tags to, for example, remove the names of people from the review. After doing that—and also setting everything to lowercase letters—the review reads, "i enjoyed 'minority report'. did n't disappoint, and is at the top of his game. the movie was long but it was n't boring. great movie!" This is certainly less clear than the original. We no longer know who is at the top of his game. But we can still sense that the reviewer was happy with the movie. We can even go farther and remove all of the stop words. Recall that stop words are very common words that do not convey much useful information.

A list of English stop words can be loaded using the stopwords method from the nltk.corpus module. Here's what our review looks like after filtering out those stop words: "enjoyed 'minority report' n't disappoint, top game.

movie long n't boring. great movie!" That's pretty neat, it reads like a telegram. We've achieved an almost 50% compression of the text from 36 to 19 words, without sacrificing its essential message.

The final two operations, 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, we'd like to 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. As with any stemmer, PorterStemmer can produce non-words. As we see here, it correctly maps joy, joyful, and joyfully to joy. However, joyous becomes 'j-o-y-o-u' and geese becomes 'g-e-e-s,' both of which are not English words. Is this a problem? Well, keep in mind that machine learning models such as logistic regression and SVM do not know the meaning of words. And so, they can just as easily correlate the character sequence 'j-o-y-o-u' with a positive sentiment, as they can 'j-o-y.' So breaking free of the real words constraint can be a good thing.

Finally, lemmatization is similar to stemming. However, the outputs are required to be 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, we call the lemmatize method on the WordNetLemmatizer object. We can see here that the lemmatizer does not actually reduce the number of tokens in this case. Joy, joyful, joyfully, and joyous are all mapped to themselves.

But it does correctly convert geese to goose. Lemmatization can be useful in situations, such as document summarization, where some measure of grammatical coherence should be preserved. Here we see the result of stemming and lemmatization on our movie review. The stemmed version is shorter and probably better for the task of sentiment analysis. Alright, that's text normalization. In the next video, we will dive into feature extraction with the bag of words and TF-IDF representations. I'll see you then.

Video 4: Bag of Words and TF–IDF

Hi. So far in this module we've learned how to process text using tokenization and normalization. These steps convert our documents into a small set of tokens that capture the essence of the text. Our next task is to convert those tokens into a numerical format that can be used to train a machine learning model. This step is called feature extraction or feature representation. There are many ways to convert a set of tokens into numerical features. The simplest of these is bag of words. The next step up is the TF-IDF approach. And beyond that, we have a full-word vectorization. This last one is an advanced technique that is beyond our scope.

As usual, which approach you choose will depend on the application, the amount of data that you have, and ultimately the performance of the resulting model. The bag-of-words concept is simple. You simply create a single feature for every distinct token in the training data. And then you enter for each document the number of occurrences of each token in that document. Say, for example, we have processed three movie reviews and boiled them down to their essential tokens. The first review says 'enjoy,' 'disappoint,' 'bore,' and 'great.' The second says 'bore,' 'disappoint,' 'bore,' and 'bore.' And the third says 'great,' 'great,' and 'enjoy.' There are a total of four unique tokens in this corpus. So, we get four feature columns.

In each column, we enter the number of times that that feature appears in the respective document. This table is now our feature matrix; along with the output of positive and negative sentiment, it constitutes our training dataset for, say, a logistic regression model. 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 that...that may not be a problem in cases where we are not interested in who did the cleaning.

Another problem with the bag-of-words approach is that it does not distinguish between informative and uninformative words. This is especially important when we are dealing with large documents with perhaps thousands of feature columns. Many of these will correspond to uninformative words that were not discarded as stop words. The TF-IDF approach attempts to correct this by quantifying the usefulness of each token. Here, each entry in the feature table is not an integer count, but a real valued TF-IDF score. These scores are computed as the product of the term frequency, tf, and the inverse document frequency, idf.

The term frequency of a term, t, in a document, d, is computed as the number of times that that term occurs in that document, divided by the total number of words in the document. The numerator here is just the bag-of-words counts. And so, the term frequency is simply a normalized version of bag of words. Here we see the tf scores for our three reviews. Just like bag of words, tf scores are meant to give larger weight to terms that appear more often in the document.

So the reviewer who said 'bore,' 'bore,' 'bore' is considered to be three times more likely to have found the movie boring than the one who said 'bore' only once. The inverse document frequency of a term, t, quantifies the importance of that term as the negative log of the number of documents where it appears, divided by the total number of documents. That is, the negative log of the proportion of documents that contain the term. Here's a plot of the idf as a function of the proportion of documents containing the term. Notice first that…that the idf is always positive. Notice also that it 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—'enjoy,' 'disappoint,' 'bore,' and 'great'—appear in two of the three documents. So, they all have the same idf score of negative log of two thirds. Thus the IDF...TF-IDF representation is essentially the same as bag of words in this case. OK? We have covered the two simplest methods for translating a set of word tokens into numbers, bag of words and TF-IDF. I encourage you to look into the more advanced methods such as Google's Word2vec if you're so inclined. With normalization, the data is now ready to be used to train a classification model, which can be any of the models that we have covered in this course.

In the next video, we will learn about yet another model that works particularly well for text classification tasks. I'll see you then.

Video 5: Naive Bayes

In this video, we will add a new model to our collection of classifiers, which so far includes K-nearest neighbors, logistic regression, decision trees, and support vector machines. The model is called Naive Bayes. We begin by recalling the setup for the multi-class classification problem. We have a dataset consisting of N input–output pairs. The inputs, x, are feature vectors of length M. And the outputs are integers between one and K, indicating the class to which the input vector belongs. Our goal is to construct a classifier—which is a function, h, that for any new input, x, returns a predicted class for that input.

One of the models that we've been that we've seen for solving this problem is multinomial logistic regression. The tuning parameters for this model are K minus 1 vectors β, each one containing M plus 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 P-hat of class kappa and input 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, it inverts the conditioning—from classes conditioned on features to features conditioned on classes—by applying Bayes's rule. Hence the Bayes in the name. Second, it assumes that features X are independent of each other when conditioned on the class.

This is a pretty strong assumption and it's the reason it's called a naive estimator. Although some people prefer to call it independent base. In NLP, it amounts to asserting that, for example, 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 is not a bad assumption, if we have already committed to a bag-of-words representation for our documents. We will now develop the Naive Bayes model.

Recall that the classifier is going to select the class that is most likely given the input. That is arc max of P of kappa, given X. Applying Bayes rules, we get that the probability of kappa given X is equal to the probability of X given kappa, times the probability of kappa, divided by the probability of X. The denominator in this formula does not depend on kappa. And so it scales all classes equally and does not alter the maximum. Hence, we can eliminate it from the formula.

And so we get that h(x) is equal to arc max over kappa of the probability of X given kappa times the probability of kappa. We will now focus on the first of these two terms, the probability of x given kappa. And begin by expanding out all of the m features in the notation. By the definition of the conditional probability, this equals the probability of x_1 through x_m minus 1, given x_m and kappa, times the probability of x_m given kappa. We can continue to apply this definition repeatedly, each time extracting another feature from the first term and adding a term to the product.

After going through all of the m features, we end up with the last line. The probability of x given kappa equals the probability of x_1 given x_2 through x_m. And kappa times the probability of x_2, given x_3 through x_m and kappa, and so forth. All the way to the probability of x_m given kappa. OK? So it may seem like we've just been playing around with the symbols without accomplishing much. So here comes the important part. We now apply our assumption that the features are independent within each class kappa.

This means that the probability of any feature, given any number of other features and kappa, equals the probability of that feature given kappa alone. If this is true, then the previous long equation—for the probability of x given kappa—boils down to the probability of the class, kappa, times the product of individual features given kappa. Notice what has happened here because it's important. Without the assumption, we would have…we would have had to estimate the full joint probability of all of the m features. If each of the x_i's could take on one of, say, five values; that would be 5^M parameters to estimate. With the assumption, however, we have simplified the problem to estimating M separate one- dimensional distributions. So just five times M parameters. For modest M of, say, 20 that is a reduction from about 100 trillion parameters to just ten.

Now that we have a nice formula for the conditional probability, the next thing is to take its maximum. We apply the standard trick of taking the logarithm in order to convert the product into a sum. And so we obtain the arg max of log of P of kappa plus the sum over features of log of P of x_i given kappa. Now let's imagine that the features are bag-of-word-type counts. Then the probability of seeing, say, x_i equals four occurrences of the i'th word in a document, of class kappa, equals the probability of seeing one occurrence of that word to the fourth power. That is P(x_i) given kappa equals some number that depends on kappa and i, elevated to x_i. This is a very convenient relationship because, when we plug it into our model, the x_i comes out of the log and the whole thing becomes a linear function of parameters β, which are the logarithms of probability estimates.

So our model becomes h is equal to arg max over kappa of a linear combination of features. That's great. We now have a simple linear model with very few parameters. All that remains is to estimate the values of the parameters from the training data. And we can do that simply by counting. The probability of seeing word i in a document of class kappa can be estimated as the total number of times we have actually seen that word, in class kappa documents, divided by the total number of words in class kappa documents. Similarly, the marginal probability of seeing a document of class kappa is estimated as the total number of documents of that class divided by the total number of documents.

Alright then, our model is complete. Or nearly complete because there is still one more wrinkle. What if there is a word that is present in the training data in some classes but not in others? In that case, there will be some N_i kappas equal to zero and the log of 0 is negative infinity. And, hence, those classes will never be chosen. To correct this, we add a positive number, alpha, to the numerator of the probability estimate. And when we do this, we also have to add an alpha_M to the denominator, so the probabilities continue adding to one. This technique is called Laplace smoothing. And with that, we are truly done deriving the Naive Bayes estimator.

Naive Bayes can be applied to any classification problem where the assumption of conditional independence of the features is appropriate. It is actually a very popular method for text classification since these are often very high-dimensional problems. And they benefit from the simplifying assumption of Naive Bayes. In the next video, we'll put Naive Bayes to the test and compare it with logistic regression on a real-world text classification task. I'll see you then.

Video 6: Training Evaluation

Hello. So far you have learned many of the basic concepts of NLP. We are now ready to put them to use in building a full sentiment analysis model for Twitter data. Like movie reviews, tweets offer a very nice toy dataset for testing NLP. They are compact, consisting of no more than 280 characters. And they are often, how shall I put this, emotionally unambiguous. Our goal will be to build a binary classifier that sorts tweets into positive and negative classes. We 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.

We have a total of 10,000 tweets to work with—5,000 are positive and 5,000 are negative. Here's what our dataset looks like in a pandas table. 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. @97sides says, "CONGRATS :)." And this has been labeled as a positive tweet. But, further down, someone's puppy has broken her foot, ":(." This is certainly a negative tweet. How can we use the normalization and feature extraction tools that we have learned to convert this raw text into usable numerical features?

The first step in the process is, of course, to separate out the test data. And we can do that with scikit-learn's train_test_split method. We will reserve one quarter of the data for testing and three quarters–or 7,500 tweets—for training. Next, we must tokenize each tweet. And interestingly enough, NLTK provides a tokenizer designed especially for tweets, called the TweetTokenizer. This tokenizer does tweet-specific things such as removing handles and also being careful not to alter emoticons when changing the case of words.

Observe that the 89th tweet has been stripped of all of its handles, and lowercased, but without altering the D in the laughing emoticon. We will keep the normalization steps to a minimum in this demo. We simply remove the stop words and apply PorterStemmer to each tweet. Unfortunately, the stemmer is not very smart and has ruined our laughing emoticon. Oh well. It is always a good idea to collect your preprocessing steps into a single function, since you will have to process the test data as well. This method applies the tokenization, stemming, and stop word removal steps that we just saw. And we 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. See, for example, that the word 'worst' showed up about ten times in negative tweets but only once in a positive tweet. This makes sense, since worst is a pretty negative word and we are more likely to use negative language in negative tweets. So, in general, we would expect to find negative words below the diagonal in this plot and positive words above the diagonal. Near the diagonal, we expect to find words…find words that are relatively neutral.

And if you look closely, that is more or less what we do find: Positive words like 'great,' 'best,' 'lol,' and smiling emoticons are above the diagonal. While negative words like 'bore,' 'bad,' 'scari,' and 'worst' are below. 'Omg' is an exception, but it seems to be used mostly in negative tweets. Neutral words, like 'come' and 'go,' are very near the diagonal. Notice also that the smiling and frowning emoticons are off on the extreme corners of positivity and negativity.

These symbols are both very common and highly predictive of the sentiment of the tweet. This means that they will be strong features in our model, as we shall see. Having normalized the tweets, our next step is to build features. We will begin with a bag-of-words approach. To do this, we must first create a set with all of the unique words, or tokens, in the corpus. In total, we have 10,225 unique words. And thus our 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. We 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.

It may seem surprising that all we see here are zeros. Indeed, this is a very sparse matrix. 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 of 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. And SciPy offers a special sparse matrix class for working with them more efficiently. This is an interesting and important topic in its own right, but it's beyond the scope of the present course and so we will leave it at that.

Now that we have cast our training data in the sparse bag-of-words representation, we are ready to use it to train a classification model. Let's first try the simplest approach, which is the Naive Bayes model that we just learned. What you see here is the full implementation of Naive Bayes, which in contrast to every other model we've seen, does not require that we solve an optimization problem or even invert a matrix. All we have to do, is to add up counts and divide. Alpha here is the Laplace smoothing parameter which we set to one. Take a little time to verify that this code implements the formulas that we derived for the Naive Bayes model in the previous video.

Of course, there's also the option of using scikit-learn's implementation of Naive Bayes. And here we verify that it produces exactly the same values for the feature coefficients as our handcrafted implementation. Our next step is to test the model using the test dataset. To do this, we must first put the test data through the same preprocessing and feature extraction steps as we did the training data. Once we do that, we can evaluate the accuracy of the model using its score method. What we find is a little surprising. This extremely simple model achieves a 99.8% accuracy on the…on classifying tweets. On the 2,500 test samples, it made only five mistakes.

That is pretty impressive. What about logistic regression? Can it compete against Naive Bayes? Just as before, we train and score a logistic regression model and, wow, we now get 99.9% accuracy. This model made only two mistakes on the test data. Again, very impressive. What's going on here? Well, these two plots hold the answer. They 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. 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, then we should be able to achieve high accuracy using only these two features. So let's try that. I've omitted the code for building the reduced training and testing datasets. But here are the results. 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 five thousand-fold. And the density of non-zero entries in the feature matrix has increased from 0.08% to 40%.

There are many other questions that we could investigate. What is the accuracy achieved if we exclude emoticons altogether? Can TF-IDF improve performance? What about other classifiers such as support vector machines and decision trees? But I think this is a good place to leave it. As I said at the beginning, NLP is a vast topic. And in this module we have barely scratched the surface. NLP is currently one of the fastest-moving areas of artificial intelligence with cutting-edge research in machine translation, text generation, and conversational AI. I hope that these lectures have inspired you to look deeper into this fascinating topic.