M01S02-Basic Lexical Processing

Natural language processing – Lexical Processing

Introduction [1/11]

Various pre-processing steps needed to be applied before doing any kind of text analytics such as apply machine learning on text, building language models, building chatbots, building sentiment analysis systems and so on. These steps are used in almost all applications that work with textual data. We will also build a spam-ham detector system side-by-side on a very unclean corpus of text. Corpus is just a name to refer to textual data in NLP jargon.

We have already built a spam detector while learning about the naive-bayes classifier. Here, we will learn all the pre-processing steps that one needs to do before using a machine learning algorithm on the spam messages dataset. Pre-processing steps taught here are not limited to building a spam detector.

Specifically:

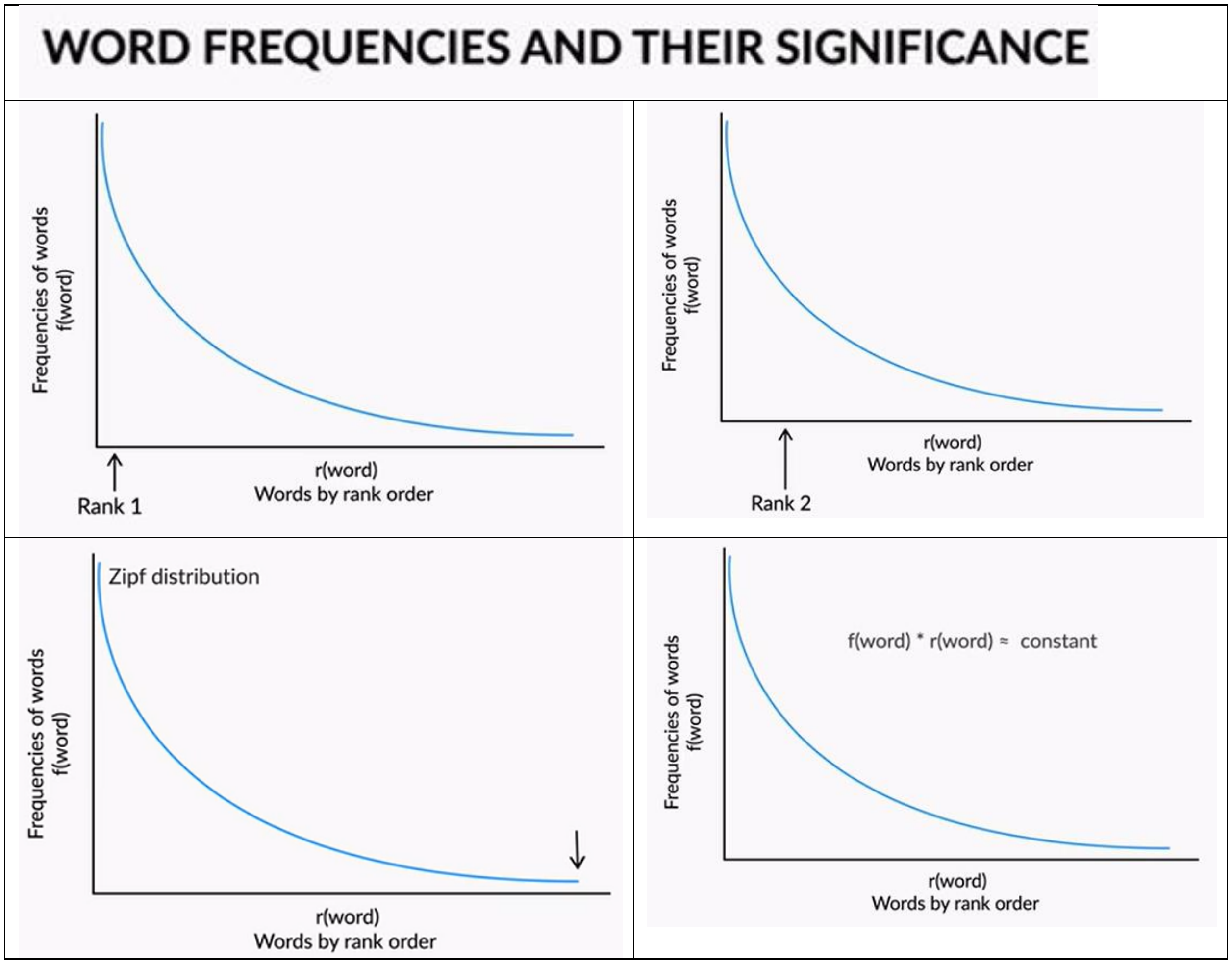
- How to pre-process text using techniques such as
    - Tokenisation
    - Stop words removal
    - Stemming
    - Lemmatization
- How to build a spam detector using one of the following models:
    - Bag-of-words model
    - TF-IDF model

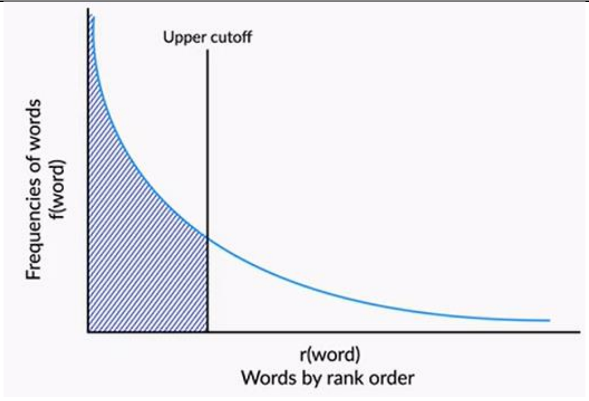
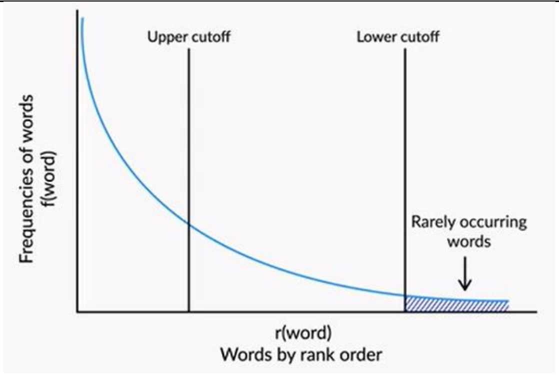
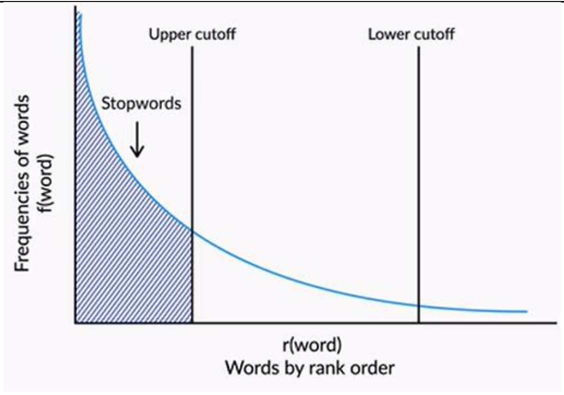
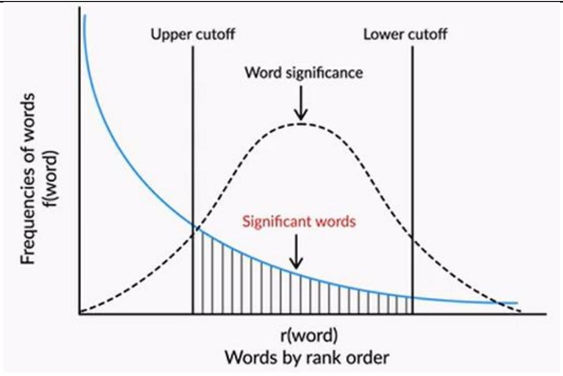
Word Frequencies and Stop Words [2/11]

While working with any kind of data, the first step is to explore and understand it better. To explore text data, some basic pre-processing steps needed. Next few segments cover some basic pre-processing and exploratory steps applicable to almost all types of textual data.

A text is made of characters, words, sentences and paragraphs. The most basic statistical analysis is to look at the **word frequency distribution,** i.e. visualising the word frequencies of a given text corpus.

There is a common pattern when plotting word frequencies in a large corpus of text, such as a corpus of news articles, user reviews, Wikipedia articles, etc. Lectures will demonstrate some interesting insights from word frequency distributions as also what **stopwords** are and why they are lesser relevant than other words.



	 80:20 rule
 Language Builder words	

Zipf's law (discovered by the linguist-statistician George Zipf) states that the frequency of a word is inversely proportional to the rank of the word, where rank 1 is given to the most frequent word, 2 to the second most frequent and so on. This is also called the **power law distribution.**

The Zipf's law helps us form the basic intuition for **stopwords -** these are the words having the highest frequencies (or lowest ranks) in the text and are typically of limited 'importance'.

Broadly, there are three kinds of words present in any text corpus:

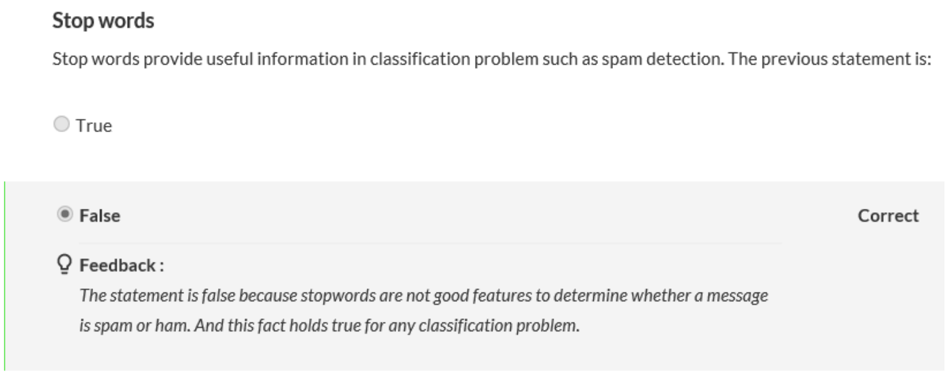
- Highly frequent words, called stop words, such as ‘is’, ‘an’, ‘the’, etc.
- Significant words, which are typically more important to understand the text
- Rarely occurring words, which are again less important than significant words

Stopwords are removed from the text for two reasons:

1. They provide no useful information, especially in applications such as spam detector or search engine. Therefore, you’re going to remove stopwords from the spam dataset.
2. Since the frequency of words is very high, removing stopwords results in a much smaller data as far as the size of data is concerned. Reduced size results in faster computation on text data. There’s also the advantage of a smaller number of features to deal with if stopwords are removed.

However, there are exceptions when these words should not be removed. In POS (parts of speech) tagging and parsing, stopwords are preserved because they provide meaningful (grammatical) information in those applications. Generally, stopwords are removed unless they prove to be very helpful in your application or analysis.

On the other hand, do not remove the rarely occurring words because they might provide useful information in spam detection. Also, removing them provides no added efficiency in computation since their frequency is so low.



Stop words

Stop words provide useful information in classification problem such as spam detection. The previous statement is:

True [✓ False] **Correct**

Feedback :
The statement is false because stopwords are not good features to determine whether a message is spam or ham. And this fact holds true for any classification problem.

After word frequencies and stopwords, let’s see how to make the frequency chart on your own. How to make frequency distribution from a text corpus and how to remove stopwords in python using the NLTK library.

Tokenisation [3/11]

We are going to build a spam detector where we will use word tokenisation, i.e. break the text into different words, so that each word can be used as a feature to detect whether the given message is a spam or not.

Take a look at the spam messages dataset to get a better understanding of how to approach the problem of building a spam detector. There is a lot of noise in the data. Noise is in the form of non-uniform cases, punctuations, spelling errors. These make it hard for anyone to work on text data.

There is another thing to think about - how to extract features from the messages so that they could be used to build a classifier. When creating any ML model such as a spam detector, we need to feed in features related to each message that the ML algorithm can take in and build the model. But, in the spam dataset, only two columns - message and the label related to the message. ML works on numeric data, not text. Earlier when working with text columns, we either treated them as categorical variables and converted to numeric variable by either assigning numeric values to each category, or dummy variables. Here, none is possible, since the message column is unique and not a categorical variable. If treated it as a category, model will fail miserably.

To deal with this problem, we extract features from the messages through **tokenization**. From each message, we extract each word by breaking each message into separate words or 'tokens'. Split the text into smaller elements. These elements can be characters, words, sentences, or even paragraphs depending on the application.

In the spam detector case, each message broken into different words, so it’s called **word tokenisation**. In other types of tokenisation, character tokenisation, sentence tokenisation, etc are done. Different types of tokenisation are needed in different scenarios.

Now, let’s take a look at what exactly tokenisation is and how to do it in NLTK.

There are multiple ways of doing things in Python. To tokenise words, we can use the split() method to splits text on white spaces, by default. This method doesn’t always give good results. NLTK’s tokeniser handles various complexities of text. It handles **contractions** such as “can’t”, “hasn’t”, “wouldn’t”, and other contraction words and splits these up although there is no space between them. But it doesn’t split words such as “o’clock” which is not a contraction word.

NLTK has different types of tokenisers present. The most popular are:

1. **Word tokeniser** splits text into different words.
2. **Sentence tokeniser** splits text in different sentence.
3. **Tweet tokeniser** handles emojis and hashtags that you see in social media texts
4. **Regex tokeniser** lets you build your own custom tokeniser using regex patterns of your choice.

Bag-of-Words Representation [4/11]

Till now, two pre-processing steps - tokenisation and removing stopwords. But these still can’t give the list of words to train an ML model.

Learn how to represent text in a format that can be fed into ML algorithms. The most common and most popular approach is to create a **bag-of-words representation** of the text data. The central idea is that any given piece of text, i.e., tweets, articles, messages, emails etc., can be “represented” by a list of all the words that occur in it (after removing the stopwords), where the **sequence of occurrence does not matter**. Visualise it as the “bag” of all “words” that occur in it. For example, consider the messages:

“Gangs of Wasseypur is a great movie”

The bag of words representation for this message would be:

gangs
wasseypur
great
movie

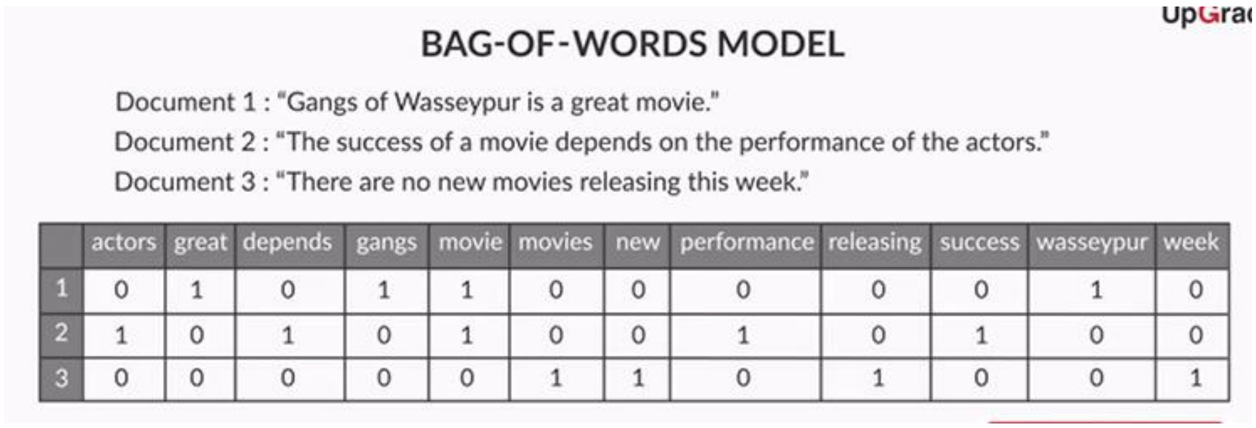
Bag-of-words representation

How to build a spam classifier from “bags” for representing each of the messages in training and test data set?

Let’s say the bags, for most of the spam messages, contain words such as prize, lottery etc., and most of the ham bags don’t. Now, whenever a new message comes, look at its “bag-of-words” representation. Does the bag for this message resemble that of messages already known as spam, or does it not resemble them? Based on the answer to the previous question, classify the message.

How do you get a machine to do all of that? Represent all the bags in a matrix format, after which use ML algorithms such as naive Bayes, logistic regression, SVM etc., to do the final classification.

But how is this matrix representation created?



BAG-OF-WORDS MODEL

Document 1 : "Gangs of Wasseypur is a great movie."

Document 2 : "The success of a movie depends on the performance of the actors."

Document 3 : "There are no new movies releasing this week."

	actors	great	depends	gangs	movie	movies	new	performance	releasing	success	wasseypur	week
1	0	1	0	1	1	0	0	0	0	0	1	0
2	1	0	1	0	1	0	0	1	0	1	0	0
3	0	0	0	0	0	1	1	0	1	0	0	1

Text is represented in the form of matrix. It can be used to train ML models. Each document sits on a separate row and each word of the vocabulary (**features** of the text) has its own column.

Also called bag-of-words model but not to be confused with a ML model. A bag-of-words model is the matrix that built from text data.

Values inside any cell can be filled in two ways –

1. Fill the cell with the frequency of a word (i.e. a cell can have a value of 0 or more),
2. Fill the cell with either 0, in case the word is not present or 1, in case the word is present (binary format).

Both approaches work fine and don't usually result in a big difference. The frequency approach is slightly more popular and the NLTK library in Python also fills the bag-of-words model with word frequencies rather than binary 0 or 1 values.

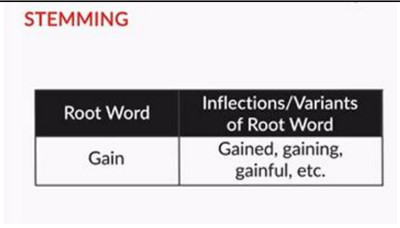
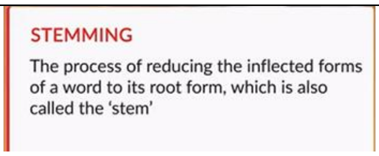
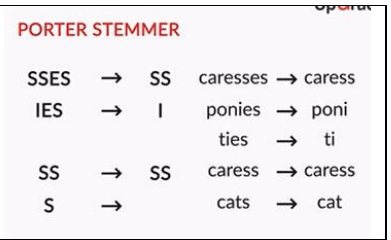
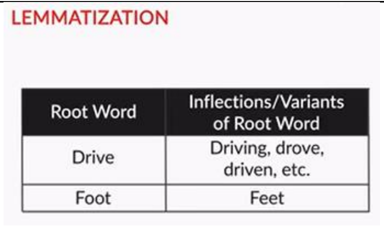
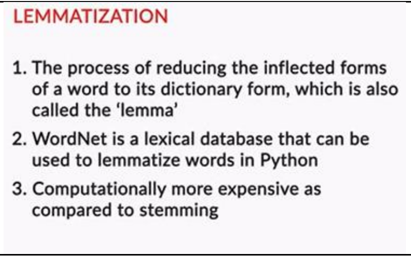
To build a bag-of-words model in Python, use the scikit-learn library. A lot of redundant features after building the model. There were features such as 'get' and 'getting', 'goes' and 'going', 'see' and 'seeing' and along with a lot of other duplicate features. They are not exactly duplicate but they're redundant in the sense that they're not giving any extra information about the message. In fact, the words 'winner' and 'win' are equivalent when goal is to detect whether a message is spam or not.

Keeping these duplicates separate will hinder the performance of the ML algorithm. Redundancy is going to increase the number of features due to which the classifier can face the *curse of dimensionality*. To get rid of this problem, use two more pre-processing techniques - **stemming** and **lemmatization**.

Stemming and Lemmatization [5/11]

Redundant tokens result in an inefficient model when building spam detector. **Stemming** makes sure that different variations of a word, say ‘warm’, warmer’, ‘warming’ and ‘warmed,’ are represented by a single token - ‘warm’, because they all represent the same information (represented by the 'stem' of the word).

Another similar pre-processing step (and an alternative to stemming) is **lemmatisation.**

 STEMMING Root Word | Inflections/Variants of Root Word Gain | Gained, gaining, gainful, etc.	 STEMMING The process of reducing the inflected forms of a word to its root form, which is also called the ‘stem’	 PORTER STEMMER SSES → SS caresses → caress IES → I ponies → poni ties → ti SS → SS caress → caress S → cats → cat
 LEMMATIZATION Root Word | Inflections/Variants of Root Word Drive | Driving, drove, driven, etc. Foot | Feet	 LEMMATIZATION 1. The process of reducing the inflected forms of a word to its dictionary form, which is also called the ‘lemma’ 2. WordNet is a lexical database that can be used to lemmatize words in Python 3. Computationally more expensive as compared to stemming	

Repeated tokens or features were nothing but a variation or an **inflected form** of the other token. ‘seeing’ is an inflection of the word ‘see’. ‘limited’ is an inflection of the word ‘limit’. The two techniques reduce these inflected words to the original base form. But which one is a better technique in what situations?

Stemming

A **rule-based** technique that just chops off the suffix of a word to get its root form, which is called the ‘stem’. If using a stemmer to stem the words of the string - "The driver is racing in his boss’ car", ‘driver’ and ‘racing’ will be converted to their root form by chopping of the suffixes ‘er’ and ‘ing’. ‘driver’ will be converted to ‘driv’ and ‘racing’ will be converted to ‘rac’.

Even though the root forms (or stems) don’t resemble the root words - ‘drive’ and ‘race’, no need to worry because stemmer will convert all variants of ‘drive’ and ‘racing’ to those root forms. So, it will convert ‘drive’, ‘driving’, etc. to ‘driv’, and ‘race’, ‘racer’, etc. to ‘rac’. This gives us satisfactory results in most cases.

There are two popular stemmers:

- **Porter stemmer**: Developed in 1980 and works only on English words. All the detailed rules of this stemmer here.
- **Snowball stemmer**: More versatile stemmer that not only works on English words but also on words of other languages such as French, German, Italian, Finnish, Russian, and many more languages. Learn more about this stemmer here.

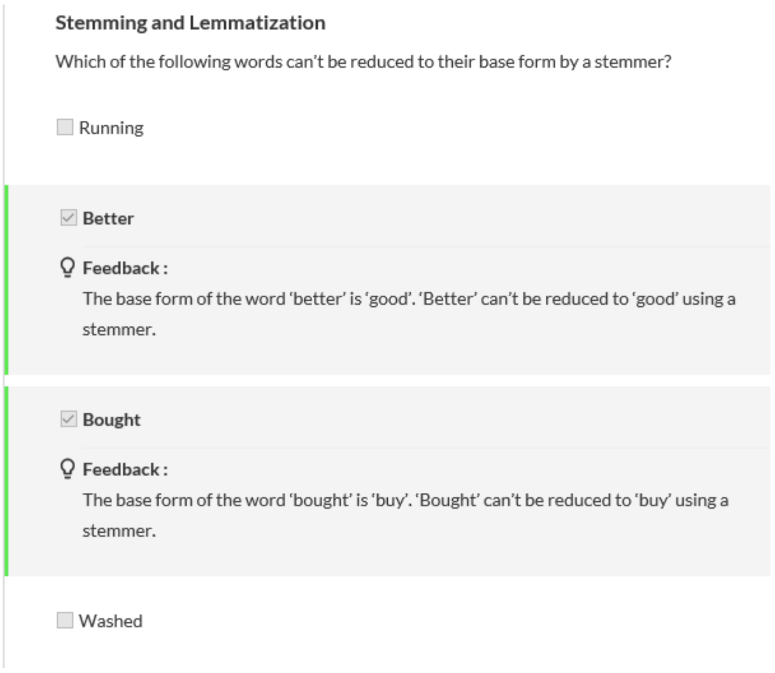
Lemmatization

A more sophisticated technique (more 'intelligent') that doesn't chop off the suffix of a word. Instead, it takes an input word and searches for its base word by going recursively through all the variations of dictionary words. The base word in this case is called the **lemma**. Words such as 'feet', 'drove', 'arose', 'bought', etc. can't be reduced to their correct base form using a stemmer. But a lemmatizer can reduce them to their correct base form. The most popular lemmatizer is the **WordNet lemmatizer** created by a team of researchers at the Princeton university. More about it here.

The following points might help make the decision between a stemmer or a lemmatizer in your application:

1. A stemmer is a rule based technique, and hence, it is much faster than the lemmatizer (which searches the dictionary to look for the lemma of a word). On the other hand, a stemmer typically gives less accurate results than a lemmatizer.
2. A lemmatizer is slower because of the dictionary lookup but gives better results than a stemmer. Now, as a side note, it is important to know that for a lemmatizer to perform accurately, provide the **part-of-speech tag** of the input word (noun, verb, adjective etc.). POS tagging in the next session - but it would suffice to know that there are often cases when the POS tagger itself is quite inaccurate, and that will worsen the performance of the lemmatiser as well. In short, consider a stemmer rather than a lemmatiser if POS tagging is inaccurate.

In general, try both and see if its worth using a lemmatizer over a stemmer. If a stemmer is giving almost same results with increased efficiency than choose a stemmer, otherwise use a lemmatizer.



Stemming and Lemmatization

Which of the following words can't be reduced to their base form by a stemmer?

☐ Running

☑ **Better**

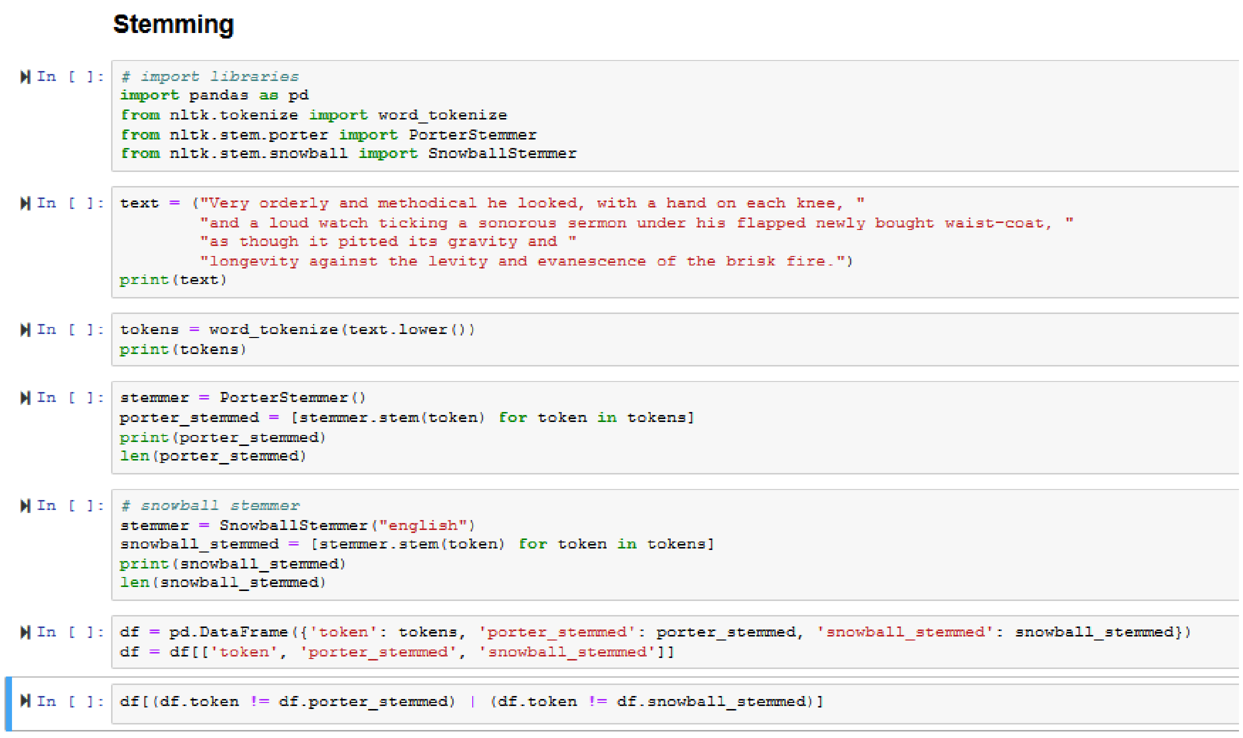
Feedback :
The base form of the word 'better' is 'good'. 'Better' can't be reduced to 'good' using a stemmer.

☑ **Bought**

Feedback :
The base form of the word 'bought' is 'buy'. 'Bought' can't be reduced to 'buy' using a stemmer.

☐ Washed

Now, let's see how to use a stemmer on a text corpus in Python.



Stemming

```
In [ ]: # import libraries
        import pandas as pd
        from nltk.tokenize import word_tokenize
        from nltk.stem.porter import PorterStemmer
        from nltk.stem.snowball import SnowballStemmer
```

```
In [ ]: text = ("Very orderly and methodical he looked, with a hand on each knee, "
                "and a loud watch ticking a sonorous sermon under his flapped newly bought waist-coat, "
                "as though it pitted its gravity and "
                "longevity against the levity and evanescence of the brisk fire.")
        print(text)
```

```
In [ ]: tokens = word_tokenize(text.lower())
        print(tokens)
```

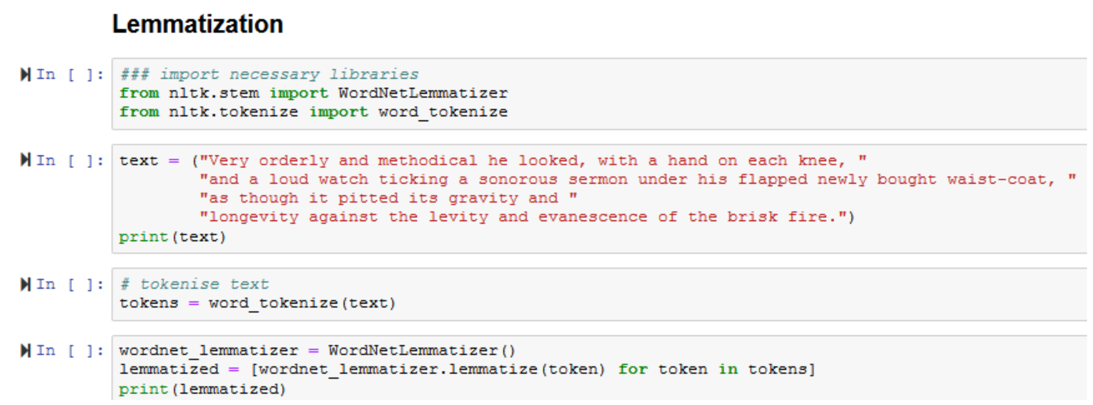
```
In [ ]: stemmer = PorterStemmer()
        porter_stemmed = [stemmer.stem(token) for token in tokens]
        print(porter_stemmed)
        len(porter_stemmed)
```

```
In [ ]: # snowball stemmer
        stemmer = SnowballStemmer("english")
        snowball_stemmed = [stemmer.stem(token) for token in tokens]
        print(snowball_stemmed)
        len(snowball_stemmed)
```

```
In [ ]: df = pd.DataFrame({'token': tokens, 'porter_stemmed': porter_stemmed, 'snowball_stemmed': snowball_stemmed})
        df = df[['token', 'porter_stemmed', 'snowball_stemmed']]
```

```
In [ ]: df[(df.token != df.porter_stemmed) | (df.token != df.snowball_stemmed)]
```

Two types of stemmers - the Porter stemmer and the Snowball stemmer. Snowball stemmer works a little better, but usually not much of a difference as both are rule based. Snowball has some updated rules and that's why it stems some words differently.



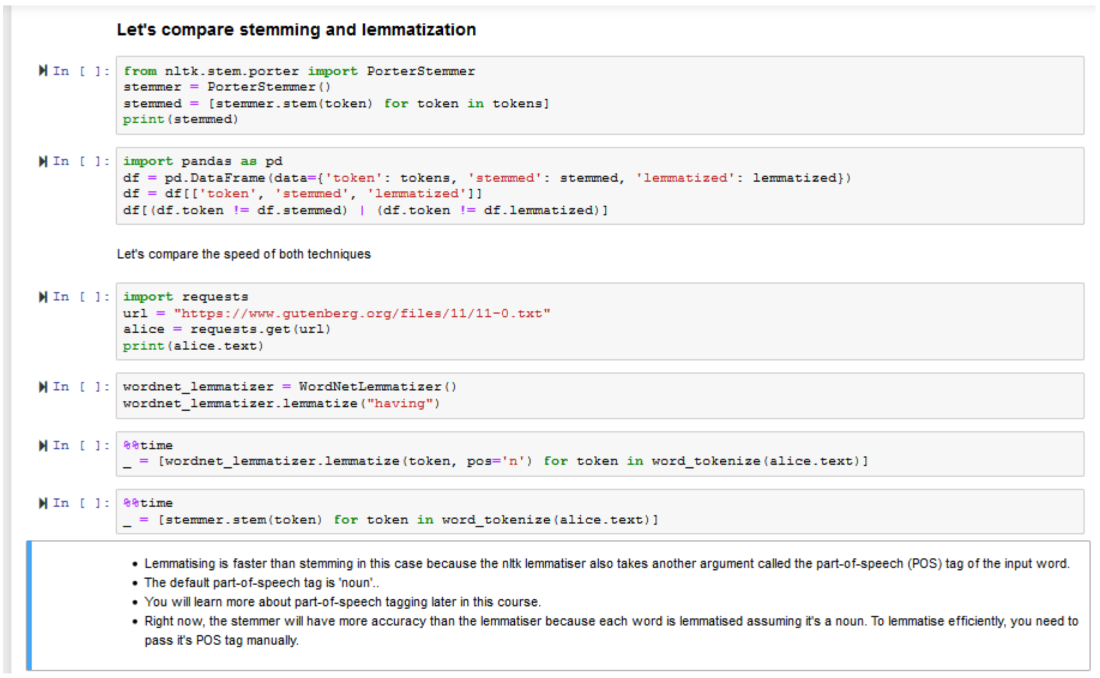
Lemmatization

```
In [ ]: ### import necessary libraries
        from nltk.stem import WordNetLemmatizer
        from nltk.tokenize import word_tokenize
```

```
In [ ]: text = ("Very orderly and methodical he looked, with a hand on each knee, "
                "and a loud watch ticking a sonorous sermon under his flapped newly bought waist-coat, "
                "as though it pitted its gravity and "
                "longevity against the levity and evanescence of the brisk fire.")
        print(text)
```

```
In [ ]: # tokenise text
        tokens = word_tokenize(text)
```

```
In [ ]: wordnet_lemmatizer = WordNetLemmatizer()
        lemmatized = [wordnet_lemmatizer.lemmatize(token) for token in tokens]
        print(lemmatized)
```


Let's compare stemming and lemmatization

```
In [ ]: from nltk.stem.porter import PorterStemmer
        stemmer = PorterStemmer()
        stemmed = [stemmer.stem(token) for token in tokens]
        print(stemmed)
```

```
In [ ]: import pandas as pd
        df = pd.DataFrame(data={'token': tokens, 'stemmed': stemmed, 'lemmatized': lemmatized})
        df = df[['token', 'stemmed', 'lemmatized']]
        df[(df.token != df.stemmed) | (df.token != df.lemmatized)]
```

Let's compare the speed of both techniques

```
In [ ]: import requests
        url = "https://www.gutenberg.org/files/11/11-0.txt"
        alice = requests.get(url)
        print(alice.text)
```

```
In [ ]: wordnet_lemmatizer = WordNetLemmatizer()
        wordnet_lemmatizer.lemmatize("having")
```

```
In [ ]: %%time
        _ = [wordnet_lemmatizer.lemmatize(token, pos='n') for token in word_tokenize(alice.text)]
```

```
In [ ]: %%time
        _ = [stemmer.stem(token) for token in word_tokenize(alice.text)]
```

- Lemmatising is faster than stemming in this case because the nltk lemmatiser also takes another argument called the part-of-speech (POS) tag of the input word.
- The default part-of-speech tag is 'noun'..
- You will learn more about part-of-speech tagging later in this course.
- Right now, the stemmer will have more accuracy than the lemmatiser because each word is lemmatised assuming it's a noun. To lemmatise efficiently, you need to pass it's POS tag manually.

CLARIFICATION: In this case, lemmatization was faster than stemming. That's since we didn't pass the part-of-speech tag with each word and so lemmatization happened quickly, but incorrectly. Had we passed the POS tag for each word, lemmatization would have had much more accuracy than stemming, but it would have also taken a lot of time.

How to find the POS tag of a word in the second module. Then, you'll be able to pass each word's POS tag along with it to lemmatize it correctly.

Final Bag-of-Words Representation [6/11]

Quite a few techniques in lexical pre-processing, namely:

1. Plotting word frequencies and removing stopwords
2. Tokenisation
3. Stemming
4. Lemmatization

Let's create the bag-of-words model, again, but this time, using stemming and lemmatization along with the other pre-processing steps. It will result in reducing the number of features by eliminating redundant features that we had created earlier. But more importantly, will lead to a more efficient representation.

```
import pandas as pd
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
from sklearn.feature_extraction.text import CountVectorizer

pd.set_option('max_colwidth', 100)

def preprocess(document):
    'changes document to lower case and removes stopwords'
    document = document.lower()
    words = word_tokenize(document)
    words = [word for word in words if word not in stopwords.words("english")]
    document = " ".join(words)
    return document

vectorizer = CountVectorizer()
bow_model = vectorizer.fit_transform(documents)
print(bow_model)  # returns the row number and column number of the cells which have 1 as value

# print the full sparse matrix
print(bow_model.toarray())

print(bow_model.shape)
print(vectorizer.get_feature_names())

messages = [preprocess(message) for message in messages]

from nltk.stem.porter import PorterStemmer
from nltk.stem import WordNetLemmatizer

stemmer = PorterStemmer()
wordnet_lemmatizer = WordNetLemmatizer()

# add stemming and lemmatisation in the preprocess function
def preprocess(document, stem=True):
    'changes document to lower case and removes stopwords'
    document = document.lower()
    words = word_tokenize(document)
    words = [word for word in words if word not in stopwords.words("english")]

    if stem:
        words = [stemmer.stem(word) for word in words]
    else:
        words = [wordnet_lemmatizer.lemmatize(word, pos='v') for word in words]

    document = " ".join(words)
    return document

messages = [preprocess(message, stem=True) for message in spam.message]
pd.DataFrame(bow_model.toarray(), columns = vectorizer.get_feature_names())

messages = [preprocess(message, stem=False) for message in spam.message]
```

How stemming and lemmatization performed on the spam dataset. Lemmatization didn't perform as good as it should have because of two reasons:

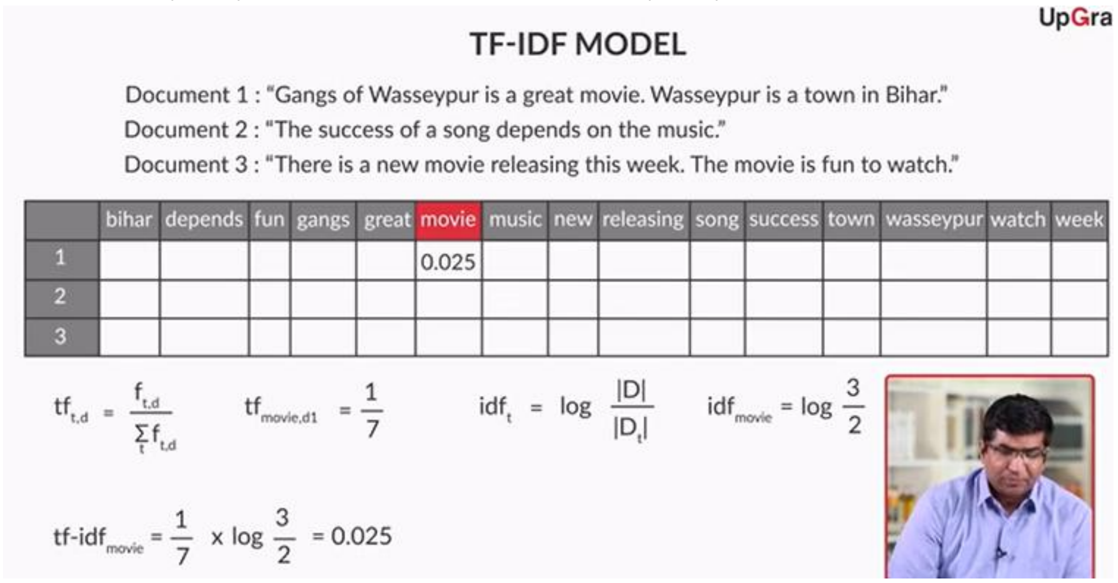
1. Lemmatization expected the POS tag of the word to be passed along with the word. We didn't pass the POS tag here. Next module covers how to assign POS tags.
2. Lemmatization only works on correctly spelt words. Since there are a lot of misspelt words in the dataset, lemmatization makes no changes to them.

In other words, the comparison of stemming and lemmatization wasn't fair. Comparison can be redone with tagging of each word with its POS tag. That will automate the process of lemmatization by passing the word along with its POS tag. It will be fair to compare the process of stemming and lemmatization only then.

TF-IDF Representation [7/11]

The bag of words representation, while effective, is a very naive way of representing text. It relies on just the word frequencies of the words of a document. But word representation shouldn't solely rely on the word frequency. TF-IDF representation represent documents in a matrix format in a smarter way. It is the one that is often preferred by most data scientists.

TF is term frequency, and IDF is inverse document frequency.



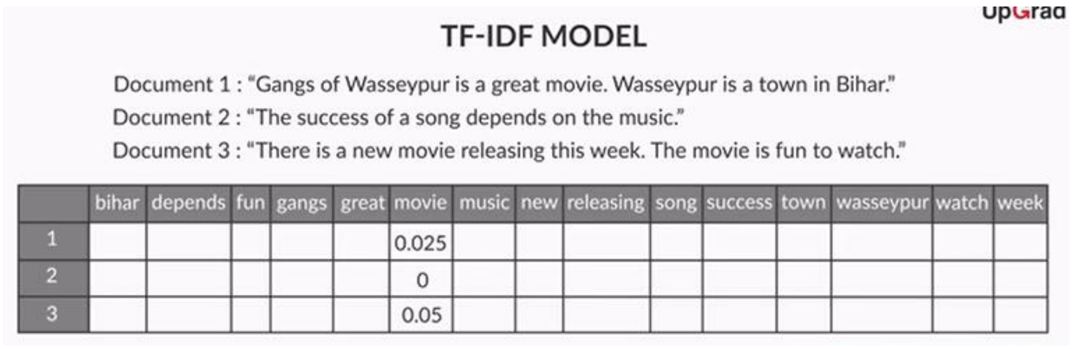
TF-IDF MODEL

Document 1 : "Gangs of Wasseypur is a great movie. Wasseypur is a town in Bihar."
Document 2 : "The success of a song depends on the music."
Document 3 : "There is a new movie releasing this week. The movie is fun to watch."

	bihar	depends	fun	gangs	great	movie	music	new	releasing	song	success	town	wasseypur	watch	week
1						0.025									
2															
3															

$$tf_{t,d} = \frac{f_{t,d}}{\sum_t f_{t,d}} \qquad tf_{movie,d1} = \frac{1}{7} \qquad idf_t = \log\frac{|D|}{|D_t|} \qquad idf_{movie} = \log\frac{3}{2}$$

$$\text{tf-idf}_{movie} = \frac{1}{7} \times \log\frac{3}{2} = 0.025$$



TF-IDF MODEL

Document 1 : "Gangs of Wasseypur is a great movie. Wasseypur is a town in Bihar."
Document 2 : "The success of a song depends on the music."
Document 3 : "There is a new movie releasing this week. The movie is fun to watch."

	bihar	depends	fun	gangs	great	movie	music	new	releasing	song	success	town	wasseypur	watch	week
1						0.025									
2						0									
3						0.05									

The TF-IDF representation, also called the **TF-IDF model**, takes into the account the importance of each word. In the bag-of-words model, each word is assumed to be equally important, which is of course not correct.

The formula to calculate TF-IDF weight of a term in a document is:

$$tf_{t,d} = \frac{frequency\ of\ term\ 't'\ in\ document\ 'd'}{total\ terms\ in\ document\ 'd'}$$

$$idf_t = log\frac{total\ number\ of\ documents}{total\ documents\ that\ have\ the\ term\ 't'}$$

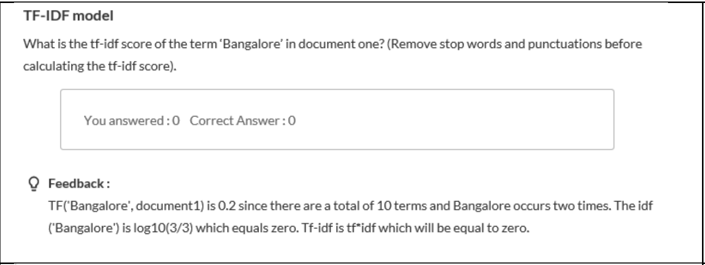
The log in the above formula is with base 10. tf-idf score for any term in a document is just the product of these two terms:

$$tf - idf = tf_{t,d} * idf_t$$

Higher weights are assigned to terms that are present frequently in a document and which are rare among all documents. On the other hand, a low score is assigned to terms which are common across all documents.

Now, attempt the following quiz. Questions 1-3 are based on the following set of documents:
Document1: "Vapour, Bangalore has a really great terrace seating and an awesome view of the Bangalore skyline"
Document2: "The beer at Vapour, Bangalore was amazing. My favourites are the wheat beer and the ale beer."
Document3: "Vapour, Bangalore has the best view in Bangalore."

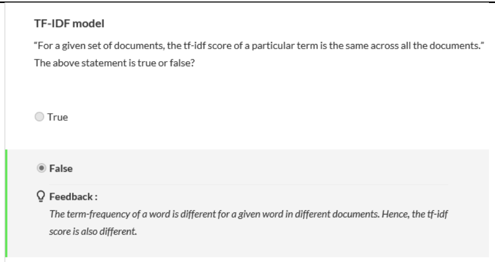
Vapour	Bangalore	really	great	terrace	seating	awesome	view	Bangalore	skyline
beer	Vapour	Bangalore	amazing	favourites	wheat	beer	ale	beer	
Vapour	Bangalore	best	view	Bangalore					


TF-IDF model

What is the tf-idf score of the term 'Bangalore' in document one? (Remove stop words and punctuations before calculating the tf-idf score).

You answered : 0 Correct Answer : 0

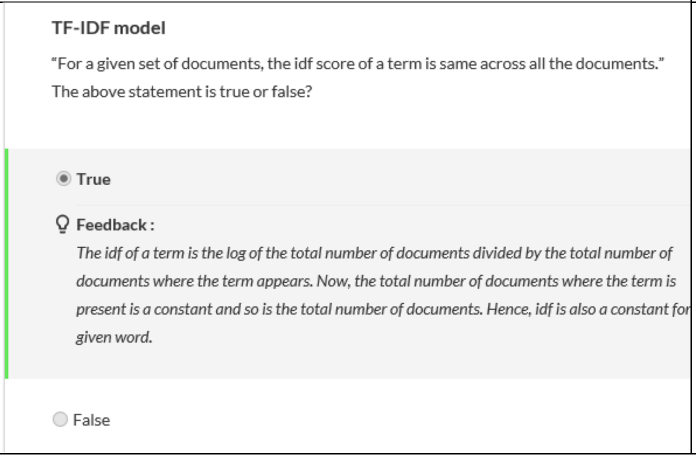
Feedback :
TF('Bangalore', document1) is 0.2 since there are a total of 10 terms and Bangalore occurs two times. The idf ('Bangalore') is log10(3/3) which equals zero. Tf-idf is tf*idf which will be equal to zero.


TF-IDF model

"For a given set of documents, the tf-idf score of a particular term is the same across all the documents."
The above statement is true or false?

True [✓ False]

Feedback :
The term-frequency of a word is different for a given word in different documents. Hence, the tf-idf score is also different.


TF-IDF model

"For a given set of documents, the idf score of a term is same across all the documents."
The above statement is true or false?

[✓ True]

Feedback :
The idf of a term is the log of the total number of documents divided by the total number of documents where the term appears. Now, the total number of documents where the term is present is a constant and so is the total number of documents. Hence, idf is also a constant for given word.

False

	Bangalore	Beer	Vapour
TF1	0.2	0	0.1
TF2	0.1111111	0.33333	0.1111111
TF3	0.4	0	0.2
IDF	0	0.47712	0
TF1-IDF	0	0	0
TF2-IDF	0	0.15904	0
TF3-IDF	0	0	0

tf-idf is implemented in different ways in different languages and packages. In the **tf score** representation, some people use only the frequency of the term, i.e. they don't divide the frequency of the term with the total number of terms. In the **idf score** representation, some people use natural log instead of the log with base 10. Due to this, you may see a different score of the same terms in the same set of documents. But the goal remains the same - assign a weight according to the word's importance.

```python
# add stemming and lemmatisation in the preprocess function
def preprocess(document):
    'changes document to lower case and removes stopwords'

    # change sentence to lower case
    document = document.lower()

    # tokenize into words
    words = word_tokenize(document)

    # remove stop words
    words = [word for word in words if word not in stopwords.words("english")]

    # stem
    #words = [stemmer.stem(word) for word in words]

    # join words to make sentence
    document = " ".join(words)

    return document
```

Creating bag of words model using count vectorizer function

```
In [ ]: vectorizer = TfidfVectorizer()
        tfidf_model = vectorizer.fit_transform(documents)
        print(tfidf_model)  # returns the row number and column number of cells which have 1 as value
```

```
In [ ]: # print the full sparse matrix
        print(tfidf_model.toarray())
```

```
In [ ]: pd.DataFrame(tfidf_model.toarray(), columns = vectorizer.get_feature_names())
```

Building a Spam Detector - I [8/11]

Apply pre-processing steps to build a spam detector.

Until now, we used the scikit-learn library to train ML algorithms. Now, build a spam detector using NLTK library, is go-to tool when working with text.

It is not necessary to learn how to use NLTK's ML functions. But it's always nice to have knowledge of more than one tool. How to extract features from raw text without using the scikit-learn package.

Building a Spam Detector

You saw that Krishna removed all the words from the corpus that are less than or equal to two characters long. Can you think of a reason why this was done?

Suggested Answer

Generally, words that are less than a certain length are removed from the corpus if it is suspected that they don't provide useful features, that is, they are considered as noise.

Get the messages and preprocess them using the preprocess function. Eliminate all the words which are less than or equal to two characters long.

Words less than a certain threshold are removed to eliminate special characters such as double exclamation marks, or double dots (the period character). And no information loss by doing this because there are no words less than two characters other than some stopwords (such as 'am', 'is', etc.).

Already learnt how to create a bag-of-words model by using the NLTK's CountVectorizer function. Now build a bag-of-words model without using the NLTK function, that is, building the model manually. The first step towards achieving that goal is to create a vocabulary from the text corpus.

Building a Spam Detector - II [9/11]

After creating the vocabulary, the next step is to create the matrix from the features (the bag-of-words model) and then train a machine learning algorithm on it. The algorithm that we're going to use is the Naive Bayes classifier.

Building a Spam Detector

Why do you think that Naive Bayes is a good choice when it comes to text classification problems such as spam detection?

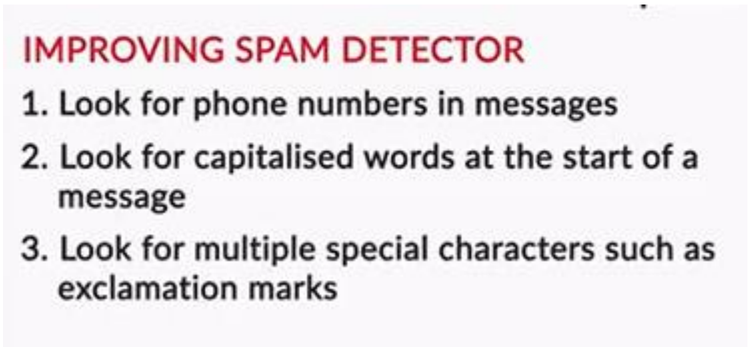
Suggested Answer

Naive Bayes assumes independence between features. Now, in text classification such as spam detection, only the presence of certain words matter. It doesn't matter if they occur before or after certain words. Hence, Naive Bayes often performs very well on these problems.

We've got an excellent accuracy of 98% on the test set. This excellent accuracy can be further improved by trying other models.

We created a bag-of-words representation that's created from scratch without using the CountVectorizer() function. We used a binary representation instead of using the number of features to represent each word. In this bag-of-words table, '1' means the word is present whereas '0' means the absence of that word in that document. You can do this by setting the 'binary' parameter to 'True' in the CountVectorizer() function.

Used the pickle library to save the model. After creating models, they are saved using the pickle library on the disk. This way, models can be used on a different computer or platform.



IMPROVING SPAM DETECTOR

1. Look for phone numbers in messages
2. Look for capitalised words at the start of a message
3. Look for multiple special characters such as exclamation marks

To get excellent results, take extra care of the nuances of the dataset. Understand the data inside-out to take these steps because these can't be generalized to every text classifier or even other spam datasets.

Summary [10/11]

A lot of essential pre-processing steps that need to apply when working with a corpus of text.

word frequencies and how to plot word frequencies on a given piece of corpus.

stop words are words that add no information in applications such as the spam detector. How to remove English stopwords using the NLTK's list of stopwords.

A document can be **tokenising** based on word, sentences, paragraphs, or even using own custom regular expression.

Importance of removing redundant words from the corpus by using two techniques - **stemming** and **lemmatization**. Stemming converts a word to its root from by chopping off its suffix. Lemmatization reduces a word to its base form, called the lemma, by going through the WordNet library.

We created a model from the text corpus that could be used to train a classifier, called the **bag-of-words model**. On similar lines, we used more advanced **tf-idf model**, which is a more robust representation of the text than the bag-of-words model.

Creating the **spam detector** uses all the pre-processing steps. Used a different library than the scikit-learn library to build the spam classifier - the NLTK library.