**Language models**

When Alan Turing proposed his test for intelligence, he based it on language, not art or haberdashery, perhaps because of its universal scope and because language captures so much of intelligent behavior: a speaker (or writer) has the goal of communicating some knowledge, then plans some language that represents the knowledge, and acts to achieve the goal. The listener (or reader) perceives the language, and infers the intended meaning. This type of communication via language has allowed civilization to grow; it is our main means of passing along cultural, legal, scientific, and technological knowledge. There are three primary reasons for computers to do natural language processing (NLP):

* To **communicate** with humans. In many situations it is convenient for humans to use speech to interact with computers, and in most situations it is more convenient to use natural language rather than a formal language such as first-order predicate calculus.
* To **learn**. Humans have written down a lot of knowledge using natural language. Wikipedia alone has 30 million pages of facts such as “Bush babies are small nocturnal primates,” whereas there are hardly any sources of facts like this written in formal logic. If we want our system to know a lot, it had better understand natural language.
* To advance the **scientific understanding** of languages and language use, using the tools of AI in conjunction with linguistics, cognitive psychology, and neuroscience.

Formal languages, such as first-order logic, are precisely defined. A grammar defines the syntax of legal sentences and semantic rules define the meaning. If we can’t make a definitive Boolean distinction between grammatical and ungrammatical strings, we can at least say how likely or unlikely each one is.

We define a language model as a probability distribution describing the likelihood of any string. Such a model should say that “Do I dare disturb the universe?” has a reasonable probability as a string of English, but “Universe dare the I disturb do?” is extremely unlikely.

With a language model, we can predict what words are likely to come next in a text, and thereby suggest completions for an email or text message. We can compute which alterations to a text would make it more probable, and thereby suggest spelling or grammar corrections. With a pair of models, we can compute the most probable translation of a sentence. With some example question/answer pairs as training data, we can compute the most likely answer to a question. So language models are at the heart of a broad range of natural language tasks. The language modeling task itself also serves as a common benchmark to measure progress in language understanding.

**Natural language processing**

Natural Language Processing (NLP) is a field of Artificial Intelligence that gives the machines the ability to read, understand and derive meaning from human languages. It is a discipline that focuses on the interaction between data science and human language, and is scaling to lots of industries. While natural language processing isn’t a new science, the technology is rapidly advancing thanks to an increased interest in human-to-machine communications, plus an availability of big data, powerful computing and enhanced algorithms. While supervised and unsupervised learning, and specifically deep learning, are now widely used for modeling human language, there’s also a need for syntactic and semantic understanding and domain expertise that are not necessarily present in these machine learning approaches. NLP is important because it helps resolve ambiguity in language and adds useful numeric structure to the data for many downstream applications, such as speech recognition or text analytics.

Natural language processing includes many different techniques for interpreting human language, ranging from statistical and machine learning methods to rules-based and algorithmic approaches. We need a broad array of approaches because the text- and voice-based data varies widely, as do the practical applications. Basic NLP tasks include tokenization and parsing, lemmatization/stemming, part-of-speech tagging, language detection and identification of semantic relationships. In general terms, NLP tasks break down language into shorter, elemental pieces, try to understand relationships between the pieces and explore how the pieces work together to create meaning.

These underlying tasks are often used in higher-level NLP capabilities, such as:

* Content categorization. A linguistic-based document summary, including search and indexing, content alerts and duplication detection.
* Topic discovery and modeling. Accurately capture the meaning and themes in text collections, and apply advanced analytics to text, like optimization and forecasting.
* Contextual extraction. Automatically pull structured information from text-based sources.
* Sentiment analysis. Identifying the mood or subjective opinions within large amounts of text, including average sentiment and opinion mining.
* Speech-to-text and text-to-speech conversion. Transforming voice commands into written text, and vice versa.
* Document summarization. Automatically generating synopses of large bodies of text.
* Machine translation.Automatic translation of text or speech from one language to another.

Natural language processing goes hand in hand with text analytics, which counts, groups and categorizes words to extract structure and meaning from large volumes of content. Text analytics is used to explore textual content and derive new variables from raw text that may be visualized, filtered, or used as inputs to predictive models or other statistical methods.

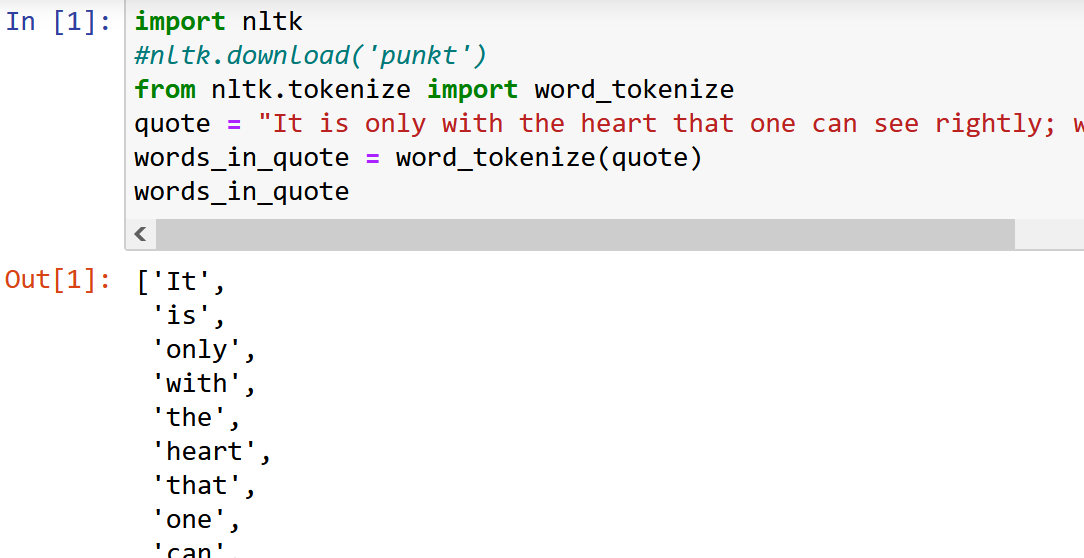
A lot of the data that we will be analyzing is unstructured data and contains human-readable text. Before that data can be analyzed programmatically, we need to preprocess it. We will use a Python package called Natural Language Toolkit (NLTK) to build NLP applications. You can find more information about NLTK at <http://www.nltk.org>.

**Tokenizing**

By tokenizing, you can conveniently split up text by word or by sentence. This will allow you to work with smaller pieces of text that are still relatively coherent and meaningful even outside of the context of the rest of the text. It’s your first step in turning unstructured data into structured data, which is easier to analyze.

There are two types of tokenization:

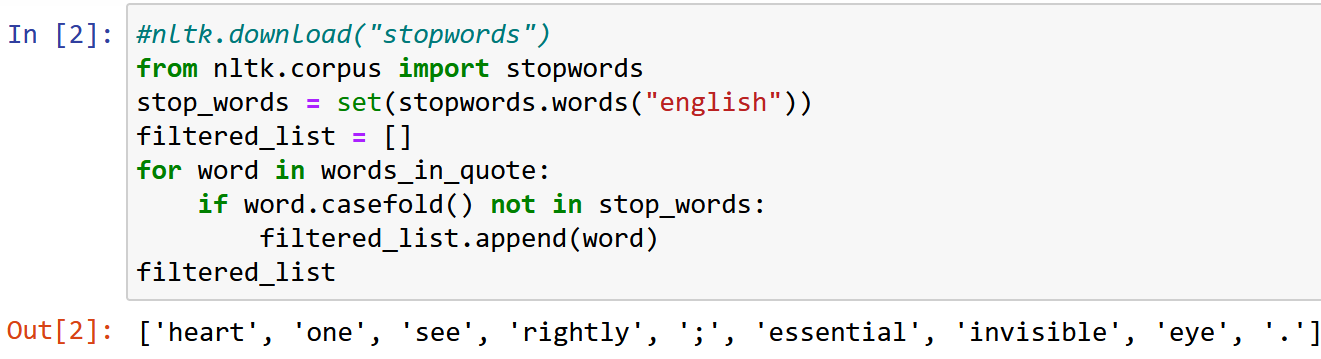
* Tokenizing by word. Tokenizing your text by word allows you to identify words that come up particularly often. For example, if you were analyzing a group of job ads, then you might find that the word “Python” comes up often. That could suggest high demand for Python knowledge, but you’d need to look deeper to know more.
* Tokenizing by sentence. When you tokenize by sentence, you can analyze how those words relate to one another and see more context. Are there a lot of negative words around the word “Python” because the hiring manager doesn’t like Python? Are there more terms from the domain of herpetology than the domain of software development, suggesting that you may be dealing with an entirely different kind of python than you were expecting?



**Filtering Stop Words**

Stop words are words that you want to ignore, so you filter them out of your text when you’re processing it. Very common words like 'in', 'is', and 'an' are often used as stop words since they don’t add a lot of meaning to a text in and of themselves.

**'not'** is technically an adverb but has still been included in NLTK’s list of stop words for English. If you want to edit the list of stop words to exclude 'not' or make other changes, then you can [download it](https://www.nltk.org/nltk_data/).

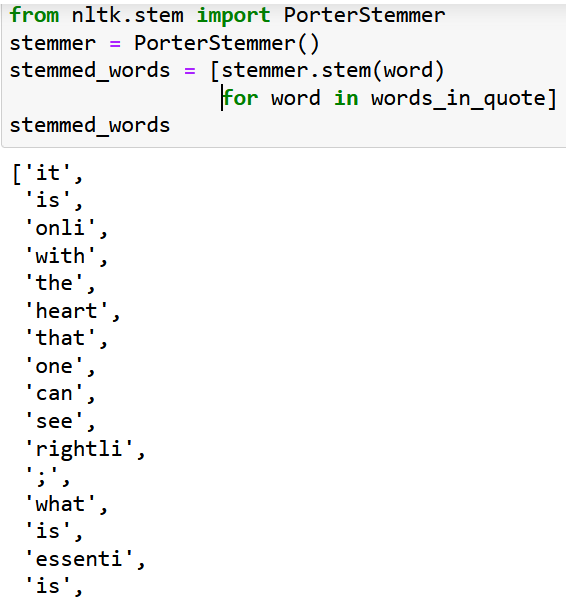


**Converting words to their base forms using stemming**

Working with text has a lot of variations included in it. We have to deal with different forms of the same word and enable the computer to understand that these different words have the same base form. For example, the word sing can appear in many forms such as sang, singer, singing, singer, and so on. We just saw a set of words with similar meanings. Humans can easily identify these base forms and derive context.

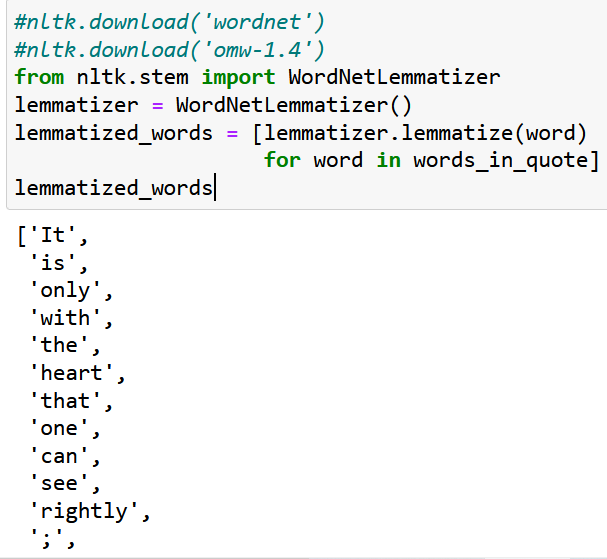
When we analyze text, it's useful to extract these base forms. It will enable us to extract useful statistics to analyze the input text. Stemming is one way to achieve this. The goal of a stemmer is to reduce words in their different forms into a common base form. It is basically a heuristic process that cuts off the ends of words to extract their base forms.

Understemming and overstemming are two ways stemming can go wrong:

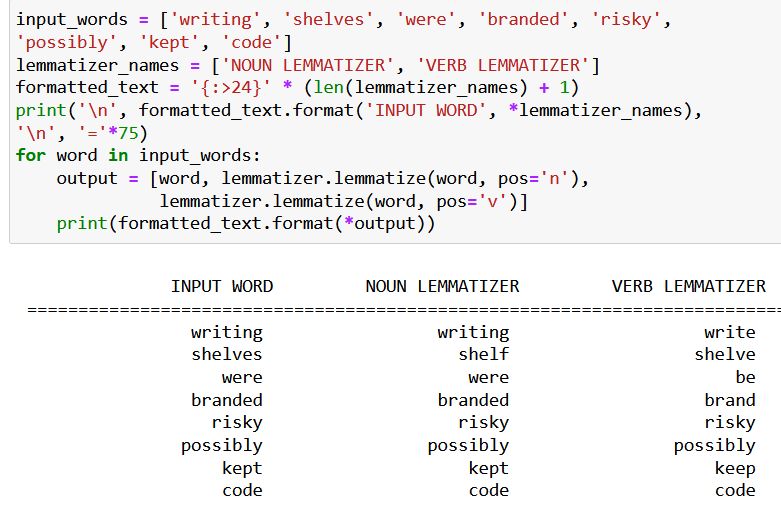
1. Understemming happens when two related words should be reduced to the same stem but aren’t. This is a false negative.
2. Overstemming happens when two unrelated words are reduced to the same stem even though they shouldn’t be. This is a false positive.

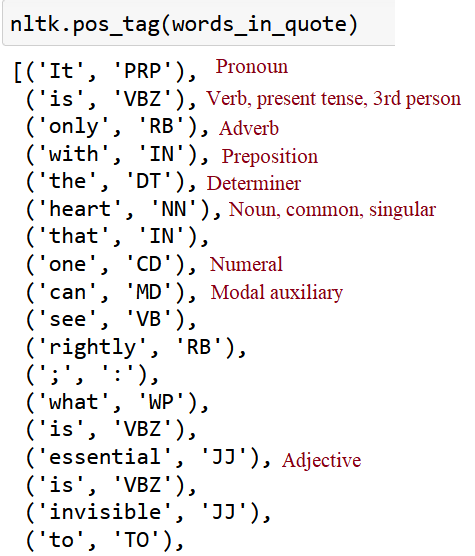
NLTK has [more than one stemmer](https://www.nltk.org/api/nltk.stem.html). All of them basically try to achieve the same goal. The difference between them is the level of strictness that's used to arrive at the base form. For example, the Porter stemmer is the least in terms of strictness and Lancaster is the strictest. If you closely observe the outputs, you will notice the differences. Stemmers behave differently when it comes to words like possibly or provision. The stemmed outputs that are obtained from the Lancaster stemmer are a bit obfuscated because it reduces the words a lot. At the same time, the algorithm is really fast. A good rule of thumb is to use the Snowball stemmer because it's a good trade off between speed and strictness.

**Converting words to their base forms using lemmatization**

Lemmatization is another way of reducing words to their base forms. Previously we saw that the base forms that were obtained from those stemmers didn't make sense. For example, all the three stemmers said that the base form of *calves* is *calv*, which is not a real word. Lemmatization takes a more structured approach to solve this problem.

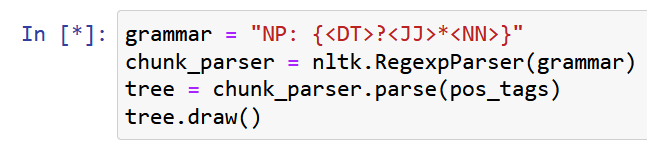
The lemmatization process uses a vocabulary and morphological analysis of words. It obtains the base forms by removing the inflectional word endings such as ing or ed. This base form of any word is known as the lemma. If you lemmatize the word calves, you should get calf as the output. One thing to note is that the output depends on whether the word is a verb or a noun.



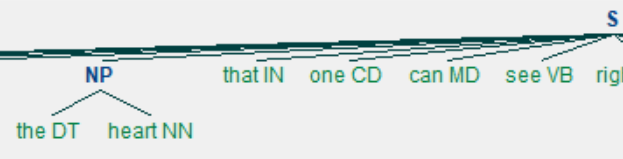
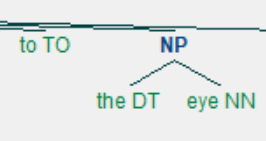
**Tagging Parts of Speech**

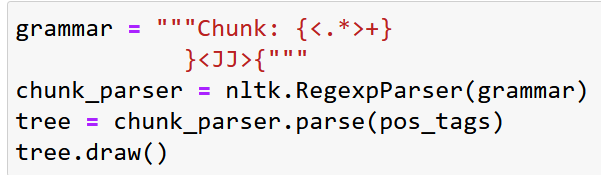
Part of speech is a grammatical term that deals with the roles words play when you use them together in sentences. Tagging parts of speech, or POS tagging (nltk.pos\_tag), is the task of labeling the words in your text according to their part of speech. In English, there are eight parts of speech: noun, pronoun, adjective, verb, adverb, preposition, conjunction, interjection (or exclamation). Some sources also include the category articles (like “a” or “the”) in the list of parts of speech, but other sources consider them to be adjectives. NLTK uses the word determiner to refer to articles.

**Dividing text data into chunks**

Text data usually needs to be divided into pieces for further analysis. This process is known as **chunking**. This is used frequently in text analysis. The conditions that are used to divide the text into chunks can vary based on the problem at hand. This is not the same as tokenization where we also divide text into pieces. During chunking, we do not adhere to any constraints and the output chunks need to be meaningful. When we deal with large text documents, it becomes important to divide the text into chunks to extract meaningful information.

In the example above a chunk grammar is created with one regular expression rule. NP stands for noun phrase. According to the rule, the chunks start with an optional (?) determiner ('DT'), can have any number (\*) of adjectives (JJ), end with a noun (<NN>). Two chunks were found: “the heart” and “the eye”.

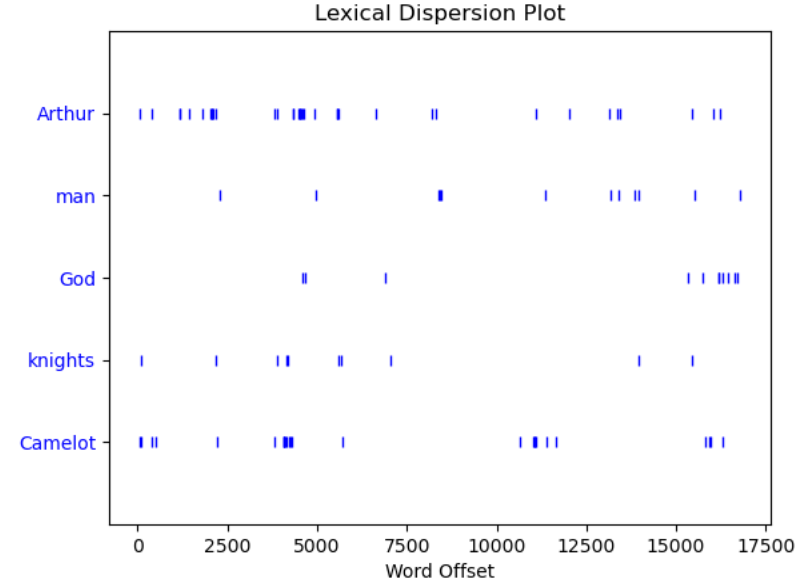
Chunking makes use of POS tags to group words and apply chunk tags to those groups. Chunks don’t overlap, so one instance of a word can be in only one chunk at a time. Before you can chunk, you need to make sure that the parts of speech in your text are tagged, so create a string for POS tagging. 

Chinking is used together with chunking, but while chunking is used to include a pattern, **chinking** is used to exclude a pattern.

We need to create a grammar to determine what we want to include and exclude in chunks. This time, there is more than one line because we have more than one rule. The first rule of the grammar is {<.\*>+}. This rule has curly braces that face inward ({}) because it’s used to determine what patterns we want to include in the chunks. In this case, we want to include everything: <.\*>+. The second rule of the grammar is }<JJ>{. This rule has curly braces that face outward (}{) because it’s used to determine what patterns we want to exclude in the chunks. In this case, we want to exclude adjectives: <JJ>. The tree shows that adjectives “essential” and “invisible” were excluded.

**Making a Dispersion Plot**

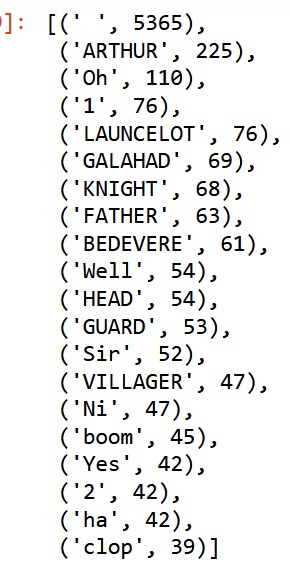
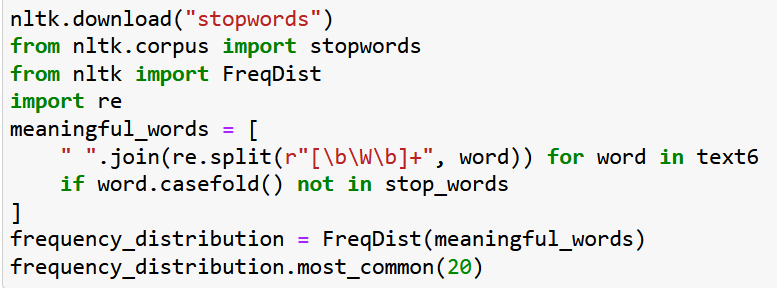
We can use a dispersion plot to see how much a particular word appears and where it appears. Each vertical blue line represents one instance of a word. Each horizontal row of blue lines represents the corpus as a whole. We use a dispersion plot when we want to see where words show up in a text or corpus. If we’re analyzing a single text, this can help us see which words show up near each other. If we’re analyzing a corpus of texts that is organized chronologically, it can help us see which words were being used more or less over a period of time. Dispersion plots are just one type of visualization we can make for textual data.

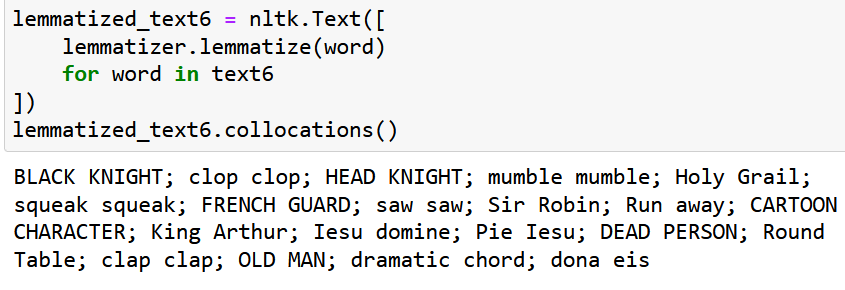


When you use a **concordance** (for example, use text6.concordance(“man”)), you can see each time a word is used, along with its immediate context. This can give you a peek into how a word is being used at the sentence level and what words are used with it.

**Making a Frequency Distribution**

With a frequency distribution, we can check which words show up most frequently in the text. We have a lot of punctuation and stop words (*a*, *the*,etc.) in the frequency distribution, so it is better to remove them using methods from *string* and *re* modules. The regular expression removes everything except whitespaces. The stop words that should be ignored are imported from *nltk*.



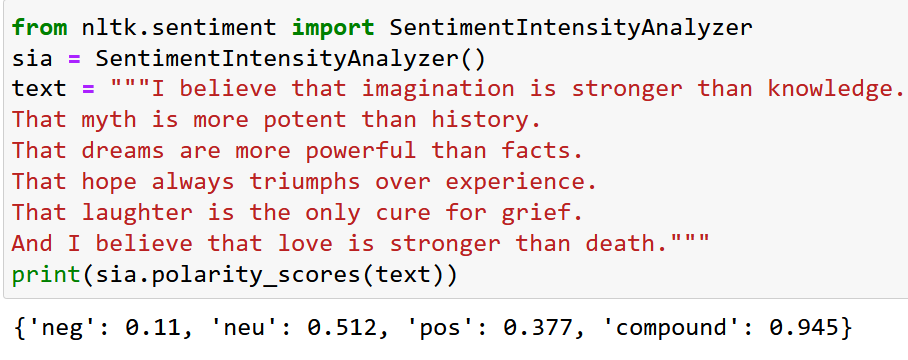
**Finding collocations**

A collocation is a sequence of words that shows up often. To see pairs of words that come up often in the corpus, we need to call *collocations()* on it. Lemmatizing the text before applying collocations() might yield more accurate results.

**Sentiment analysis**

Sentiment analysis is used to determine the ratio of positive to negative engagements about a specific topic. We can analyze bodies of text, such as comments, tweets, and product reviews, to obtain insights from the audience. So, basically, sentiment analysis is used to classify various samples of related text into overall positive and negative categories. With NLTK, we can employ these algorithms through powerful built-in machine learning operations to obtain insights from linguistic data.

NLTK already has a built-in, pretrained sentiment analyzer called VADER (Valence Aware Dictionary and sEntiment Reasoner). Since VADER is pretrained, we can get results more quickly than with many other analyzers. However, VADER is best suited for language used in social media, like short sentences with some slang and abbreviations. It’s less accurate when rating longer, structured sentences.

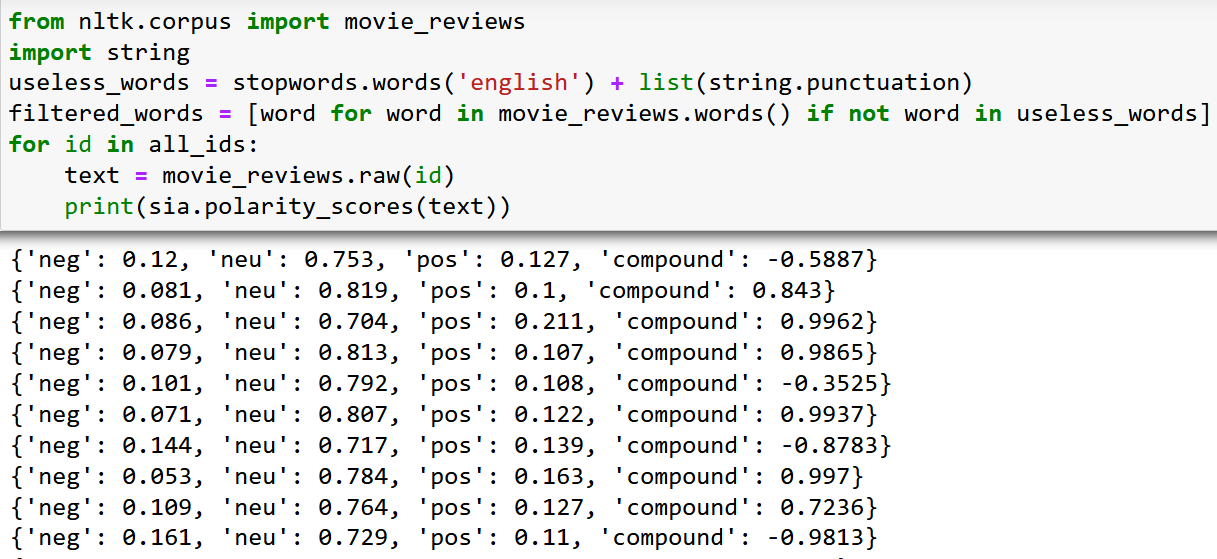


To use VADER, first create an instance of SentimentIntensityAnalyzer, then use .polarity\_scores() on a raw string. We get back a dictionary of different scores. The negative, neutral, and positive scores are related: they all add up to 1 and can’t be negative. The compound score is calculated differently.

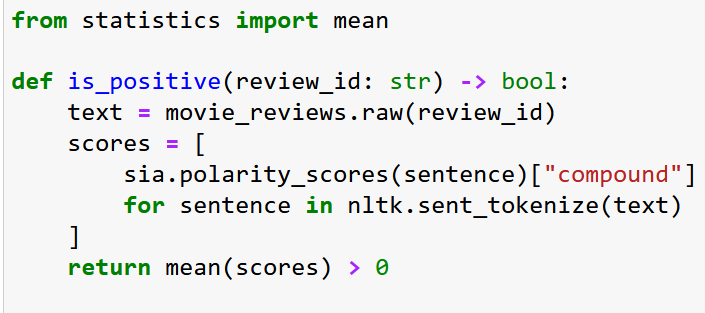
The compound score is computed by summing the valence scores of each word in the lexicon, adjusted according to the rules, and then normalized to be between -1 (most extreme negative) and +1 (most extreme positive). The *pos*, *neu*, and *neg* scores are ratios for proportions of text that fall in each category*.* This is the most useful metric if we want a single unidimensional measure of sentiment for a given sentence. Calling it a 'normalized, weighted composite score' is accurate. It is also useful for setting standardized thresholds for classifying sentences as either positive, neutral, or negative. Typical threshold values are:

1. positive sentiment: compound score >= 0.05
2. neutral sentiment: (compound score > -0.05) and (compound score < 0.05)
3. negative sentiment: compound score <= -0.05

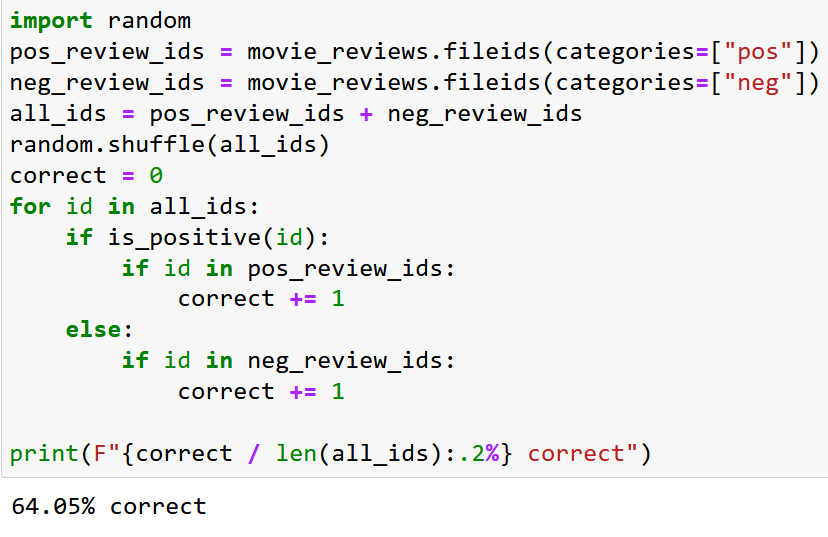
Let’s have a look at the corpus *movie\_reviews*, a collection of movie reviews. In the example below we remove stop words and punctuation (this time without regular expressions) and apply *polarity\_scores*(). Since VADER needs raw strings for its rating, we use method *raw*() instead of usual *words*().



The special thing about this corpus is that it’s already been classified. Therefore, we can use it to judge the accuracy of the algorithm when rating similar texts. For that, we define a method *is\_positive*() to obtain a specific review using its file ID and then split it into sentences before rating.



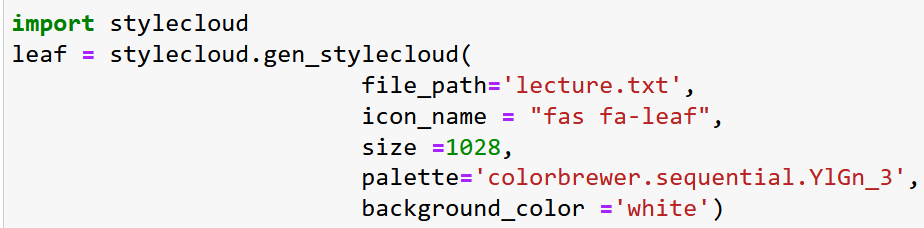
Also, we need to make a list of the file IDs that the corpus uses, which we use to reference individual reviews. *fileids*() exists in most, if not all, corpora. In the case of *movie\_reviews*, each file corresponds to a single review. We can also filter the list of file IDs by specifying categories. This categorization is a feature specific to this corpus and others of the same type.



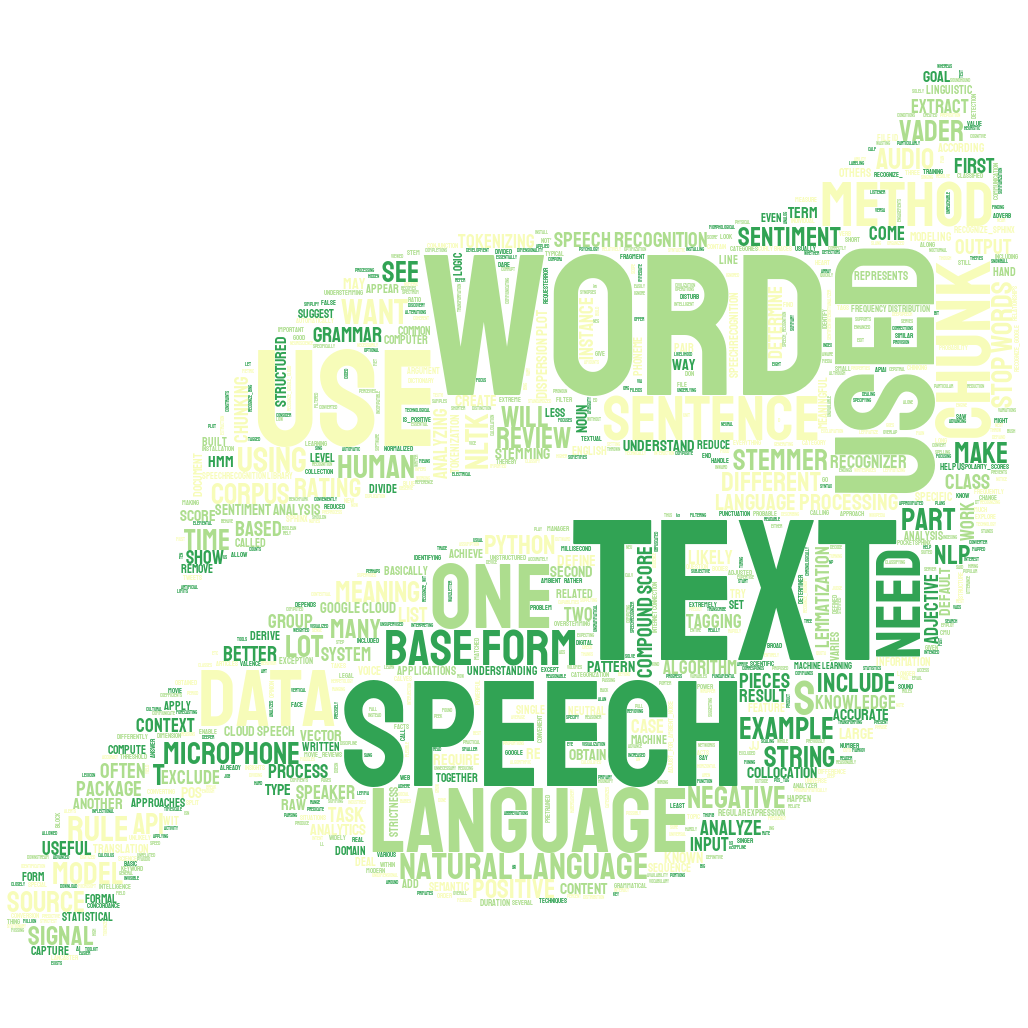
After rating all reviews, we can see that only 64 percent were correctly classified by VADER using the logic defined in *is\_positive*(). That is because VADER is better at rating tweets than it is at rating long movie reviews. One of the ways to get better results is to set up VADER to rate individual sentences within the review rather than the entire text like it was done in the example above.

**Keywords**

Wordcloud is a popular technique that helps us identify the keywords in a text. In a wordcloud, more frequent words have a larger and bolder font, while less frequent words have smaller or thinner fonts. In Python, we can make simple wordclouds with the *wordcloud* library and nice-looking wordclouds with the *stylecloud* library.



The resulting cloud is saved in .png file in the source directory:



**Speech recognition**

Speech must be converted from physical sound to an electrical signal with a microphone, and then to digital data with an analog-to-digital converter. Once digitized, several models can be used to transcribe the audio to text. Most modern speech recognition systems rely on what is known as a Hidden Markov Model (HMM). This approach works on the assumption that a speech signal, when viewed on a short enough timescale (for instance, ten milliseconds), can be reasonably approximated as a stationary process—that is, a process in which statistical properties do not change over time.

In a typical HMM, the speech signal is divided into 10-millisecond fragments. The power spectrum of each fragment, which is essentially a plot of the signal’s power as a function of frequency, is mapped to a vector of real numbers known as cepstral coefficients. The dimension of this vector is usually small — sometimes as low as 10, although more accurate systems may have dimension 32 or more. The final output of the HMM is a sequence of these vectors.

To decode the speech into text, groups of vectors are matched to one or more phonemes — a fundamental unit of speech. This calculation requires training, since the sound of a phoneme varies from speaker to speaker, and even varies from one utterance to another by the same speaker. A special algorithm is then applied to determine the most likely word (or words) that produce the given sequence of phonemes.

In many modern speech recognition systems, neural networks are used to simplify the speech signal using techniques for feature transformation and dimensionality reduction before HMM recognition. Voice activity detectors (VADs) are also used to reduce an audio signal to only the portions that are likely to contain speech. This prevents the recognizer from wasting time analyzing unnecessary parts of the signal.

Python speech recognition packages: apiai, pocketsphinx, wit, DeepSpeech, google-cloud-speech, SpeechRecognition, wav2letter. Some of these packages—such as wit and apiai—offer built-in features, like natural language processing for identifying a speaker’s intent, which go beyond basic speech recognition. Others, like google-cloud-speech, focus solely on speech-to-text conversion.

We’ll work with the SpeechRecognition package. The SpeechRecognition library acts as a wrapper for several popular speech APIs and is thus extremely flexible. One of these—the Google Web Speech API—supports a default API key that is hard-coded into the SpeechRecognition library. The SpeechRecognition library has many classes but we will be focusing on a class called Recognizer. This is the class that will help us to convert audio files into text.

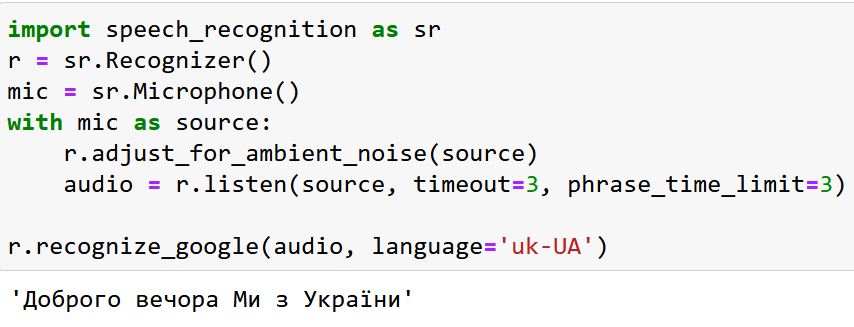
Each Recognizer instance has methods for recognizing speech from an audio source using various APIs:

* recognize\_bing(): Microsoft Bing Speech
* recognize\_google(): Google Web Speech API
* recognize\_google\_cloud(): Google Cloud Speech - requires installation of the google-cloud-speech package
* recognize\_houndify(): Houndify by SoundHound
* recognize\_sphinx(): CMU Sphinx - requires installing PocketSphinx
* recognize\_wit(): Wit.ai

Only recognize\_sphinx() works offline with the CMU Sphinx engine. Others require internet connection. Each recognize\_\*() method will throw a speech\_recognition.RequestError exception if the API is unreachable. For recognize\_sphinx(), this could happen as the result of a missing, corrupt or incompatible Sphinx installation. For the other methods, RequestError may be thrown if quota limits are met, the server is unavailable, or there is no internet connection.

To access the microphone with SpeechRecognizer, we have to install the PyAudio package. If the system has no default microphone, or we want to use a microphone other than the default, we need to specify which one to use by supplying a device index. We can get a list of microphone names by calling the list\_microphone\_names() static method of the Microphone class.

Microphone class is a context manager. We can capture input from the microphone using the listen() method of the Recognizer class inside of the with block. This method takes an audio source as its first argument and records input from the source until silence is detected. Then the input can be recognized.



To handle ambient noise, we need to use the adjust\_for\_ambient\_noise() method of the Recognizer class. adjust\_for\_ambient\_noise() analyzes the audio source for one second. If it’s too long, the duration keyword argument can be adjusted. The SpeechRecognition documentation recommends using a duration no less than 0.5 seconds. In some cases, durations longer than the default of one second generate better results. The minimum value depends on the microphone’s ambient environment.

To recognize speech in a different language, set the language keyword argument of the recognize\_\*() method to a corresponding string. In this case:

r.recognize\_google(audio, language='uk-UA')

Audio that cannot be matched to text by the API raises an UnknownValueError exception. It is better to wrap calls to the API with try and except blocks to handle this exception.

