MSAN
Summer 2017
Natural Language Processing

---

# Topic 04: Sentiment Analysis



## Sentiment Analysis

Sentiment analysis is the process of determining whether some language (usually text, sometimes speech) is positive, negative or neutral, deriving the attitude or opinion of a writer or speaker with respect to a subject (person, place, event).

For example, given a line from a movie review like:

> *Wonder Woman is beautiful, kindhearted, and buoyant in ways that make me eager to see it again.*

… a properly-trained sentiment analysis tool would predict that this sentence belies Roger Ebert's positive opinion. Combined with named entity recognition, this becomes a powerful tool in determining how a person or group of people feel about a thing. This type of analysis can have important value.

### Terms & Definitions

---

We will not use many unfamiliar terms for this unit, but you should become familiar with a few concepts. Sentiment analysis is also known as *opinion mining*, *the voice of the customer* and other names. We can apply this technique to any language — text, speech, tweets, etc. We generally use the term "speaker" or "writer" interchangeably, regardless of whether the language being produced is in the form of text or speech. Let's use the term "writer" consistently because most of the analysis we'll perform will be on text.

Sentiment analysis may be used to understand the attitude of the writer to the subject or the underlying affective (emotional?) state of the writer. With the understanding that the task looks at emotions, we should probably know a little about human emotions. More on this later.

## Examples

Given some straightforward language such as the (product) review, "`I love this camera`" we can find useful sentiment cues in the tokens used.

However, we can quickly get to less straightforward language in which the sentiment is less trivial to determine. Here are some examples:

- Negation: Even a simple sequence like "`I really did not like this movie`" causes trouble if not treated correctly. We quickly understand this as negative review, but it is close (4 characters in edit distance) to a very positive one, a version which we may see in training, with the opposite meaning.

- Comparison: Consider the following review: "`Well as usual Keanu Reeves is nothing special, but surprisingly the very talented Laurence Fishburne is not so good either. I was surprised`" ... which makes a comparison between the acting ability of the lead stars in a movie.

- Subtlety: It may be archaic to *damn with faint praise*, but a (movie) review like, "`This movie gave me some much-needed time to catch up with my email`" is extremely difficult to process. While the review may appear on the surface to be positive, it is scathingly the opposite. Even when the writer stays on topic, such as in "`I was very happy with its 88-minute running time`", detection of the subtle negativity is not trivial.

- … and many other phenomena, such as qualifications: Consider the sentence "`Most movies with Nicolas Cage are horrible`," or "`Roger Dodger is one of the least compelling variations on this theme`", slang: "`sentiment analysis is the shit`", etc.

These examples and more show how sentiment analysis can become complicated. Still, we can construct systems which are able to predict the sentiment of writing and speech. One of these systems is the Tweet Sentiment Visualization App. Let's have a look. What is your opinion of this tool?

## Psychology

In psychology, the study of emotions is its own area. There are two (competing) theories about which emotions exist. There are several examples of the atomic model of emotions. In one of the mode famous variants on this, Robert Plutchik produced what is now called Plutchik's Wheel — see Figure 1 — wherein there are eight (8) basic dimensions. The inner portion of the wheel represents "basic" emotions and the outer represent more "complex" emotions, formed by blending basic emotions together. There are several criticisms of this model, including that their formation represents the assumption that Western, Anglocentric cultural view applies universally to all people in the world. (Plutchik's Wheel does not account for schadenfreude, for example.)

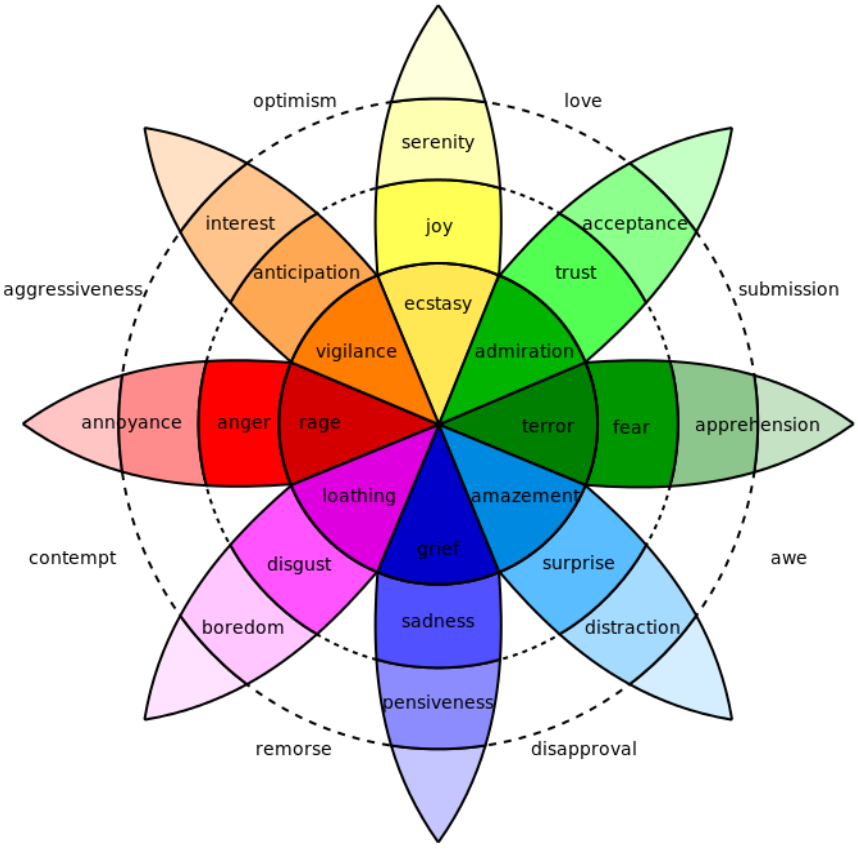


Figure 1: Plutchik's Wheel by Machine Elf 1735 - Own work, Public Domain, https://commons.wikimedia.org/w/index.php?curid=13285286

Other psychologists use different numbers of categories. Paul Ekman, for example, uses six (anger, disgust, fear, happiness, sadness, surprise) based on human facial reactions. His faces are famous and can be found with a simple search.

In contrast to these models which lend themselves easily to categories, a competing model posits that combinations of valence (pleasantness / unpleasantness) and arousal (activated / deactivated) lead to the emotions we may recognise. While we may think of these emotions as categories, they are actually continuous — i.e. lead themselves more to regression than to classification. Figure 2 shows how an orthogonal mixture of valence and arousal can lead to a number of emotions, including many of those which appear in Plutchik's Wheel.

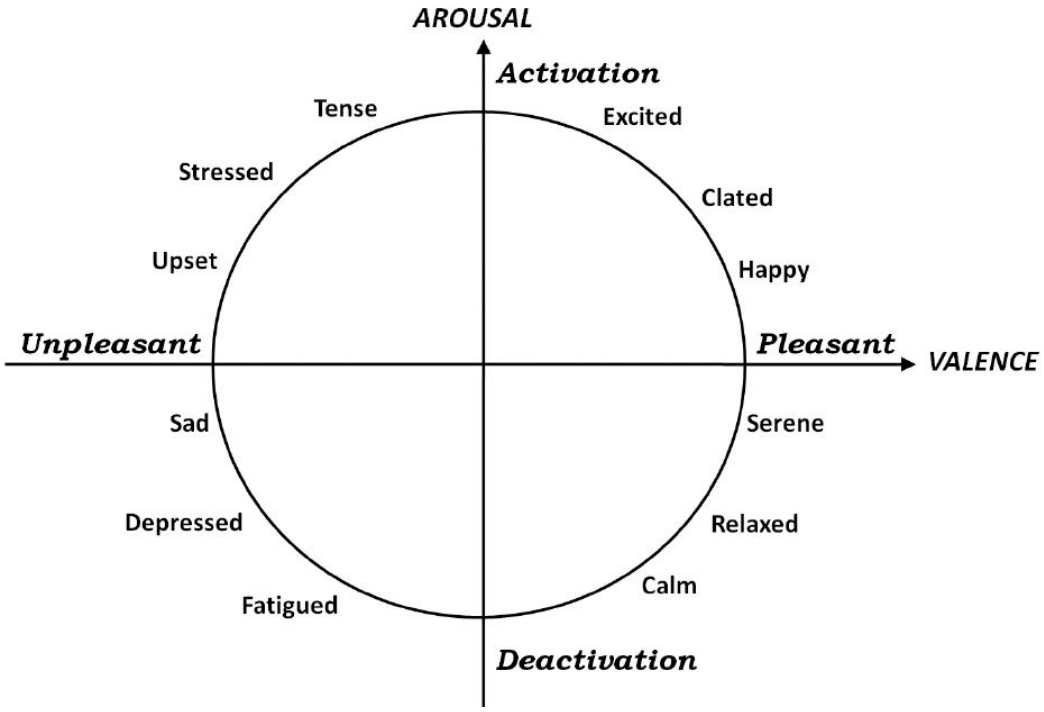


Figure 2: Valence/Arousal and resulting categorical emotions

Also useful are ideas of personality types. These were popular some years ago but may be less so now. We should feel free to take the test to determine our personality type and determine the implications of the type to which we are matched. These personality types have been suggested as having some influence in sentiment analysis and have been shown to affect other NLP tasks. James Pennebaker has shown that depression correlates with frequencies of certain function words. In practice, this finding implies that if we look at the rates of closed-class words (I, the, a, …) in text, we can determine something about the personality type or temporary state of the person who generated the text.

Generally, task of written sentiment analysis is seen as a classification between some imaginary catch-all positive sentiment and the opposite negative one. Strangely enough, this falls closer to

the valence / arousal model but is usually greatly simplified to poles of valence. However, some sentiment analysis tasks, especially those in speech, require that arousal be considered as an additional dimension. (Some speech tasks consider “cold anger” to be an emotion, such as when talking with a customer service representative by phone.) Under these models, discrete categories are often imposed on a regression. NLP work which uses one of the atomic models is increasing in popularity.

## Language Models

For many NLP tasks, including sentiment analysis, we use some kind of language modelling (LM), also known as a grammar. Intuitively, we probably already understand what language modelling is, but let’s be formal about it.

One of the goals — the main goal — of language modelling is the ability to assign a probability to a sequence of tokens, usually to determine the probability of an unseen event. In essence, we want to be able to compare hypothesised sequences like we see in Table 1 and use the sequence with higher probability.

| **Application** | **Sentence 1** | **Sequence 2** |
|---|---|---|
| Machine Translation (Proposed translation) | High winds tonight | Large winds tonight |
| Spelling Correction (Which replacement is more likely?) | … about fifteen minutes from | … about fifteen minuets from |
| Speech Recognition | I saw a van | Eyes awe of an |

Table 1: Example sequences for comparison by language models

### Basics of Language Models

In language modelling, we calculate the probability of a sequence of words (for example, in a sentence) the way we compute probability of any sequence. Given the sequence $W = w_1, w_2, w_3, w_4, w_5$ = “`colorless green ideas sleep furiously`,” we calculate the probability of this sentence (sequence) as: $p(W) = p(w_1, w_2, w_3, w_4, w_5)$. Or generally: $p(W) = p(w_1, \ldots, w_n)$. This is one use of the language model.

Let’s say we only had the first four words. What is the probability of the final word, “`furiously`”? It is determined by: $p(w_5 \mid w_1, w_2, w_3, w_4)$. This is another use of the language

model. We can use the chain rule of probability to determine the probability of a long sequence. Generally: $p(w_1, \ldots, w_n) = p(w_1)\, p(w_2|w_1)\, p(w_3|w_1w_2) \ldots p(w_n|w_1, \ldots, w_n)$
We never use long sequences in NLP because a lot of language is novel. Instead, we use a Markov assumption that the probability of a long sequence can be approximated by chaining several shorter sequences. In other words:

p("furiously" | "colorless green ideas sleep") ~ p("furiously" | "ideas sleep")

We can use more or fewer words in the history. If we use one word in the history (n), we call it a unigram model; bigram = 2 words; trigram = 3 words. Beyond 3, we usually refer to a number (4-gram, 5-gram, etc.)... but we almost never use n-gram models greater than 3 because it introduces too much sparseness in our model. It also does not help with the thing we really care about: long distance dependencies. As an example of a long-distance dependency, determine the correct variant of the verb "to be" which should be used in the following sentence:

The computers which I had in the server room on the 8th floor ___ crashing.

An n-gram grammar (language model) is related to a hypothesis by Noam Chomsky known as "poverty of the stimulus." The idea is that humans have a genetic predisposition to grammar. Evidence of this is that children can create grammatically correct novel sentences at a young age.

We can estimate the probability of a sequence using the formula:

$p(w_i|w_{i-1}) = \text{count}(w_{i-1}, w_i) / \text{count}(w_{i-1}) = c(w_{i-1}, w_i) / c(w_{i-1})$

Let's shore up this concept with an example. Given the sequence:

```
<s> I am Sam </s> <s> Sam I am </s> <s> I do not like green
eggs and ham </s>
```

$p(\texttt{I}|\texttt{<s>}) = ⅔ = 0.67$
$p(\texttt{</s>}|\texttt{Sam}) = ½ = 0.50$
$p(\texttt{do}|\texttt{I}) = ⅓ = 0.33$

The above is a unigram model. It is useful to keep a dictionary of word types and their counts, as in the partial sample shown in Table 2.

| am | and | eggs | ham | green | I |
|---|---|---|---|---|---|
| 2 | 1 | 1 | 1 | 1 | 3 |

Table 2: Unigram counts for "I am Sam…"

Some of the bigram counts are shown in Table 3.

Implementing a language model is reasonably trivial — see Katrin Erk's code for bigrams, for example. A lot of the code to implement code for language models makes choices about how to tokenize the text, how to represent the start of a document or sentence, etc. (The code in the

link does not use NLTK-type tokenization. For example, “it’s” is split into two tokens by NLTK, but the code in the link treats it as a single token.)

| | am | and | eggs | ham |
|---|---|---|---|---|
| am | 0 | 0 | 0 | 0 |
| and | 0 | 0 | 1 | 0 |
| eggs | 0 | 0 | 0 | 0 |
| ham | 0 | 1 | 0 | 0 |

Table 3: Bigram counts for “I am Sam…”

Generally (but not always), we want to normalise the unigram or bigram counts into probabilities. If we do, we use log space to avoid very small probabilities (“underflow”). Log space also has a computational advantage: once log probabilities have been calculated, we simply add them to determine (log) probabilities. Adding is computationally faster than multiplying.

## Language Model Practical

Language models are the basis of many things in NLP. There are two excellent recent efforts in NLP which would probably use LMs as one important source: the Fake News Challenge, which identifies fake news from real; there is also a Google Terrorist Propaganda project. While either would be a great project, the Fake News Challenge may be more appealing because it can be done without .

Like many language tools, NLTK contains a package for calculating n-gram counts. In the package nltk.util, we can use the ngrams function. Pass it any list of tokens and a degree for an n-gram and it returns a list (technically, a generator) for the n-grams of this list. If we combine this with collections.Counter, we can get a frequency-sorted list of bigrams.

Let’s try it for unigrams and bigrams on one of the files in our corpus, `cv109_21172.txt`. We have a sentiment-tagged corpus, the (IMDB) movie review data, which we will use later. Let’s use part of this to construct a language model. For now, we’ll make some simplifying assumptions and refine them later. Let’s do the following:

1) Download and expand the polarity dataset v2.0 from the movie review data.
2) Find the (positive) review example file (`imdb_review/txt_sentoken/pos/cv109_21172.txt`).
3) Read the file and use NLTK’s word_tokenize to get a list.
4) Get the bigrams of this list using NLTK’s ngrams with n=2.

5) Use collections.Counter on the bigrams to order and count bigrams

Let's use this code as a bigram reference or to catch up. (We can easily modify it for unigrams.)

In theory, the language model for a positive review (or sentiment) is different from the language model for a negative review. What can we say about this language model when compared against a negative review, for example `imdb_review/txt_sentoken/neg/cv067_21192.txt`?

## Smoothing

Unsurprisingly, language models fit their domains better than the fits outside the domain. In NLP we also refer to this as overfitting. One area for overfitting is the high number of 0 counts (and the associated probabilities). For example, our corpus has c(`president|impeach the`) = 0. (The nearest example we have may be "`impeachment for this`" in `cv230_7913.txt`.

Having a count (and therefore probability) of 0 is a problem. It is useful to assign a very low probability to an unseen event, but since probabilities must sum to 1, we have to remove probability mass from observed events in order to assign it to the unseen. This technique is called ***smoothing***.

There are a number of smoothing techniques. The easiest is called *Laplace* or *add-one estimation* smoothing. In it, we pretend we saw a word once more than we actually did. Where the MLE probabilities in a bigram model are given by:

$$p_{MLE}(w_i|w_{i-1}) = c(w_{i-1}, w_i) / c(w_{i-1})$$

The add-one bigram estimate is:

$$p_{Add\text{-}One}(w_i|w_{i-1}) = (1 + c(w_{i-1}, w_i)) \ / \ (c(w_{i-1}) + V)$$

… where "V" is the size of the lexicon. Afterwards, our formerly sparse bigram table looks like Table 4.

| | am | and | eggs | ham |
|---|---|---|---|---|
| am | 1 | 1 | 1 | 1 |
| and | 1 | 1 | 2 | 1 |
| eggs | 1 | 1 | 1 | 1 |
| ham | 1 | 2 | 1 | 1 |

Table 4: Bigram counts for "I am Sam…" with Laplace smoothing

---

Generally, we use more sophisticated smoothing approaches such as an interpolation language model. The intuition behind an interpolation LM is that we construct several n-gram (normally, n=1-3) language models and we calculate the probability of an event based on the weighted sum of the probabilities of each n-gram model. A simple interpolation could be:

$$p_{Int}(w_i|w_{i-2}w_{i-1}) = \boldsymbol{\lambda}_1 p(w_i|w_{i-2}w_{i-1}) + \boldsymbol{\lambda}_2 p(w_i|w_{i-1}) + \boldsymbol{\lambda}_3 p(w_i)$$

And we ensure that sum($\boldsymbol{\lambda}_i$ = 1). We set the weight of $\boldsymbol{\lambda}_i$ based on training / development.

One additional problem is unknown words, called OOV (out of vocabulary). One popular solution to this is to create a special word, <UNK> and assign probability to it based on the size of the lexicon.

### Toolkits (FYI)

NLTK contains a number of models for smoothing, collectively under the nltk.model.ngram package, accessible via the NgramModel class. However, NgramModel does not exist in the latest version (3.2.1), neither in some of the previous versions. Anyway, like NLTK, it's old and slow.

There are many toolkits which can generate a smoothed language model. The most popular (and probably the best in terms of implementation) is SRILM. While there is a Python binding of the toolkit, this binding has a large set of dependencies, including Python 3.

For fast prototyping with a "good enough" system, KenLM may be the best alternative. It has the limitation of using only Kneser-Ney smoothing, a type of backoff (pseudo-interpolated) model.

## Approaches, Features & Classification

Bringing our discussion back to the practical, let's start by discussing the one system that many use as their inspiration. We'll then discuss some of the improvements, variants, etc.

### Baseline System: Pang, Lee & Vaithyanathan

One of the first models created for sentiment analysis is described by the 2002 paper "Thumbs up? Sentiment Classification using Machine Learning Techniques" by Pang, Lee and Vaithyanathan, and was applied to the same (IMDB) movie review data we used above. Initially, they looked at a bag-of-words on intuitive (unprincipled) features words which they thought should convey positive or negative sentiment, words such as "wonderful" or "suck." The baseline accuracy for this system was 0.69. The system they built consistently surpassed this baseline and achieved a final accuracy of approximately 0.83.

---

Most successful sentiment analysis systems since then have been a variation on their theme, so it makes sense to understand their system. It is a very simple. The procedures are: extract tokens, extract features and classify. Let's look at the highlights of this system.

Step 1: Extract tokens. We have dealt with this correctly in the past. Our normal solution involves a call to NLTK, MITIE or a tool of our choice. But to be explicit: since a lot of sentiment data comes across as tweets or through social media, we should ensure that the tokens we extract have been handled correctly with respect to:

- Standard markup formats (HTML/XML, etc.) — for example, we want to consider the markup in "I really love that" as a feature but in a way that does not double-count the markup; ideally, we would like to convert this text to something like: "I REALLY love that"
- Platform-specific markup formats (hashtags, username references — eg. @msan) — all of which could be useful in finding the subject of the language
- Keeps word shape formats which may be useful for features (eg. ALL uppercase)
- Does something useful with emoticons, emoji, etc.
- Preserves dates, phone numbers and other formats
- Deliberate misspellings — for example "Yaaas queen!" indicates a satisfaction (or perhaps deep sarcasm) by the writer. SImilarly, the lengthening of the "Yes" misspelling indicates something about the writer's state.
- Considers punctuation — for example, we want to treat the following three strings differently: "Clearly." … "Clearly!" … "Clearly!!!!!"

The last point is a supremely important one if we're relying on a toolkit like NLTK. Let's determine what it does. Let's use NLTK's word_tokenize() function to determine how it tokenizes the following sequence:

```
@diane said, "Clearly, we want the #msan programme to be
the best. Clearly! Clearly!!!"
```

If the above sequence is a variable called "document", here is some code to do it:

```
print(nltk.word_tokenize(document))
```

Does it do what we want and expect? How would we have to change this if we process text from social media?

Another major issue for sentiment concerns negation. If some text contains a token sequence like "I can't decide", we may choose to declare the middle to be one token (can't) based on whitespace or two tokens (ca + n't) based on treebank grammars. The features we derive from each has its advantages, so the decision to prefer one over the other is not trivial. Generally, we will prefer the latter since it allows us to choose a larger set of tools.

Step 2: Extract features. The most useful features in the baseline system were unigram counts, known as a "bag of words" approach. In this baseline system, the tokens were not stemmed or

lemmatized and the counts were not normalised by the length of the review. All-caps tokens were treated differently from their lowercase or mixed-case equivalents.

One of the important innovations of this work[1] was the manner in which negation was handled. Let's use one part (from the second-to-last sentence) of our example tweet as an illustration:

```
[...] I still don't know.
```

… which should generate the tokens as follows:

```
I   still   do   n't   know   .
```

Upon detecting a negation token, the system adds (prefixes or suffixes) a negation marker ("NOT" or "NEG") and an underscore to the unigram feature name for all subsequent tokens until punctuation is detected. The system uses a gazetteer for negation tokens as follows: n't, never, no, nothing, nowhere, noone, none, not, havent, hasnt, hadnt cant, couldnt, shouldnt, wont, wouldnt, dont, doesnt, didnt, isnt, arent, aint. Using the example above, we want to keep counts on the following unigrams:

```
I   still   do   n't   NEG_know   .
```

Features based on bigram counts were also generated but were not found to be less effective than just the unigram counts, either alone or in combination with unigrams. This is not an atypical or unexpected result for this data. In other domains, bigrams should be quite useful.

Step 3: Classify. This being a somewhat older system, the baseline used classifiers based on three algorithms: Naive Bayes, Maximum Entropy and Support Vector Machines. All performed well and none consistently out-performed any other for the different combinations of features.

There is an example of this system at http://text-processing.com/demo/sentiment/.

## VADER additions to the Baseline System

One practical system for sentiment analysis is VADER, which is an acronym for Valence Aware Dictionary and sEntiment Reasoner. (The correct name is all uppercase, but we'll refer to it with all lowercase characters by default.) Vader began as a standalone system and was subsequently incorporated into NLTK. We'll run vader later.

Table 5 contains some additional features used by vader for sentiment analysis. The basic vader system generates a valence prediction for each bigram — i.e. positive, negative, neutral or compound. Some words serve to intensify the prediction; others only dampen the prediction[2]. Words in all caps often serve the same purpose as boosters, intensifying a valence prediction or

[1] Technically, Pang et al. were not responsible for this — it's from Das & Chen a year earlier. But the fusion with other features was unique.
[2] Both intensifiers and dampeners are called "boosters" in vader. We make the distinction here and in Table 5.

softening it. The same is true for specific punctuation combinations — compare “no way!” and “no way!!” for example.

| Feature | Example |
|---|---|
| Boosters | + Absolutely<br>+ Utterly |
| Dampeners | - Marginally<br>- Partly |
| All caps | … REALLY good. |
| Specific Punctuation | !! ??? ?!?!? |
| Canned phrases | Hella<br>“The shit” |

Table 5: Additional features in the vader sentiment analysis system

The final feature, canned phrases, may be specific to a “linguistic subculture.” For example, “hella” is an intensifier which originated with (Northern) Californian people in the Generation X and Millennial populations. It’s now widespread. While it’s generally understood throughout the US, perhaps through the English-speaking world, deriving its usage is computationally difficult without an external resource.

Vader has already been trained on a general sentiment-tagged corpus. It outputs a dictionary of scores which have the following keys: `pos`, `neg`, `neu`, each in the range (0..1) and `compound`, which ranges from (-1..1). Let’s see how this works for a file in our IMDB review data.

In the package `nltk.sentiment.vader`, we want an instance of the class called `SentimentIntensityAnalyzer`. On this class is a function, `polarity_scores(...)` which accepts a string of text. Let’s determine the score for a positive file in our corpus (`imdb_review/txt_sentoken/pos/cv109_21172.txt`). We’ll need to write a “get_text” function to return a long string of text. Once we have that, here is the code to print the score:

```
from nltk.sentiment.vader import SentimentIntensityAnalyzer

vader = SentimentIntensityAnalyzer()
text = get_text('imdb_review/txt_sentoken/pos/cv109_21172.txt')
score = vader.polarity_scores(text)
print score
```

With this working, the output should be:

```
Score {'neg': 0.117, 'neu': 0.796, 'pos': 0.087, 'compound': -0.9486}
```

---

The prediction in vader can be determined by comparing the negative to the positive score, with the larger score being the prediction about whether positive or negative valence ("sentiment") is predicted. In this case, because negative has a larger score (0.117) than the positive (0.087), vader's prediction is for a negative review. We can also determine the valence from the score in `compound`. (There's a longer discussion at StackOverflow about the details of this scoring.) Because the score in `compound` is negative, the overall valence hypothesis is negative.

The prediction for `cv109_21172.txt` is incorrect. Because of the location of this file, we know it is a positive review. If we have enough time, let's determine the overall accuracy for vader on this corpus. What are some things can we do to improve this accuracy?

### Other Ideas

In addition to the features in the baseline system and to vader, there have been several changes by the vader system. From Information Retrieval, we've seen we can remove stopwords for higher performance. (Stopwords are generally "function words" in English, such as articles ("a", "the"), prepositions ("of", "by") or pronouns ("them", "mine"). Since these words fall into one or more closed class words in most languages (English included), we can get a list of them easily. Stathis Fotiadis shows a ~5% performance improvement by pruning stopwords.

Another feature which is difficult to extract is elongation of words like, "gooood movie" because it may appear to be a misspelling. Correctly detected, this has the effect of being a valence intensifier. We know from the first meeting that the same character never appears in an English word more than twice in a row, so elongation may be detectable.

## Resources

Several external resources may be useful for sentiment analysis. Let's briefly survey four of these tools.

- **The General Inquirer** — originally from 1966?!?, this resource contains a spreadsheet (http://www.wjh.harvard.edu/~inquirer/inquirerbasic.xls) with words tagged for positive (Positiv) and negative (negativ) valence as well as other attributes (strength, weakness, etc.). The main page for the General Inquirer is http://www.wjh.harvard.edu/~inquirer.
- **LIWC** (2007) — from the same person who gave us the "look only at function words", we get the Linguistic Inquiry and Word Count resource, available at http://ww.liwc.net. This resource requires payment.
- **MPQA Subjectivity Cues Lexicon** (2005) — built for question answering, the Multi-Perspective Question Answering resource, available at

---

http://mpqa.cs.pitt.edu/lexicons/subj_lexicon/ contains words tagged for sentiment. It requires registration but no payment.

- **SentiWordNet** (2010) — available at http://sentiwordnet.isti.cnr.it/, SentiWordNet is built from the same WordNet resources we've seen before, with the addition of valence.

## Other Tools

We've seen a baseline system and vader. There are many other tools for sentiment analysis, including one in the Stanford CoreNLP, a tool we've seen before. Here are two posts which link to open source sentiment analysis tools:

- https://opensource.com/business/15/7/five-open-source-nlp-tools
- https://breakthroughanalysis.com/2012/01/08/what-are-the-most-powerful-open-source-sentiment-analysis-tools/

Lately, as with all things NLP, the trend is to move toward deep neural architectures. A really recent post on medium describes how to use LSTMs for sentiment analysis. This is a great idea for a project.