NLP: Text Similarity

mariogarcia.github.io/blog/2021/04/nlp_text_similarity.html

In NLP, the study of text similarities could be very helpul for tasks such as **looking for similar texts**, or creating **spell checkers** replacing a mispelled word by the right one.

Looking for similar texts

In order to measure how similar two different texts are, we usually calculate "the distance" between them, how far two text are to be the same. There are a few distance metrics available in NLTK:

- Jaccard distance
- Levenshtein edit-distance
- Jaro-Winkler distance
- ...

In this article I'll be using the Jaccard distance to calculate the similarity of two texts. I leave some links about the theory behind the Jaccard distance in the resources section.

SIMILARITY != DISTANCE

Similarity is not the same as distance. While similarity is how similar a text is compared to another one, distance would be how far is a given text to be the same as another text. They're kind two sides of the same story. Mathematically speaking The similarity is 1 minus the distance between both texts, therefore, regarding Jaccard distance / similarity:

- a **similarity of 1** means both texts are **identical**
- a **similarity of 0** means both texts have **nothing in common**
- a **distance of 0** means both texts are **identical**
- a **distance of 1** means both texts have **nothing in common**

I went to a famous sentences site and I picked a few and mixed them with another few made-up sentences:

famous sentences

```
sentences = [
"If you look at what you have in life, you'll always have more. If you look at what
you don't have in life, you'll never have enough.",
"Are you ok with that situation ?",
"If you set your goals ridiculously high and it's a failure, you will fail above
everyone else's success.",
"Life is what happens when you're busy making other plans.",
"I don't like to wake up early",
"Did you do your homework ?",
"This sentences is made up to be an outsider",
"I don't like the way he drives"
]
```

So I’m basically going **to compare all of them using the Jaccard similarity**, which, as I mentioned earlier would be the result of 1 minus the Jaccard distance between two sentences. I’m calculating Jaccard similarity using NLTK’s nltk.metrics.distance.jaccard_distance in the following function:

jaccard similarity with NLTK

```
from nltk.metrics.distance import jaccard_distance

def jaccard_similarity(x, y):
    return 1 - jaccard_distance(x, y)
```

The **jaccard_distance** function receives two sequences (x and y). These sequences could be:

- sequences of characters
- sequences of tokens
- sequences of n-grams

We’ll see along the next sections how **using one type of sequences or another will impact in the distance result**.

Comparing characters

The first stop it’s to compare texts by their characters. Our initial asumption is that a given text could be similar to another text if they share enough characters between them. With that in mind we’re calculating the Jaccard similarity between all sentences, this time **using sequences of characters**.

creating an 8x8 matrix with Jaccard similarities

```
import pandas as pd

rows = []

for text_1 in sentences:
    row = []
    for text_2 in sentences:
        sequence_1 = set(text_1) # sequence of characters
        sequence_2 = set(text_2) # sequence of characters

        row.append(jaccard_similarity(sequence_1, sequence_2))
    rows.append(row)

df = pd.DataFrame(data=rows, columns=["sentence_{}".format(i + 1) for i in
range(0,8)])
```

Remember that **the arguments for the jaccard_distance function are sequences**. In this particular case I'm passing a sequence of characters as a result of invoking **set(string)** which returns all unique characters contained in the string passed as parameter. As I'll point out in the next section, you can also pass a sequence of words (tokens) or n-grams.

I'm using a **heatmap to highlight which sentences are more similar than others**. The darker the color the more similar two sentences are. You can see how the main diagonal is comparing a sentence with itself, that's why it's always 1.

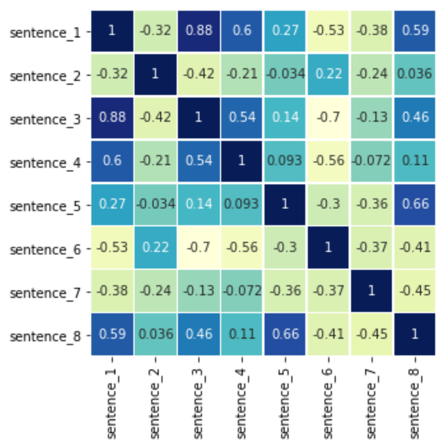
similarity matrix

```
import seaborn as sns
import numpy as np
import matplotlib.pyplot as plt

corr_matrix = np.corrcoef(df.T)

plt.figure(figsize=(5, 5))
sns.heatmap(
    corr_matrix,
    cbar=False,
    annot=True,
    square=True,
    linewidths=.5,
    cmap="YlGnBu",
    xticklabels=df.columns,
    yticklabels=df.columns)
```

After looking the heatmap carefully it seems that sentences 5 and 8 have a higher similarity than the rest:

```
"I don't like to wake up early"
"I don't like the way he drives"
```

Well, both sentences seemed to have similar grammar constructs, yes. But **there're other pairs that have a relative high similarity even though it's clear that they don't have much to do with each other**. There must be a better way of comparing texts without that much noise.



Comparing tokens

As a perfect example of how wrong could be to compare texts using just characters, you can find the following sentences:

```
from nltk.metrics.distance import jaccard_distance

sentence_1 = "aloha amigo is fun"
sentence_2 = "I am a go fan"

similarity_by_chars = 1 - jaccard_distance(set(sentence_1), set(sentence_2))
similarity_by_chars

0.5384615384615384
```

As I was saying, altough they have nothing to do with each other, still the output raised a 0.53 similarity which makes little sense. On the other hand if we compare both sentences using a sequence of tokens we will demostrate that they have nothing to do with each other:

```
import nltk
from nltk.metrics.distance import jaccard_distance

def get_similarity_by_tokens(left, right):
    tokens_1 = set(nltk.word_tokenize(left)) # sequence of tokens
    tokens_2 = set(nltk.word_tokenize(right)) # sequence of tokens

    similarity = 1 - jaccard_distance(tokens_1, tokens_2)
    return similarity


similarity_by_tokens = get_similarity_by_tokens(sentence_1, sentence_2)
similarity_by_tokens

0.0
```

However comparing two similar sentences outputs a fair result:

```
sentence_1 = "I am a big fan"
sentence_2 = "I am a tennis fan"

similarity_by_tokens = get_similarity_by_tokens(sentence_1, sentence_2)
similarity_by_tokens

0.6666666666666667
```

Comparing n-grams

A n-gram is a sequence of phonems where n could be any positive number. That allows our comparison procedure to be smarter. Instead of comparing all characters, or word by word, in this case, we would like to find certain tuples of common tokens between two sentences. For example, **What are the n-grams of sentence_1, with n=2 using tokens ?**

```
import nltk
from nltk.util import ngrams

seq_1 = set(nltk.word_tokenize("I am a big fan"))
seq_2 = set(nltk.word_tokenize("I am a tennis fan"))

list(ngrams(seq_1, n=2)), list(ngrams(seq_2, n=2))
```

n-grams

```
([('am', 'fan'), ('fan', 'big'), ('big', 'I'), ('I', 'a')],
 [('am', 'tennis'), ('tennis', 'fan'), ('fan', 'I'), ('I', 'a')])
```

Which shows that only 1 out of 4 tuples is the same. Now I'm creating a function which calculates the similarity using n-grams and jaccard distance:

```
def get_similarity_by_ngrams(left, right, n):
    seq_1 = set(ngrams(nltk.word_tokenize(left), n=n))
    seq_2 = set(ngrams(nltk.word_tokenize(right), n=n))

    similarity = 1 - jaccard_distance(seq_1, seq_2)
    return similarity
```

And apply it to the previous sentences:

```
sentence_1 = "I am a big fan"
sentence_2 = "I am a tennis fan"

similarity_by_tokens = get_similarity_by_ngrams(sentence_1, sentence_2, n=2)
similarity_by_tokens

0.33333333333333337
```

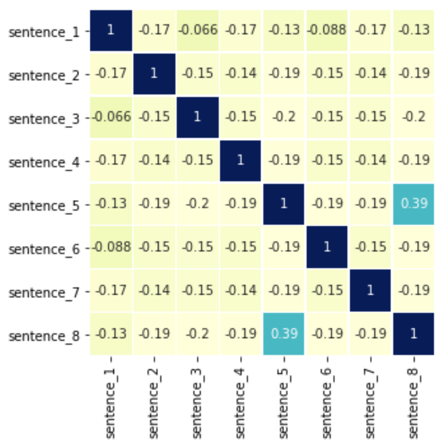
Which outputs something I would expect.

In this section, we’ve tried different strategies of comparing texts, we started comparing characters, then moving to tokens, and finally with n-grams. The overall idea, is that every step we took the higher was the normalization of the comparison. **With characters we had a lot of noise** trying to find similar sentences and as we went through the other to types, we managed to reduce the noise by a lot. In the end **n-grams and specially the n parameter could reduce the noise as a form of normalization**.

> n-grams and specially the n parameter could be used as a form of normalization

Before moving forward, lets show the heatmap of the beginning using n-grams (n=2):

Now it’s pretty clear that the only two sentences with some significant similarity are the 5th and 8th:



A simple spell checker

Another possible use of text similarity is to correct spelling mistakes. For instance, lets say we’ve got the following text written incorrectly:

> it could be a greal busines

For us, humans, is easy to check the dictionary and spot the error we did while writing a word, but it’s not as easy for a computer. In this section I’m using NLTK and the Jaccard distance to create a basic spell checker. What do we need to create a spell checker ?

- **A corpus**: the ground truth where the correct words can be found. Generally speaking a **corpus would be the dictionary where we go looking for the right spelling**
- **A method**: to check the similarity between the wrong word and the most similar word from the corpus. The method in this case would be the Jaccard similarity.

So in this context **we’re using n-grams to establish which is the minimal number of similar phonemes to compare**. In other words, instead of comparing two words letter by letter, we will be comparing two words by n-grams.

using jaccard distance

```
from nltk.util import ngrams

def get_ngrams(word, n):
    return set(ngrams(word,n=n))

def get_recommendation_for(word, n=3):
    from nltk.corpus import words
    from nltk.metrics.distance import jaccard_distance

    # we don't want stop-words to be processed
    if len(word) < 3:
        return word

    ngram        = get_ngrams(word, n)
    spellings    = words.words()
    scores       = [(w, jaccard_distance(ngram, get_ngrams(w, n))) for w in spellings
if w.startswith(word[0])]
    higest_first = sorted(scores, key=lambda entry: entry[1])

    return highest_first[0][0]
```

Now we can apply the recommender with our initial text using **n=3**:

applying jaccard distance recommender

```
wrong_text     = "it could be a greal busines"
tokens         = nltk.word_tokenize(wrong_text)
correct_words  = [get_recommendation_for(w) for w in tokens]
corrected_text = " ".join(correct_words)

print(corrected_text)
```

Which outputs:

```
it could be a great business
```

I'm curious and I wanted to see what the output would be **playing with the value of n**, and for **n=1** (aka comparing letter by letter) the spell checker turned out to be a spell mess:

```
it cloud be a gaggler busine
```

Resources

- Jaccard Index in Wikipedia
- Levenshtein_distance in Wikipedia
- N-Gram in Wikipedia

- N-Gram ranking explained: A quick-start guide to creating and visualizing n-gram ranking using nltk for natural language processing
- http://billchambers.me/tutorials/2014/12/21/tf-idf-explained-in-python.html
- Jaccard Similarity as similarity metric
- Cosine similarity vs The Levenshtein distance: from datascience.stackexchange.com