Published in Towards Data Science

Rafał Wójcik
Nov 27, 2019 · 12 min read · Member-only · Listen

Unsupervised Sentiment Analysis

How to extract sentiment from the data without any labels

. . .

Photo by Amanda Jones on Unsplash

Note: full code available in github repo:

rafaljanwojcik/Unsupervised-Sentiment-Analysis
How to extract sentiment from opinions without having them labeled Repo for article .... on Unsupervised Sentiment...
github.com

One of the common applications of NLP methods is sentiment analysis, where you try to extract from the data information about the emotions of the writer. Mainly, at least at the beginning, you would try to distinguish between positive and negative sentiment, eventually also neutral, or even retrieve score associated with a given opinion based only on text.

There are two main approaches you could take, to teach an algorithm to distinguish between positive and negative emotions in writing — a supervised, and an unsupervised one.

The first one would inquire from you to collect labeled data, and teach an algorithm (e.g. LSTM network) in a supervised manner how each word in a sequence (actually all words appearing one after another, if we talk about RNNs) corresponds to the outcome of overall sentence being negative or positive. This approach requires manually labeled data, which is often time consuming, and not always possible. Secondly, for not well exploited languages in terms of NLP, such as Polish language, there are not so many pretrained models to work with, so it's not possible to use libraries which have already pretrained models for estimating sentiment scores for each word in a sentence.

The latter approach would be an unsupervised one, and this one is an object of interest in this article. The main idea behind unsupervised learning is that you don't give any previous assumptions and definitions to the model about the outcome of variables you feed into it — you simply insert the data (of course preprocessed before), and want the model to learn the structure of the data itself. It is extremely useful in cases when you don't have labeled data, or you are not sure about the structure of the data, and you want to learn more about the nature of process you are analyzing, without making any previous assumptions about its outcome.

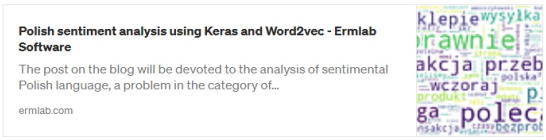
Before we hop into the main part of this article, there is one more thing that could be important to mention. One of the most important ideas in recent breakthroughs both in NLP and computer vision was efficient usage of transfer learning. In the field of NLP most of transfer learning happens in a way, that some model (let it be MLP in case of Word2Vec, or transformer like BERT) is at first trained in unsupervised manner (actually fake supervised)

Rafał Wójcik
92 Followers
Data Scientist @ MojoHire
Follow

Help Status Writers Blog Careers Privacy Terms About Text to speech

on the data, and then fine tuned on specific task, or just used in another model to produce better quality features. Usually, it is given a fake supervised task, such as predicting word based on words that surround it, or predict surrounding words based on a given word (see: word2vec), or predict next word/sentence based on previous words/sentences (transformer models). Such training shouldn't be thought of as directly supervised, as there is no human factor, that tells an algorithm what answer is the correct one (except human writing the sentence itself). I mention this because Word2Vec algorithm can be taught as an example of transfer learning.

With that being said, we arrive at the subject of this article, which is unsupervised sentiment analysis. To make things harder, and because I'm from Poland, I've chosen to analyse Polish Sentiment Dataset, though this approach should also work with any language, as I didn't make any assumptions specific to Polish language, nor use pretrained models. The dataset was collected and analyzed in a supervised approach by Szymon Płotka in this article:



There are different approaches to this problem, which I will mention at the bottom of this article, that could probably work better, but I find it quite exciting that mine actually worked, as it's one of these ideas that just bump into your head, and turn out to actually work without a lot of effort being put into them (which might also be worrying).

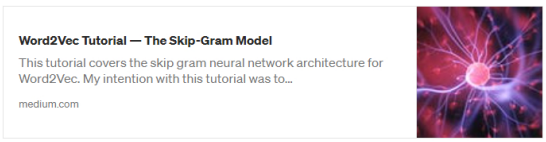
. . .

1. The main idea

Photo by Nicole Honeywill / Sincerely Media on Unsplash

The main idea behind this approach is that negative and positive words usually are surrounded by similar words. This means that if we would have movie reviews dataset, word 'boring' would be surrounded by the same words as word 'tedious', and usually such words would have somewhere close to the words such as 'didn't' (like), which would also make word didn't be similar to them. On the other hand, it would be unlikely to have happened, that word 'tedious' had more similar surrounding to word 'exciting', than to word 'boring'. With such assumption, words could form clusters (based on similarity of their surrounding) of negative words that have similar surroundings, positive words that have similar surroundings, and some neutral words that end up between them (such as 'movie'). It might seem not quite convincing at the beginning, and I might not be perfect explainer, but it actually turns out to be true.

The perfect tool for such problem (of having words that are similar to their surrounding) is the one and only word2vec! If you haven't heard of it before, here is a article about word2vec algorithm by Chris McCornick:



And perfect tutorial by Pierre Megret, which I used in this article to train my own word embeddings:

Gensim Word2Vec Tutorial

Download Open Datasets on 1000s of Projects + Share Projects on One Platform. Explore Popular Topics Like Government...

www.kaggle.com

2. The Data

Photo by Robert Bye on Unsplash

The first, the only, and the most important step in every Data Science/Machine Learning project is data preparation. Without good data quality, it is always possible to end up with a biased model, that is either not performing well according to some metric we choose(e.g. F-score on test set) or, which is harder to diagnose at the beginning, has been taught biased relations, that actually doesn't reflect it's availability to e.g. distinguish positive and negative emotions, but just allowed it to perform well on given data set.

The cell below presents one of basic text preparation steps that I've chosen to use, but I didn't include all of them, as everything is included in my repository, and I don't want to make the article less readable. Frankly speaking, I didn't spend a lot of time on it, and there is still plenty of space to do your own preparations, especially if you would try to implement it for languages like English, that have libraries for text normalization. For Polish language it could be really important to use tools like Morfologik, to stem the words to their basic structure, as we have a lot of different word suffixes that change the word for the model, but actually mean exactly the same thing (e.g. 'beznadziejny' and 'beznadziejna' both mean hopeless but the first one refers to a man, and the other to a woman).

```python
from unidecode import unidecode

def text_to_word_list(text, remove_polish_letters):
    ''' Pre process and convert texts to a list of words
    method inspired by method from eliorc github repo: https://github.com/eliorc/Medium/blob/mas
    text = remove_polish_letters(text)
    text = str(text)
    text = text.lower()

    # Clean the text
    text = sub(r"[^A-Za-z0-9^,!?.\/'+]", " ", text)
    text = sub(r"\+", " plus ", text)
    text = sub(r",", " ", text)
    text = sub(r"\.", " ", text)
    text = sub(r"!", " ! ", text)
    text = sub(r"\?", " ? ", text)
    text = sub(r"'", " ", text)
    text = sub(r":", " : ", text)
    text = sub(r"\s{2,}", " ", text)

    text = text.split()

    return text

file_cleaned.title = file_cleaned.title.apply(lambda x: text_to_word_list(x, unidecode))
```

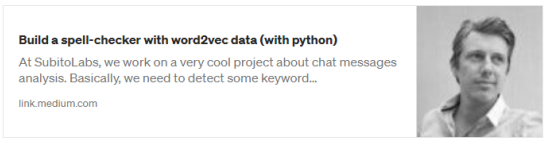
cleaning.py hosted with ♥ by GitHub view raw

https://gist.github.com/rafaljanwojcik/f00dfae9843dadc0220eba3d36694e27

All the steps that I've chosen to include:

- dropping rows with missing (NaN) values,
- dropping duplicated rows,
- removing rows with rate equal to 0, as it contained some error, probably from the data gathering phase,
- replacing polish letters with use of unidecode package,
- replacing all non-alphanumeric signs, punctuation signs, and duplicated white spaces with a single white space
- retaining all rows with sentences with a length of at least 2 words

Another idea could be to implement spell checker, in order to prevent training too many embeddings of words, that actually mean exactly the same thing. Here is great article about spell checker that uses Word2Vec and Levenstein distance, to detect semantically most similar words:



After cleaning the words, there were several other steps taken to prepare the data for word2vec model, all of which are included in my github repo. Main steps included most frequent bigrams of words detection and replacement with gensim's *Phrases* module. All these steps and most of the hyperparameters in Word2Vec model I used were based on the Word2Vec tutorial from kaggle that I linked before.

3. Word2Vec model

Photo by Max Kleinen on Unsplash

In this exercise, I used gensim's implementation of word2vec algorithm with CBOW architecture. I trained 300 dimensional embeddings with lookup window equal to 4, negative sampling was set to 20 words, sub-sampling to 1e-5, and learning rate decayed from 0.03 to 0.0007.

```
w2v_model = Word2Vec(min_count=3,
                     window=4,
                     size=300,
                     sample=1e-5,
                     alpha=0.03,
                     min_alpha=0.0007,
                     negative=20,
                     workers=multiprocessing.cpu_count()-1)
```

4. K-Means clustering

Photo by Arnaud Mariat on Unsplash

K-means clustering is a basic technique for data clustering, and it seemed most suitable for a given problem, as it takes as an input number of necessary clusters, and outputs coordinates of calculated clusters centroids (central points of discovered clusters). It is an iterative algorithm, in which in first step n random data points are chosen as coordinates of clusters centroids (where n is the number of seeked clusters), and next in every step all points are assigned to their closest centroid, based on euclidean distance. Next, new coordinates of every centroid are calculated

euclidean distance. Next, new coordinates of every centroid are calculated, as mean of coordinates of all data points assigned to each centroid, and iterations are repeated till reaching minimal value of squared sum of distances between points assigned to centroids, and their centroid coordinates (which just simply means that coordinates of clusters stop to change), or number of iterations reach given limit.

In the given problem I used sklearn's implementation of K-means algorithm with 50 repeated starting points, to presumably prevent the algorithm from choosing wrong starting centroid coordinates, that would lead the algorithm to converge to not optimal clusters, and 1000 iterations of reassigning points to clusters.

```
word_vectors = Word2Vec.load("../preprocessing_and_embeddings/word2vec.model").wv
model = KMeans(n_clusters=2, max_iter=1000, random_state=True, n_init=50).fit(X=word_vectors.vect
positive_cluster_center = model.cluster_centers_[0]
negative_cluster_center = model.cluster_centers_[1]
```

clustering.py hosted with ♥ by GitHub view raw

https://gist.github.com/rafaljanwojcik/275f18d3a02f6946d1ff3bf50a563c2b

After running it on estimated word vectors, I got 2 centroids, with coordinates that can be retrieved with method:

```
model.cluster_centers_
```

Next, to check which cluster is relatively positive, and which negative, with use of gensim's most_similar method I checked what word vectors are most similar in terms of cosine similarity to coordinates of first cluster:

```
word_vectors.similar_by_vector(model.cluster_centers_[0], topn=10,
restrict_vocab=None)
```

which outputs:

```
[('pelen_profesjonalim', 0.9740794897079468),
 ('superszybko_supersprawnie', 0.9732559919357 3),
 ('bardzon', 0.9731361865997314),
 ('duzu_wybor', 0.971358060836792),
 ('ladne_garnki', 0.9698898196220398),
 ('najlpszym_porzadku', 0.9690271615982056),
 ('wieloma_promocjami', 0.9684171676635742),
 ('cudowna_wspolpraca', 0.9679782390594482),
 ('pelen_profesjonaliz', 0.9675517678260803),
 ('przyzwoicie_cenowo', 0.9674378633499146)]
```

As you can see (if you know Polish, which I encourage you to learn if you want to have some superpowers to show off with) 10 closest words to cluster no. 0 in terms of cosine distance are the ones with positive sentiment. Some words classified to cluster 0 are even contextually positive, e.g. collocation 'miod_malina', which consists of words that literally mean 'honey' and 'raspberry', means that something is amazing and perfect, and it got sentiment score (inverse of distance from cluster it was assigned to, see the code in repository for details) of +1.363374.

The negative cluster is harder to describe, as not all most similar words that end up closest to it's centroid are directly negative, but when you check if words like 'hopeless', 'poor' or 'broken' are assigned to it, you get quite good results, as all of them end up where they should have.

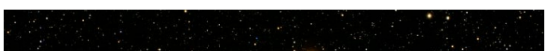
```
temp[temp.words.isin(['beznadziejna', 'slaba', 'zepsuty'])]
```

gives:

```
{'zepsuty': -1.2202580315412217,
 'slaba': -1.0219668995616882,
 'beznadziejna': -1.0847829796850548}
```

It might seem tricky, that I use cosine distance to determine the sentiment of each cluster, and then euclidean distance to assign each word to a cluster, but there is no motivation behind it, I just used available methods from both libraries, and it worked.

5. Assigning clusters



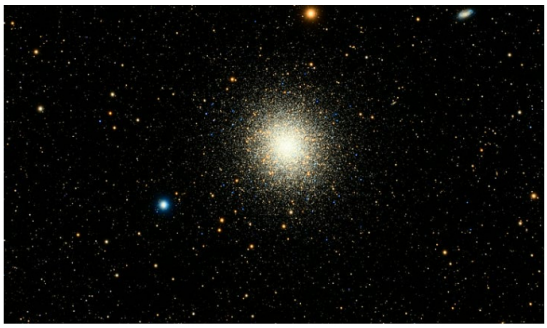


Photo by Guillermo Ferla on Unsplash

Next step, partially mentioned in the previous chapter, was to assign each word sentiment score — negative or positive value (-1 or 1) based on the cluster to which they belong. To weigh this score I multiplied it by how close they were to their cluster (to weigh how potentially positive/negative they are). As the score that K-means algorithm outputs is distance from both clusters, to properly weigh them I multiplied them by the inverse of closeness score (divided sentiment score by closeness score).

```
words = pd.DataFrame(word_vectors.vocab.keys())
words.columns = ['words']
words['vectors'] = words.words.apply(lambda x: word_vectors.wv[f'{x}'])
words['cluster'] = words.vectors.apply(lambda x: model.predict([np.array(x)]))
words.cluster = words.cluster.apply(lambda x: x[0])
words['cluster_value'] = [1 if i==0 else -1 for i in words.cluster]
words['closeness_score'] = words.apply(lambda x: 1/(model.transform([x.vectors]).min()), axis=1)
words['sentiment_coeff'] = words.closeness_score * words.cluster_value
```

assigning_clusters.py hosted with ♥ by GitHub view raw

https://gist.github.com/rafaljanwojcik/865a9847e1fbf3299b9bf1f1a164bdf9

With these steps being complete, there was full dictionary created (in form of pandas DataFrame), where each word had it's own weighted sentiment score. To assess how accurate these weighted sentiment coefficients were, I randomly sampled dataframe with obtained coefficients. As you can see, for most of you probably with help of google translate, words in the table below mostly end up in the correct cluster, though I must admit that many words didn't look so promising. Probably, the best option to correct it would be to normalize data properly or to create 3rd, neutral cluster for words that shouldn't have any sentiment at all assigned to them, but in order to not make this project too big, I didn't improve them, and it still worked pretty well, as you will see later.

	words	sentiment_coeff
8603	bezzwlocznie	-0.970751
61219	przyczyny_opoznienia	-1.275320
38906	delikatnie_uszkodzony	-1.276431
38670	chetnie_sluza	1.160198
50968	mniej_popularne	1.190393
11128	czterokrotnie	-1.038108
50111	pekniete	-1.144333
14846	expresowe_zalatwienie	1.534966
25455	/nie	-1.149102
5587	samoczynnie	-1.337816

sample of words with calculated weighted sentiment coefficients

6. Tfidf weighting and sentiment prediction

Photo by Brett Jordan on Unsplash

Next step was to calculate tfidf score of each word in each sentence with sklearn's TfidfVectorizer. This step was conducted to consider how unique every word was for every sentence, and increase positive/negative signal associated with words that are highly specific for given sentence in comparison to whole corpus.

```
tfidf = TfidfVectorizer(tokenizer=lambda y: y.split(), norm=None)
tfidf.fit(file_weighting.title)
features = pd.Series(tfidf.get_feature_names())
transformed = tfidf.transform(file_weighting.title)
```

tfidf_vectorizer.py hosted with ♥ by GitHub view raw

https://gist.github.com/rafaljanwojcik/9d9a94249388112862966458 3e66fb3a

Finally, all words in every sentence were on one hand replaced with their tfidf scores, and on the other with their corresponding weighted sentiment scores.

```
def create_tfidf_dictionary(x, transformed_file, features):
    '''
    create dictionary for each input sentence x, where each word has assigned its tfidf score

    inspired  by function from this wonderful article:
    https://medium.com/analytics-vidhya/automated-keyword-extraction-from-articles-using-nlp-bfd

    x - row of dataframe, containing sentences, and their indexes,
    transformed_file - all sentences transformed with TfidfVectorizer
    features - names of all words in corpus used in TfidfVectorizer

    '''
    vector_coo = transformed_file[x.name].tocoo()
    vector_coo.col = features.iloc[vector_coo.col].values
    dict_from_coo = dict(zip(vector_coo.col, vector_coo.data))
    return dict_from_coo

def replace_tfidf_words(x, transformed_file, features):
    '''
    replacing each word with it's calculated tfidf dictionary with scores of each word
    x - row of dataframe, containing sentences, and their indexes,
    transformed_file - all sentences transformed with TfidfVectorizer
    features - names of all words in corpus used in TfidfVectorizer
    '''
    dictionary = create_tfidf_dictionary(x, transformed_file, features)
    return list(map(lambda y:dictionary[f'{y}'], x.title.split()))

%%time
replaced_tfidf_scores = file_weighting.apply(lambda x: replace_tfidf_words(x, transformed, featu
```

tfidf_weighting.py hosted with ♥ by GitHub view raw

https://gist.github.com/rafaljanwojcik/ec7cd1f4493db1be44d83d32e8a6c6c5

Gists above and below present functions for replacing words in sentences with their associated tfidf/sentiment scores, to obtain 2 vectors for each sentence

```
def replace_sentiment_words(word, sentiment_dict):
    '''
    replacing each word with its associated sentiment score from sentiment dict
    '''
    try:
        out = sentiment_dict[word]
    except KeyError:
        out = 0
    return out

  replaced_closeness_scores = file_weighting.title.apply(lambda x: list(map(lambda y: replace_se
```

sentiment_replacement.py hosted with ♥ by GitHub view raw

https://gist.github.com/rafaljanwojcik/fa4c85f22cc1fedda25f156d3715ccae

The dot product of such 2 sentence vectors indicated whether overall sentiment was positive or negative (if the dot product was positive, the sentiment was positive, and in opposite case negative).

```
replacement_df = pd.DataFrame(data=[replaced_closeness_scores, replaced_tfidf_scores, file_weight
replacement_df.columns = ['sentiment_coeff', 'tfidf_scores', 'sentence', 'sentiment']
replacement_df['sentiment_rate'] = replacement_df.apply(lambda x: np.array(x.loc['sentiment_coeff
replacement_df['prediction'] = (replacement_df.sentiment_rate>0).astype('int8')
```

predictions.py hosted with ♥ by GitHub view raw

https://gist.github.com/rafaljanwojcik/9add154cb42b2450d68134a7150de65c

7. Model scores

Photo by Markus Spiske on Unsplash

Chosen metric for evaluating model's performance was precision, recall, and F-score, mainly because classes in dataset were highly imbalanced, but in fact, the dataset was so highly imbalanced, that I should have probably come up with a metric that would punish this imbalance even more. It turned out, that model achieved 0.99 precision, which shows that it was really good at discriminating negative sentiment observations (it almost didn't mistake negative observations with positive ones). One could argue that it's quite obvious that it should have, as it had very few negative observations, and they probably differed the most from others, and it's partially true, but if you consider that the model also achieved almost 80% recall (which means that 80% of all positive observations in the dataset were correctly classified as positive), it might show, that it also learned quite a lot, and didn't just split the data in half, with negative observations ending up in the correct cluster. If you compare these results with ones achieved by Szymon Płotka in his article, precision of unsupervised model is actually higher than his supervised model, and accuracy and recall are ~17.5 p.p. lower, though it's hard to compare, as we used different test sets (mine consisted of full dataset, and his from 20% of original data).

Confusion Matrix

	0	1
0	9523	306
1	127125	508277

Scores	
accuracy	0.802503
precision	0.999398
recall	0.799930
F1	0.888608

To sum up, unsupervised approach achieved quite good results (in my opinion), as without the use of any pretrained models, and actually no previous information what is positive or negative in given text, it achieved quite high metrics, significantly higher than predicted at random. Frankly speaking, I'm quite interested in hearing from you how it worked for your datasets!

8. Further discussion

Photo by Zdeněk Macháček on Unsplash

This article was written mainly to present an idea about unsupervised language processing, not to create the best possible solution based on it, so there is plenty of space to improve it. Improvements that come into my mind, other than ones I already mentioned before, include:

- K-Means clustering based on cosine, not euclidean distances
- Include third, neutral cluster, or assign some words that end up somewhere between positive and negative clusters sentiment score equal to zero
- Hyperparameter tuning of Word2Vec algorithm, based on e.g. F1-score achieved on dataset (though it would require splitting the dataset into

train and test datasets, as the training would become supervised)

- Not considering bi-grams of words

Here we arrive at the end of this short article—I really hope you enjoyed it and look forward to hearing from you about any improvements that you came up with. I also hope that it was somehow informative to you, and thank you for reading it!

All the best, and may the high F1-score be with you!

Rafael

. . .

Machine Learning · Sentiment Analysis · Deep Learning · NLP · Unsupervised Learning

530 · 6

Can ChatGPT Write Better SQL than a Data Analyst?

I tried ChatGPT, a variant of the GPT-3 language model that is specifically designed for generating human-like text in a conversational context. And of course, like most of us, I wondered: c...

Chatgpt 6 min read

Ahmed Besbes · Feb 7 ✦ Member-only

12 Python Decorators to Take Your Code to the Next Level

Do more things with less code without compromising on quality — Python decorators are powerful tools that help you produce clean,...

Programming 11 min read

Read more from Towards Data Science