Photo by Kate Stone Matheson on Unsplash

[NLP] Performance of Different Word Embeddings on Text Classification

compared among word2vec, TF-IDF weighted, GloVe and doc2vec

Tom Lin Follow
Jul 10, 2019 · 9 min read ★

The Incentive

It's been a while not able to write new posts, so sad, but now finally I am back again to share some of the knowledge I've just acquired. This time is about NLP.

As a fresh rookie in NLP, I'd like to play around and test out how different methods of creating doc vector perform on text classification. This post will be highly focused on feature engineering side, that is word vectorization, and less on modeling. Thus, without further due, let's get started.

Brief Introduction

The word embeddings being investigated here are word2vec, TF-IDF weighted word2vec, pre-train GloVe word2vec and doc2vec. The packages needed are Gensim, Spacy and Scikit-Learn. Spacy is used in doc preprocessing, including stop word removal and custom token selection based on its part of speech. Gensim is heavily applied for training word2vec and doc2vec, and lastly, Scikit-Learn is for classifier building and training.

Quick Summary

After a series of comparison on different word embedding/averaging methods, it turns out that **custom-trained word embedding** and its averaging method, either simple mean or TF-IDF weighted has the best performance, while on the contrary, GloVe word embedding or custom-trained Doc2vec perform slightly worse than the former word embedding.

Besides, even if we try to concatenate both word2vec and doc2vec as a whole feature set, it performs equally the same to just using averaging word embedding alone. In other words, no need to use both word2vec and doc2vec at the same time.

Special Credits to the Following Posts and Authors

In creating my **python class object** used for text preprocessing, I referred from these well-written posts.

- The post "Text Classification with Word2vec" by nadbor demos how to write your own class to compute average word embedding for doc, either simple averaging or TF-IDF weighted one.

- “Multi-Class Text Classification Model Comparison and Selection” by Susan Li teaches me how to write beautiful averaging function for word embedding.
- This tutorial “Gensim Doc2vec Tutorial on the IMDB Sentiment Dataset” has step by step guidance on how to create doc2vec via Gensim.
- “Distributed representations of sentences and documents” by Le & Mikolov presents a clear and easy-to-understand explanation on what’s going under doc2vec.

. . .

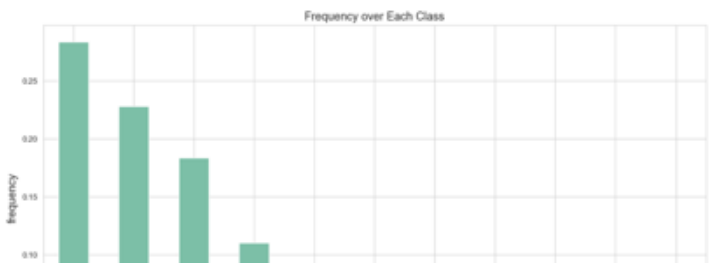
Data Preparation

The dataset I am gonna use here is consumer complaints dataset on financial product/service as referred from the post[1]. The dataset is collected and published by US GOV CFPB, while we can also download the dataset from Kaggle.

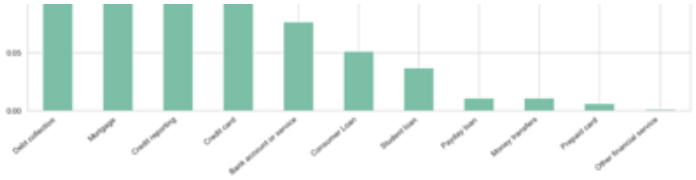
The original dataset contains more than 500 thousands records, and columns include *product, sub_product, issue, consumer_complaint_narrative, and company_response_to_consumer etc.*. We will just use **product** as text label and **consumer_complaint_narrative** as text itself. After dropping rows of missing values on consumer complaint we are left with around 60 thousands records. In order to lessen the computing pressure, I will just experiment on the first 25 thousands records only.

Now, let’s see how frequency distributed among each label.

	freq	count
Debt collection	0.28372	7093
Mortgage	0.22812	5703
Credit reporting	0.18336	4584
Credit card	0.11000	2750
Bank account or service	0.07692	1923
Consumer Loan	0.05168	1292
Student loan	0.03668	917



Payday loan	0.01092	273
Money transfers	0.01080	270
Prepaid card	0.00616	154
Other financial service	0.00164	41



Distribution of Each Label in the Dataset

We can tell that it's a highly imbalanced dataset, where **Debt Collection and Mortgage** account for half of the total records, while the most scarce class, **Prepaid Card and Other Financial Service** account for less than 1% in the dataset.

Following is the demo of (label, text) examples.

```
Demo of product and its complaint example...
```

	product	consumer_complaint_narrative
190126	Debt collection	XXXX has claimed I owe them {$27.00} for XXXX years despite the PROOF of PAYMENT I sent them : canceled check and their ownPAID INVOICE for {$27.00}! \nThey continue to insist I owe them and collection agencies are after me. \nHow can I stop this harassment for a bill I already paid four years ago? \n
190135	Consumer Loan	Due to inconsistencies in the amount owed that I was told by M & T Bank and the amount that was reported to the credit reporting agencies, I was advised to write a good will letter in order to address the issue and request the negative entry be removed from my credit report all together. I had a vehicle that was stolen and it was declared a total loss by insurance company. The insurance company and the GAP insurancw companypaid the outstanding balance of the loan, but I was told by M & T Bank that there was still a balance due on the loan. In good faith, without having received any proof as to why there was still a balance, I made a partial payment towards the remaining debt. I then sent the goodwill letter still offering to pay the remainder of the debt, but in exchange for the removal of the negative entry on my credit report. At one point, in XXXX 2015, per my credit monitoring agency, it showed a delinquent balance of {$0.00}, but when I checked my credit report again on XXXX X...
190155	Mortgage	In XX/XX/XXXX my wages that I earned at my job decreased by almost half, by XX/XX/XXXX I knew I was in trouble with my home loan. I began contacting WFB whom my home loan is with, for assitance and options. \nIn early XX/XX/XXXX I began the Loan Modification process with Wells Fargo Bank. I was told that they would not assist me with anything financial on my home loan until I fell 90 days behind, though at the time I started to inquire for assistance from WFB I was only a few weeks behind. So, I began working with a program called XXXX. They approved me for a variety of assistence and reached out to Wells Fargo Bank to determine what they could assist with. Wells Fargo then turned down the assistance from XXXX and finally offered to do a Loan Modification for me. The outcome was totally unknow about what I would be offered in the end by WFB for assistance. Wells Fargo lost my paperwork twice during this process, so it took 2 months from the time I started to the time my paperwork b...
190207	Mortgage	I have an open and current mortgage with Chase Bank # XXXX. Chase is reporting the loan payments to XXXX but XXXX is surpressing the information and reporting the loan as Discharged in BK. This mortgage was reaffirmed in a Chapter XXXX BK discharged dated XXXX/XXXX/2013. Chase keeps referring to BK Law for Chapter XXXX and we keep providing documentation for Chapter XXXX, and the account should be open and current with all the payments \n
190208	Mortgage	XXXX was submitted XX/XX/XXXX. At the time I submitted this complaint, I had dealt with Rushmore Mortgage directly endeavoring to get them to stop the continuous daily calls I was receiving trying to collect on a mortgage for which I was not responsible due to bankruptcy. They denied having knowledge of the bankruptcy, even though I had spoken with them about it repeatedly and had written them repeatedly referencing the bankruptcy requesting them to cease the pursuit, they continued to do so. When they were unable to trick me into paying, force me into paying in retaliation they placed reported to my credit bureaus a past due mortgage amount that had been discharged in Federal Court. On XX/XX/XXXX Rushmore responded the referenced complaint indicating that they would remove the reporting from my bureau, yet it is still there now in XX/XX/XXXX. I would like them to remove it immediately and send me a letter indicating that it should not have been there in the first place and they ar...

Demo of Product(Label), Consumer Complaints(Text)

. . .

Document Preprocessing

Now comes the first step —Doc Preprocessing. Before we create our own word embedding based on the input texts, we need to preprocess the text so that it complies with the input format as Gensim requires. It involves multiple steps starting from word tokenization, bi-gram detection, lemmatization etc..

Here, I wrote a python class called **DocProcess**. This class implements all the nitty-gritty jobs mentioned above for us under the hood, such as:

1. First, the class takes in a series of texts, then tokenizes the text and removes all punctuations.
2. It has the option `build_bi`, meaning whether to build up bi-gram, function adopted from Gensim. The default is False, if option `build_bi` is set to True, then the class will train a bi-gram detector and create bi-gram words for the text.
3. Now, all the processed tokens are concatenated back to form a sentence again.
4. The texts are tokenized once again, but this time, both **stop words** and **parts of speech** that are not allowed in the text will be removed and all tokens are **lemmatized**. These tokens are stored as `self.doc_words` — list of the tokens for each text(doc).
5. Finally, these `self.doc_words` are wrapped up into **TaggedDocument**, a object type in Gensim for later use in doc2vec training. It's stored in `self.tagdocs`

```
class DocPreprocess(object):

        def __init__(self,
                nlp,
                stop_words,
                docs,
                labels,
                build_bi=False,
                min_count=5,
                threshold=10,
                allowed_postags=['ADV', 'VERB', 'ADJ', 'NOUN', 'PROPN', 'NUM']):

                self.nlp = nlp  # spacy nlp object
                self.stop_words = stop_words  # spacy.lang.en.stop_words.STOP_WORDS
                self.docs = docs  # docs must be either list or numpy array or series of
                self.labels = labels # labels must be list or or numpy array or series o
                self.doc_ids = np.arange(len(docs))
                self.simple_doc_tokens = [gensim.utils.simple_preprocess(doc, deacc=True

                if build_bi:
```

```
                if build_bi:
                        self.bi_detector = self.build_bi_detect(self.simple_doc_tokens,
                        self.new_docs = self.make_bigram_doc(self.bi_detector, self.simp
                else:
                        self.new_docs = self.make_simple_doc(self.simple_doc_tokens)
                self.doc_words = [self.lemmatize(doc, allowed_postags=allowed_postags) f
                self.tagdocs = [TaggedDocument(words=words, tags=[tag]) for words, tag i
```

DocPreprocess.py hosted with ❤️ by GitHub view raw

Snippet of Class "DocPreprocess"

With the class, I can easily implement doc preprocess with just one line.

```
from UtilWordEmbedding import DocPreprocess
import spacy

nlp = spacy.load('en_core_web_md')
stop_words = spacy.lang.en.stop_words.STOP_WORDS

all_docs = DocPreprocess(nlp, stop_words,
df['consumer_complaint_narrative'], df['product'])
```

Now let's inspect what the output of doc preprocess is like.

```
In [6]: # Demo: structure of preprocessed docs.
        type(all_docs)
Out[6]: UtilWordEmbedding.DocPreprocess

In [7]: len(all_docs.tagdocs)
Out[7]: 25000

In [8]: print('Demo of tagged document...')
        all_docs.tagdocs[4]
        Demo of tagged document...
Out[8]: TaggedDocument(words=['xxxx', 'submit', 'xx', 'time', 'submit', 'complaint', 'deal', 'rushmore', 'mortgage', 'directl
        y', 'endeavor', 'stop', 'continuous', 'daily', 'call', 'receive', 'try', 'collect', 'mortgage', 'responsible', 'bankr
        uptcy', 'deny', 'have', 'knowledge', 'bankruptcy', 'speak', 'repeatedly', 'write', 'repeatedly', 'reference', 'bankru
        ptcy', 'request', 'cease', 'pursuit', 'continue', 'unable', 'trick', 'pay', 'force', 'pay', 'retaliation', 'place',
        'report', 'credit', 'bureau', 'mortgage', 'discharge', 'federal', 'court', 'xx', 'xx', 'xxxx', 'rushmore', 'respond',
        'reference', 'complaint', 'indicate', 'remove', 'reporting', 'bureau', 'xx', 'xx', 'like', 'remove', 'immediately',
        'send', 'letter', 'indicate', 'place', 'go', 'remove', 'bureaus', 'rushmore', 'speak', 'represent', 'new', 'note', 'h
        older', 'cfpb', 'involve', 'identify', 'servicing', 'agency', 'xxxx', 'credit', 'bullying', 'racial', 'discriminatio
        n', 'practice', 'damaging', 'expose', 'tactic', 'need', 'stop', 'deny', 'intent', 'walk', 'away', 'penalty', 'kind',
        'reason', 'continue', 'assist', 'procure', 'resolution'], tags=[4])

In [10]: print('Demo of doc words...')
         all_docs.doc_words[4][:10]
         Demo of doc words...
Out[10]: ['xxxx',
          'submit',
          'xx',
          'time',
          'submit',
          'complaint',
          'deal',
          'rushmore',
          'mortgage',
          'directly']

In [11]: print('Label of tagged document...')
         all_docs.labels.iloc[4]
```

```
Label of tagged document...
Out[11]: 'Mortgage'
```

The Content Stored in DocPreprocess Class

From above, we can tell it's very handy that the class has stored tokenized words, labels and tagged document, which all are ready for use later.

. . .

Word Model — Word2vec Training

Since the text are properly processed, we're ready to train our word2vec via Gensim. Here I chose the dimension size 100 for each word embedding and window size of 5. The training iterates for 100 times.

```
word_model = Word2Vec(all_docs.doc_words, min_count=2, size=100,
window=5, workers=workers, iter=100)
```

Photo by Daria Nepriakhina on Unsplash

It's Break Time, and shortly, let's continue...

Averaging Word Embedding for Each Doc

OK! Now we have the word embedding at hand, we'll be using the word embedding to compute for representative vector for whole text. It then serves as feature input for text classification model. There are various ways to come up with doc vector. First, let's start with the simple one.

(1) Simple Averaging on Word Embedding

This is a rather straightforward method. It directly averages all word embedding occurred in the text. Here I adapted the code from these two posts [2][3] and created the class **MeanWordEmbeddingVectorizer**.

```
class MeanEmbeddingVectorizer(object):

        def __init__(self, word_model):
                self.word_model = word_model
                self.vector_size = word_model.wv.vector_size

        def fit(self):  # comply with scikit-learn transformer requirement
                return self

        def transform(self, docs):  # comply with scikit-learn transformer requirement
                doc_word_vector = self.word_average_list(docs)
                return doc_word_vector

        def word_average(self, sent):
                """
                Compute average word vector for a single doc/sentence.


                :param sent: list of sentence tokens
                :return:
                        mean: float of averaging word vectors
                """
                mean = []
                for word in sent:
                        if word in self.word_model.wv.vocab:
                                mean.append(self.word_model.wv.get_vector(word))

                if not mean:  # empty words
```

```
                        # If a text is empty, return a vector of zeros.
                        logging.warning("cannot compute average owing to no vector for {
                        return np.zeros(self.vector_size)
                else:
                        mean = np.array(mean).mean(axis=0)
                        return mean


        def word_average_list(self, docs):
                """
                Compute average word vector for multiple docs, where docs had been toker

                :param docs: list of sentence in list of separated tokens
                :return:
                        array of average word vector in shape (len(docs),)
                """
                return np.vstack([self.word_average(sent) for sent in docs])
```

MeanEmbeddingVectorizer.py hosted with ❤️ by GitHub view raw

Class of MeanWordEmbeddingVectorizer

It has both `self.fit()` and `self.transform()` method so that to be compatible with other functionalities in scikit-learn. What the class does is rather simple. Initiate the class with the word model(trained word embedding), it then can transforms all tokens in the text into vectors and does the averaging to come up with representative doc vector. If the doc has no tokens, then it will return a zero vector.

Just one reminder that the input for `self.transform()` must be list of doc tokens, instead of doc text itself.

```
from UtilWordEmbedding import MeanEmbeddingVectorizer

mean_vec_tr = MeanEmbeddingVectorizer(word_model)
doc_vec = mean_vec_tr.transform(all_docs.doc_words)
```

(2) TF-IDF Weighted Averaging on Word Embedding

Not just satisfied with simple averaging? We can further adopt TF-IDF as weights for each word embedding. This will amplify the role of significant

word in computing doc vector. Here, the whole process is implemented under class **TfidfEmbeddingVectorizer**. Again, the code is adapted from the same post source.

One thing worth noted is that, the **Term Frequency** has already been considered when we conduct averaging over the text, but not **Inverse Document Frequency**, thus the weight literally being the IDF, and the unseen word is assigned the max IDF in default setting. The snippet of code can be checked in this gist.

And the other thing to note is that we need to **fit the class with tokens first**, for it must loop through all the words before hand in order to compute IDF.

```
from UtilWordEmbedding import TfidfEmbeddingVectorizer

tfidf_vec_tr = TfidfEmbeddingVectorizer(word_model)
tfidf_vec_tr.fit(all_docs.doc_words)  # fit tfidf model first
tfidf_doc_vec = tfidf_vec_tr.transform(all_docs.doc_words)
```

(3) Leverage Pre-train GloVe Word Embedding

Let's include another option — leveraging the existing pre-trained word embedding and see how it performs in text classification. Here I follow up the instructions from Stanford NLP course(CS224N) notebook, importing GloVe word embedding into Gensim to compute for averaging word embedding on text.

As a side note, I've also tried to apply Tf-IDF weighted method on GloVe vector, but found out that the result is basically the same as the ones from TF-IDF weighted averaging doc vector. Thus, I omit the demonstration and just include simple averaging on GloVe word vector here.

```
# Apply word averaging on GloVe word vector.
```

```
glove_mean_vec_tr = MeanEmbeddingVectorizer(glove_word_model)
glove_doc_vec = glove_mean_vec_tr.transform(all_docs.doc_words)
```

(4) Apply Doc2vec Training Directly

Last but not least, we still have one more option — to directly train doc2vec, and no need to average all word embeddings. Here I chose **PV-DM model** to train my doc2vec.

The script is mostly referred from Gensim tutorial[4]. And again, to save all the labor, I create a class **DocModel** for it. The class just needs to take in the **TaggedDocument** and then we call `self.custom_train()` method, the doc model will train itself.

```
class DocModel(object):

        def __init__(self, docs, **kwargs):
                """

                :param docs: list of TaggedDocument
                :param kwargs: dictionary of (key,value) for Doc2Vec arguments
                """
                self.model = Doc2Vec(**kwargs)
                self.docs = docs
                self.model.build_vocab([x for x in self.docs])

        def custom_train(self, fixed_lr=False, fixed_lr_epochs=None):
                """
                Train Doc2Vec with two options, without fixed learning rate(recommended)
                Fixed learning rate also includes implementation of shuffling training c


                :param fixed_lr: boolean
                :param fixed_lr_epochs: num of epochs for fixed lr training
                """
                if not fixed_lr:
                        self.model.train([x for x in self.docs],
                                         total_examples=len(self.docs),
                                         epochs=self.model.epochs)
                else:
                        for _ in range(fixed_lr_epochs):
                                self.model.train(utils.shuffle([x for x in self.docs]),
                                                 total_examples=len(self.docs),
                                                 epochs=1)
```

```
                self.model.alpha -= 0.002
                self.model.min_alpha = self.model.alpha  # fixed learnin


    def test_orig_doc_infer(self):
        """
        Use the original doc as input for model's vector inference,
        and then compare using most_similar()
        to see if model finds the original doc id be the most similar doc to the
        """
        idx = np.random.randint(len(self.docs))
        print('idx: ' + str(idx))
        doc = [doc for doc in self.docs if doc.tags[0] == idx]
        inferred_vec = self.model.infer_vector(doc[0].words)
        print(self.model.docvecs.most_similar([inferred_vec]))  # wrap vec in a
```

DocModel.py hosted with ❤️ by GitHub view raw

Class of DocModel

Noted that `self.custom_train()` has the option to use *fixed learning rate*. It's said that fixed learning rate reaches better result[5] as quoted here,

> *1. randomizing the order of input sentences, or*
>
> *2. manually controlling the learning rate over the course of several iterations.*

but it didn't happen on my experiment. As I manually decrease the learning rate (code down below), I found out doc2vec model was not able to infer most similar doc correctly. That says, if I feed in the *doc vector* from the same doc, `self.test_orig_doc_infer()` didn't return that same doc as the most similar doc, while it's supposed to do so.

A side note, the `self.test_orig_doc_infer()` method is used to test if the predicted doc given the *doc vector* from the original doc really return the same doc as most similar doc. If so, we can fairly judge that the model successfully captures the hidden meaning of whole doc, and thus giving representative doc vector.

```
# Failed Attempt (Not achieving better result.)
```

```
for _ in range(fixed_lr_epochs):
   self.model.train(utils.shuffle([x for x in self.docs]),
                total_examples=len(self.docs),
                epochs=1)
   self.model.alpha -= 0.002
   self.model.min_alpha = self.model.alpha  # fixed learning rate
```

Therefore, instead, just leave the default setting is suffice to achieve better result. Here, the learning rate is set 0.025, training epochs is 100 and negative sampling is applied.

```
from UtilWordEmbedding import DocModel

# Configure keyed arguments for Doc2Vec model.
dm_args = {
    'dm': 1,
    'dm_mean': 1,
    'vector_size': 100,
    'window': 5,
    'negative': 5,
    'hs': 0,
    'min_count': 2,
    'sample': 0,
    'workers': workers,
    'alpha': 0.025,
    'min_alpha': 0.025,
    'epochs': 100,
    'comment': 'alpha=0.025'
}

# Instantiate a pv-dm model.
dm = DocModel(docs=all_docs.tagdocs, **dm_args)

dm.custom_train()
```

(5) Labels

And finally, don't forget the LABELS!!!

```
target_labels = all_docs.labels
```

. . .

Prepare the Classification Model

Now, we've prepared all the necessary ingredients — different types of features. Let's experiment to observe their effect on classification performance. Here, I'll use **basic logistic model** as the base model and feed in different kind of features created earlier. Hence, to compare their effectiveness.

In addition to compare effects of each word embedding averaging method, I also try to **concatenate word2vec and doc2vec** together, and see if it can boost up the performance even more.

I used TF-IDF weighted word embedding and PV-DM doc2vec together. The result shows that it increases the accuracy on training dataset (perhaps a sign of over-fitting?), but not so significant improvement on testing dataset compared using TF-IDF word2vec alone.

Reflections

Let's inspect which word embedding performs the worst. Surprisingly, the pre-train GloVe word embedding and doc2vec perform relatively worse on text classification, with accuracy of 0.73 and 0.78 respectively, while other are above 0.8. Perhaps, it's because the custom trained word2vec is specifically fitted for this dataset, and thus provides most relevant information to the docs at hand.

It doesn't necessarily mean that we should not use GloVe word embedding or doc2vec anymore, for in the phase of inference, we might probably run into new words that haven't had word embedding in our word model. In this case, GloVe word embedding would be a great help for its coverage on wide vocabulary.

As for doc2vec, we could say that it can assist the trained word embedding to further boost up the performance of text classification model, though

pretty small and fine to opt-out as well.

WordEmbedding Method	F1 Score - Training	F1 Score - Testing	Accuracy - Training	Accuracy - Testing
Mean Word2vec	0.82	0.81	0.82	0.81
Tf-Idf Mean Word2vec	0.82	0.81	0.82	0.81
GloVe Mean Word2vec	0.72	0.71	0.73	0.72
PV-DM Doc2vec	0.79	0.78	0.79	0.78
Tf-Idf Word2vec + Doc2vec	0.84	0.81	0.85	0.82

Table of Classification Performance over Different Word Embedding

The full jupyter notebook can be found under this link.

. . .

Reference

[1] Susan Li, Multi-Class Text Classification with Doc2Vec & Logistic Regression (2018), *Towards Data Science*

[2] nadbor, Text Classification With Word2Vec (2016), *DS lore*

[3] Susan Li, Multi-Class Text Classification Model Comparison and Selection (2018), *Towards Data Science*

[4] Gensim Doc2Vec Tutorial on the IMDB Sentiment Dataset (2018), *github*

[5] Radim Řehůřek's, Doc2vec tutorial (2014), *Rare Technologies*

NLP Word2vec Doc2vec Word Embeddings Text Classification