Sentence2Vec

November 20, 2018

An baseline python3 implementation of "A Simple But Tough to Beat Baseline for Sentence Embeddngs" by Sanjeev Arora, Yingyu Liang, Tengyu Ma.

Significantly modified from https://github.com/peter3125/sentence2vec.git, which is under the Apache license.

This is a method for quickly deriving sentence embeddings by simpy taking the weighted average of individual word embeddings in a sentence and normalizing them by a 'smooth inverse frequency' (SIF). This SIF is a function of the constant `a` parameter and each word's individual frequency.

In this way, we don't have to encode sentence embeddings directly, a process which would be a lot more expensive than encoding word embeddings. After Arora et al., if you've already got a serialized word vector model, you also already have a sentence embedding model.

It's not much more work to create a framework to compare the similarity of an input sentence to a collection of other sentences in a corpus. For our purposes we're comparing an input utterance against a set of questions. The most similar question in this set of questions is matched to a corresponding answer.

```
In [1]: # import libraries and load english language model and word vector model
        from __future__ import print_function
        import time

        import gensim
        from gensim.corpora.wikicorpus import WikiCorpus
        from gensim.models.doc2vec import Doc2Vec
        import spacy

        import math
        import numpy as np
        import pandas as pd

        from sklearn.decomposition import PCA
        from sklearn.metrics.pairwise import cosine_similarity # <<< this isn't being used!
        from typing import List

        nlp = spacy.load('en')

        gnews_model = gensim.models.KeyedVectors.load_word2vec_format('~/Downloads/GoogleNews-
```

We create `Word` object with `text` and `vector` attributes. A `Sentence` object is merely a list

of `Word` objects, with some additional functions for exposing its orthographic representation and length.

```
In [6]: class Word:
            def __init__(self, text, vector):
                self.text = text
                self.vector = vector


        class Sentence:
            def __init__(self, word_list):
                self.word_list = word_list
            # return the length of a sentence
            def len(self):
                return(len(self.word_list))
            def __str__(self):
                word_str_list = [word.text for word in self.word_list]
                return ' '.join(word_str_list)
            def __repr__(self):
                return self.__str__()
```

This helper function takes a list of strings, and converts them into a list of `Sentence` objects (themselves lists of `Words`, which are just tuples of `text`/`vector` data). This format is necessary to pass sentences to the `sentence2vec` algorithm.

```
In [7]: def preloading_sentences(sentence_list, model):
            """
            Converts a list of sentences into a list of Sentence (and Word) objects

            input: a list of sentences, embedding_size
            output: a list of Sentence objects, containing Word objects, which contain 'text' a
            """
            embedding_size = 300
            all_sent_info = []
            for sentence in sentence_list:
                sent_info = []
                spacy_sentence = nlp(sentence)
                for word in spacy_sentence:
                    if word.text in model.vocab:
                        sent_info.append(Word(word.text, model[word.text]))
                # todo: if sent_info > 0, append, else don't
                all_sent_info.append(Sentence(sent_info))
            return(all_sent_info)
```

We need to use some proxy for word frequency, in order to find the smooth inverse frequency. Here we're using the count from the Google News word2vec model. Note that this model doesn't accurately represent our own financial services domain text - but it still works.

```
In [8]: def get_word_frequency(word_text, vec_model):
            wf = vec_model.vocab[word_text].count
            return(wf)
```

This is the main sentence 'vectorization' function. Except all we're really doing here is iterating over words in sentences and building a list of word vector lists. Then we normalize it by a 'smooth inverse frequency', and take its first principal component. We're also padding our sentences with a lot of zeroes to fit them to the size of the embedding model. By using our own word vector model, this should be a lot more accurate.

```
In [9]: def sentence_to_vec(sentence_list, embedding_size, a=1e-3):
            """
            A SIMPLE BUT TOUGH TO BEAT BASELINE FOR SENTENCE EMBEDDINGS

            Sanjeev Arora, Yingyu Liang, Tengyu Ma
            Princeton University
            """
            sentence_set = [] # intermediary list of sentence vectors before PCA
            sent_list = [] # return list of input sentences in the output
            for sentence in sentence_list:
                this_sent = []
                vs = np.zeros(embedding_size) # add all w2v values into one vector for the sen
                sentence_length = sentence.len()
                for word in sentence.word_list:
                    this_sent.append(word.text)
                    word_freq = get_word_frequency(word.text, gnews_model)
                    a_value = a / (a + word_freq) # smooth inverse frequency, SIF
                    vs = np.add(vs, np.multiply(a_value, word.vector)) # vs += sif * word_vect
                vs = np.divide(vs, sentence_length) # weighted average, normalized by sentence
                sentence_set.append(vs) # add to our existing re-caculated set of sentences
                sent_list.append(' '.join(this_sent))
            # calculate PCA of this sentence set
            pca = PCA(n_components=embedding_size)
            pca.fit(np.array(sentence_set))
            u = pca.components_[0]  # the PCA vector
            u = np.multiply(u, np.transpose(u))  # u x uT
            # pad the vector? (occurs if we have less sentences than embeddings_size)
            if len(u) < embedding_size:
                for i in range(embedding_size - len(u)):
                    u = np.append(u, 0)
            # resulting sentence vectors, vs = vs -u * uT * vs
            sentence_vecs = []
            for vs in sentence_set:
                sub = np.multiply(u, vs)
                sentence_vecs.append(np.subtract(vs, sub))
            return(sentence_vecs, sent_list)
```

This function loads a list of sentences and returns a list of sentence vectors and text.

```
In [10]: def get_sen_embeddings(sentence_list):
             """
             Create Sentence and Word objects from a list and pass them to sentence_to_vec()
             Return a list of _sentence embeddings_ for all sentences in the list
             """
             embedding_size = 300
             all_sent_info = preloading_sentences(sentence_list, gnews_model)
             sentence_vectors, sent_list = sentence_to_vec(all_sent_info, embedding_size)
             return(sentence_vectors, sent_list)
```

This `sklearn` function takes a list of lists (for each sentence, we have a list of sentence embeddings reprepsenting their relative positions in non-Euclidean space).

```
In [11]: def get_cos_distance(sentence_list):
             """
             Create Sentence and Word objects from a list and pass them to sentence_to_vec()
             Return a matrix of the _cosine distance_ of elements in the matrix
             This is used for sentence similarity functions

             input: A list of plaintext sentences
             output: A list of sentence distances
             """
             sentence_vectors, sent_list = get_sen_embeddings(sentence_list)
             cos_list = cosine_similarity(sentence_vectors, Y=None, dense_output=True)
             return(cos_list, sent_list)
```

This simple function just looks up the cosine similarity matrix for the last sentence in the list (this is the input sentence), and returns the index of the most similiar question sentence to it.

```
In [12]: def get_most_similar(utterance, sentence_list):
             """
             Takes an input utterance and a corpus sentence_list to compare it to,
             and returns a dict of the utterance, closest question, and relevant answer
             """
             sentence_list.append(utterance)
             cos_list, sent_text = get_cos_distance(sentence_list)
             # check out the similarity matrix for the utterance
             tmp_list = list(cos_list[len(cos_list)-1])
             # get the index of the question with the highest simliarity
             tmp_indx = tmp_list.index(max(tmp_list[0:len(tmp_list)-2]))
             return(tmp_indx)
```

This interactive demo reads an input `utterance`, appends it to the list of questions. Then it builds a sentence embedding model of this larger list, derives cosine distances between all sentences, and returns the index of the closest question to the input utterance. Then we print the question itself, and its corresponding answer.

```
In [17]: utterance = "I have a shortage :("
```

```
In [18]: faq_csv = pd.read_csv("./utterances_test.csv")
         sentence_list = list(faq_csv['utterance'])
         answer_list = list(faq_csv['answer'])
         max_idx = get_most_similar(utterance, sentence_list)
         sentence_list[max_idx]

Out[18]: "What's a shortage?"

In [19]: answer_list[max_idx]

Out[19]: 'If the funds in your escrow account are projected to be below your minimum balance at
```