Deep Neural Network training with GloVe embeddings

April 28, 2020

Author: Nguyen Luong

1 Abstract

Word embeddings are the heart of many Natural Language Processing (NLP) tasks. They open the possibility to resolve many classic NLP problems: sentiment analysis, documents translation, semantic checking to name a few. In this project, we will demonstrate how to develop our own embeddings from a dataset and how to apply a pre-trained embeddings in a recurrent neural network (RNN). Additionally, we will use our trained networks to solve a sentiment analysis problem and evaluate some documents similarity.

2 Introduction

We will train 2 RNNs in this project. The first one is a Vanilla network which we will train from scratch with our own corpus. The secone one is a pre-trained RNN using GloVe embeddings [1]. The corpus we are using in this project is an IMDB reviews dataset. The dataset consists of a plethora of film reviews and their corresponding sentiments. The reviews are in raw format and thus require some pre-processing before they can be put into use. We will use our models to evaluate a binary classification problem: analysing a review's sentiment. Additionally, our embedding vectors will be assessed with a nearest neighbour check. Given the limitation in size of our corpus, we decided not to perform an analogical reasoning task on our vector space since they will likely fail anyway. Instead, we will conduct the check on the GloVe embeddings. Finally, a document similarity task is evaluated on the Glove embeddings.

3 Data analysis

3.1 IMDB Reviews dataset and Preprocessing

The IMDB data set was acquired from a Kaggle competition. Overall, the dataset consists of 50000 movie reviews from IMDB and their corresponding sentiments. The sentiment can be whether positive or negative. Additionally, the sentiments ratio is 50-50 with half the reviews labelled as positive and the other half is negative. This is particularly useful for training since we will not have to deal with the imbalance in the dataset.

For training procedure, we will encode the sentiment values, assigning 1s to positives and 0s to negatives.

Let's take a quick glance at the dataset.

```
Dataset summary
positive    25000
negative    25000
Name: sentiment, dtype: int64
```

Furthermore, we will checkout the sample length distribution. This is done as a prepration task for training our models. Knowing the average length will help us later when choosing a suitable fixed length for our samples [2].

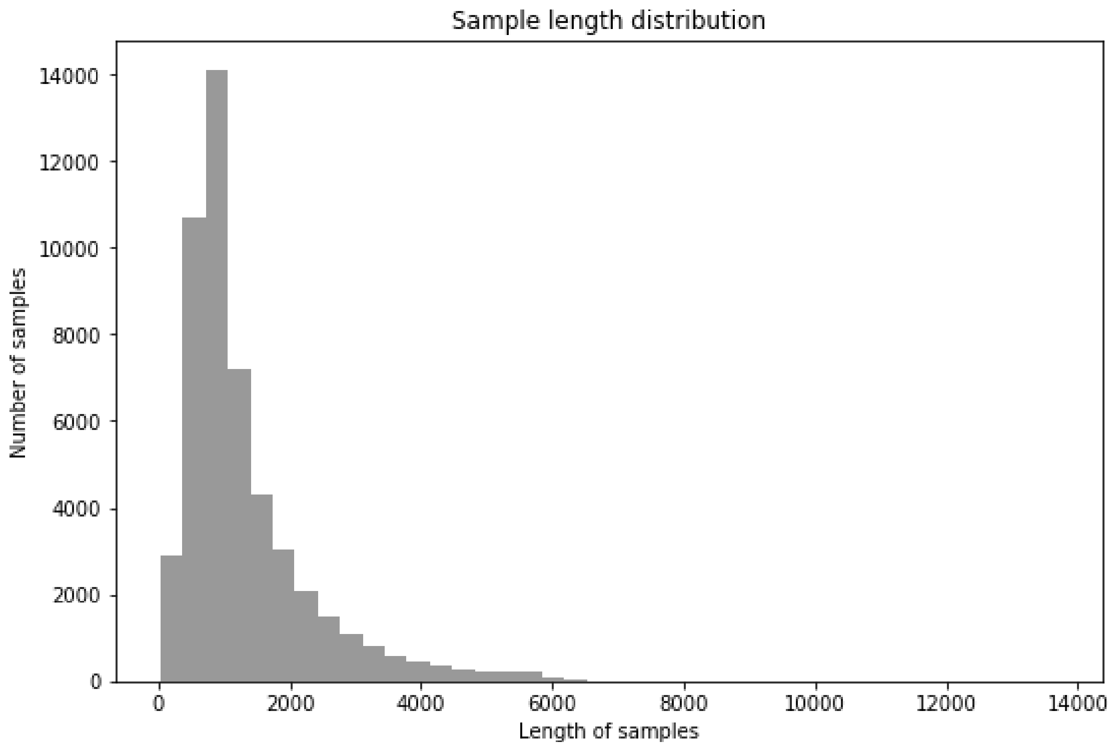

Figure 1: Reviews' length distribution in IMDB dataset

The reviews have an average length of 500 words. This means that we can cover the majority of reviews with a 500D vector space. However, we also need to check out the number of unique words and their frequencies.

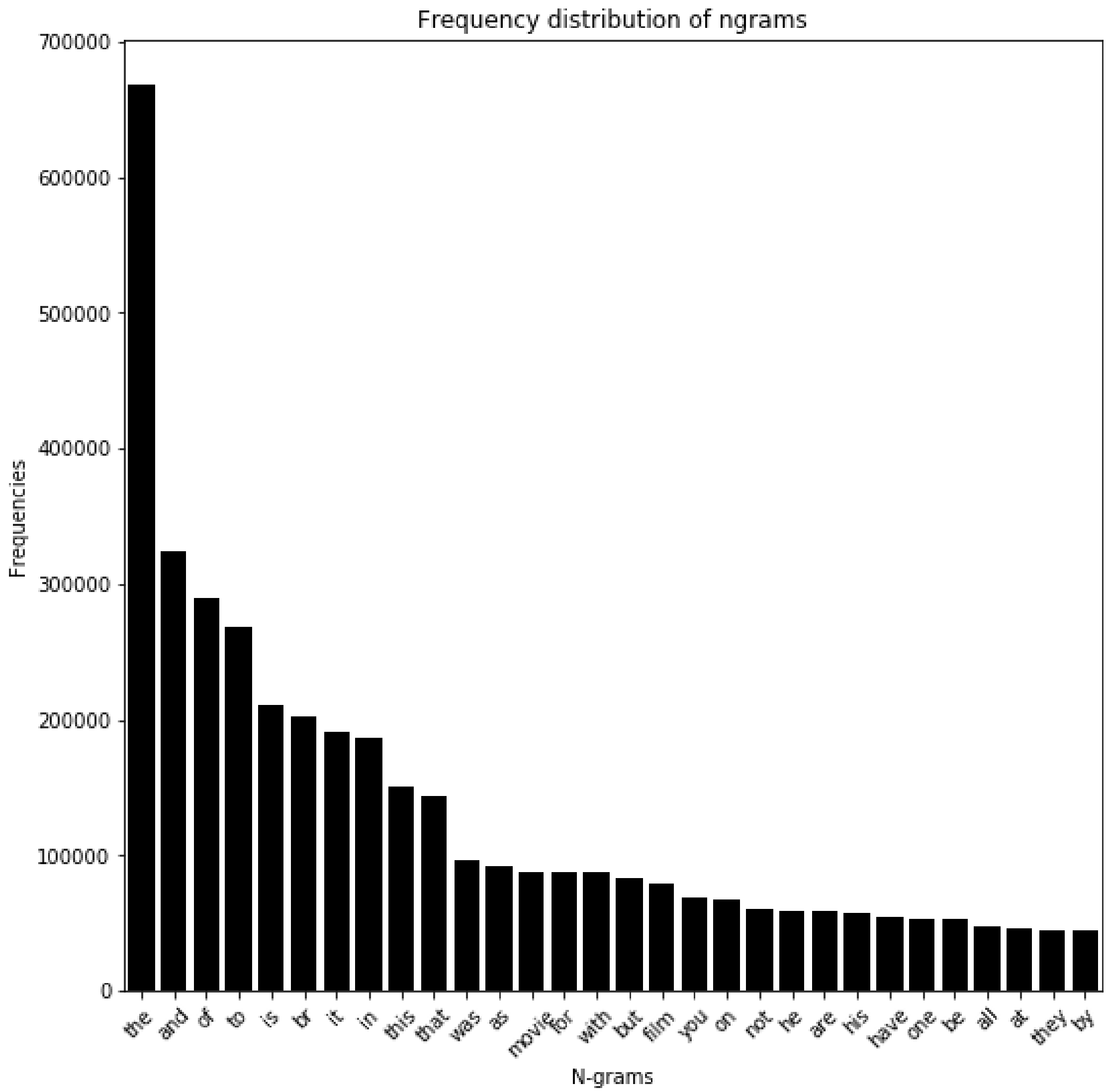

Figure 2: N-grams distribution in IMDB dataset

As seen from the plot above, the majority of words are stopwords. Those bring no value to our model since they do not represent the sentence's context. Using gensim preprocessing and Keras utilities, we will load the reviews corpus into a dictionary and tokenize them. The process consists of the following steps:

1. Simple preprocessing: excluding stopwords, punctuations, removing tags ("<i>", "
") etc.
2. Tokenization: assign each word with an id.
3. Sequence padding: this step is needed for the model training. As observed, the average length of reviews is 500, hence we will set a max length of 500 and pad the sequences with 0 or truncate the sequences whose lengths are greater than max value.

Now, let's demonstrate the result of our process. Notice that we have removed all the stopwords, punctuations and tags from the original reviews. The resulting sequence is much shorter than the original while the words order is also maintained.

```
Review: Basically there's a family where a little boy (Jake) thinks there's a zombie
in his closet & his parents are fighting all the time.\<br/>\<br/>This movie
is slower than a soap opera... .

Preprocessed review: ['basically', 's', 'family', 'little', 'boy', 'jake', 'thinks',
's', 'zombie', 'closet', 'parents', 'fighting',
'time', 'movie', 'slower', 'soap', 'opera', ...]

Sequence: [ 462 1 79 26 215 3053 991 1 743 3910 537 740 ... 0 0]
```

3.2 GloVe embeddings

GloVe is a pre-trained word embeddings developed by Stanford researchers which can be acquired from here. Upon unarchiving, we can see that there are 4 different files, each file contains the embeddings for 400000 words with different vector size. We will use the embeddings with 200 dimensions to train the model.

Loaded 400000 word vectors.

Some embeddings samples, only 5 first elements are shown:

```
film: [-0.0715  0.0934  0.0237 -0.0903  0.0561]
wonderful: [-0.071 0.093 0.023 -0.0903 0.0561]
terible: [-0.0715  0.0934  0.0237 -0.0903 0.0561]
```

We will then assign the weights to our vocabulary. The results will be a weights matrix of size (vocab_size , 200), in which vocab_size being the number of distinct tokens from our processed sequences.

3.3 Train/Test splitting

Let's split our data into 2 different sets: training and test with a ratio of 80-20.

```
Training samples: 40000
Test samples: 10000
```

4 Models

Using Keras, we will perform a sentiment analysis on the IMDB dataset with a Recurrent Neural Network. There are 2 approaches to tackle this task. Firstly, we can use our processed corpus to train the Embedding layer of the network. Secondly, we can simply use the pre-trained GloVe embeddings to pass into the layer. Typically, training on top of a pre-trained embeddings often leads to a better result but we can evaluate both scenarios to see what works best for our problem.

4.1 Original model

4.1.1 Setup

This model takes as input the padded corpus and train its embedding layer. We will use a Sequential Keras model with the following layers: 1. Embedding layer: this layer will take in the padded corpus and assign weights to the token. The layer will output the dense vectors for all the words in our corpus. 2. Flatten layer: simply flatten the (vocab size, embedding dimesion) matrix to a 1D (vocab size*embedding dimesion). This step prepares an approriate input for the Dense layer. 3. Dense layer: Output a 1D vector which we will use to evaluate against the true labels.

Since we are tackling a binary classification problem, the binary-cross entropy loss is employed to compute the output loss. It simply computes the probability of an output and determine whethere this review is a positive one or not. The formula is as follows:

$-(ylog(p) + (1-y)log(1-p))$

in which:

p: probability this review is positive

y: binary indicator (1 for positive, 0 for negative)

The model's summary is displayed below:

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 500, 200)          20906800
_________________________________________________________________
flatten_1 (Flatten)          (None, 100000)            0
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 100001
=================================================================
Total params: 21,006,801
Trainable params: 21,006,801
Non-trainable params: 0
_________________________________________________________________
None
```

The number of trainable parameters is really big because our embeddings layer has not learnt anything and thus needs to train our input vectors. Training the model on GPU is recommended.

4.1.2 Evaluation

Evaluating the model on train data, we achieve an accuracy of 99.9%. The model performs particularly well on test data with an accuracy of around 84%. The ROC curve reveals the same information with the test curve being close to the middle baseline.

```
Train accuracy: 99.997503
Test accuracy: 84.289998
```

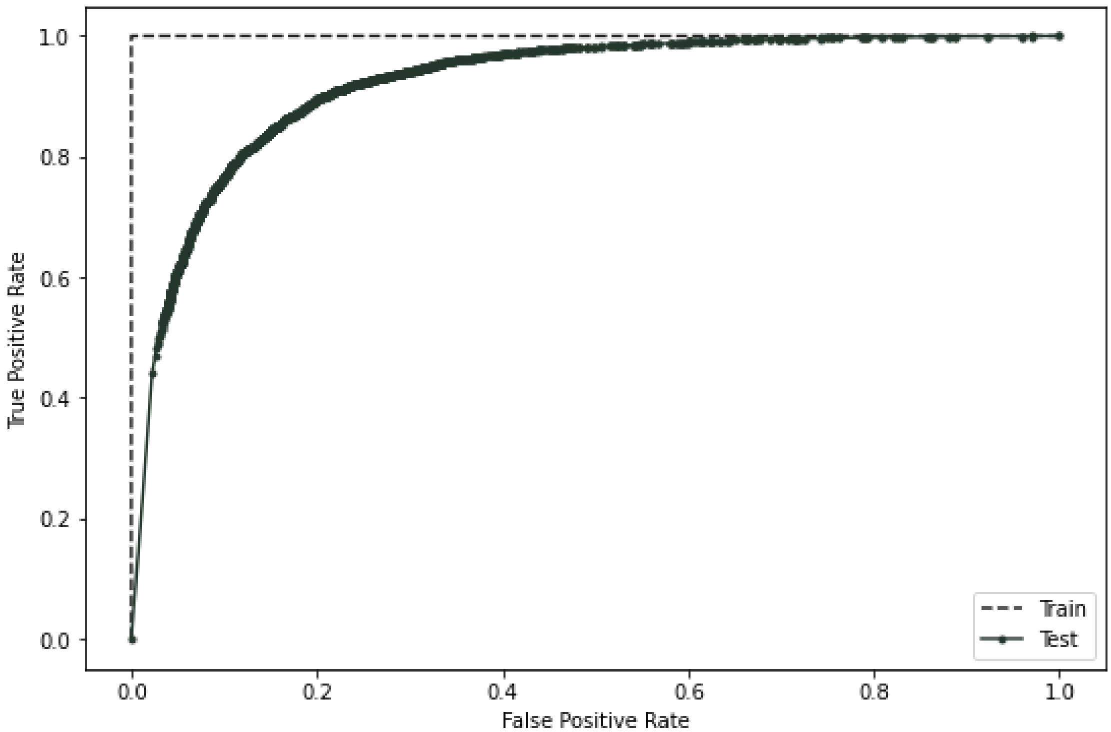

Figure 3: ROC curve for Vanilla RNN model

4.1.3 Nearest neighbours

We have acquired a set of embeddings after training the model. Let's perform some sanity checks to see if they are semantically meaningful. We will evaluate the words' similarity using cosine similarity.

The cosine similarity is defined as:

$$\cos(\mathbf{t}, \mathbf{e}) = \frac{\mathbf{te}}{\|\mathbf{t}\|\|\mathbf{e}\|} = \frac{\sum_{i=1}^{n} \mathbf{t}_i \mathbf{e}_i}{\sqrt{\sum_{i=1}^{n} (\mathbf{t}_i)^2} \sqrt{\sum_{i=1}^{n} (\mathbf{e}_i)^2}}$$

Intuitively, the cosine similarity represents the angle between 2 vectors in a vector space. The closer the value to 1, the more similar the 2 vectors, in terms of orientation (moving in the same direction). That said, we will check the closest neighbours of 3 random words from the corpus.

```
Word similarities:

movie: [('gods', 0.44475925),
        ('ladies', 0.39626837),
        ('newest', 0.39573383)]

actor: [('assailant', 0.48430166),
        ('radko', 0.47733304),
        ('agatha', 0.45624617)]
```

```
comedy: [('conveying', 0.43724343),
        ('binging', 0.37571022),
        ('tribeca', 0.3715757)]
```

We can see that the neighbour words are not particularly meaningful. This is mostly due to our limited vocabulary with only 100000 tokens. However, we can easily improve the embeddings by training our model on a much larger corpus and a deeper network.

4.2 Pre-trained GloVe model

Using the above architecture, we will again train our model with the dataset. However, instead of making the models learn the embeddings, we will apply the pre-trained GloVe models in the Embedding layer

4.2.1 Setup

The model's summary is displayed below:

```
Model: "sequential_3"

_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_3 (Embedding)      (None, 500, 200)          20906800
_________________________________________________________________
flatten_3 (Flatten)          (None, 100000)            0
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 100001
=================================================================
Total params: 21,006,801
Trainable params: 100,001
Non-trainable params: 20,906,800
_________________________________________________________________
None
```

The number of trainable params is significantly lower because we do not have to train the embeddings from scratch. Training time is shortened due to the small amounts of samples as input.

4.2.2 Evaluation

We will perform the same evaluation tasks as we did with our vanilla model. Additionally, we will evaluate an analogical reasoning task (included in the project's evaluation) to make sure that the GloVe embeddings are semantically correct.

```
Train accuracy: 99.742502
Test accuracy: 73.400003
```

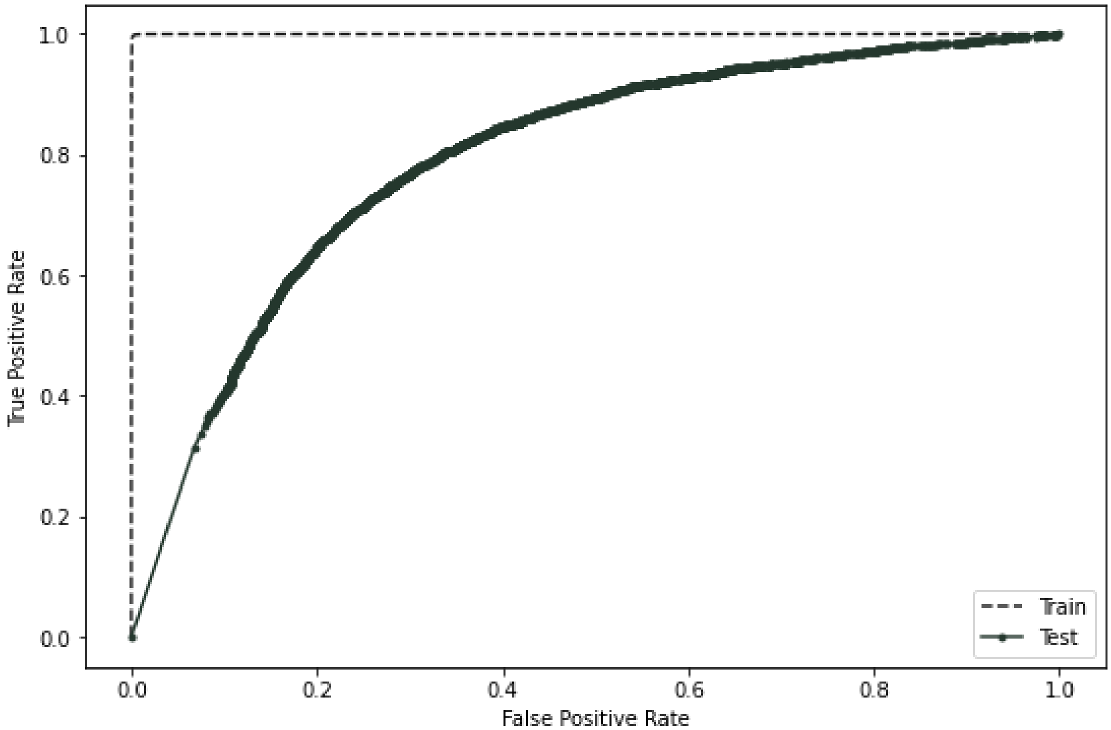

Figure 4: ROC for GloVe RNN model

This time we observed a significant drop in accuracy on the test set. This can be argued that our corpus contains many out-of-vocabulary (OOV) words and the GloVe embeddings fail to capture many of them. More thorough pre-processing may result in a better outcome.

Analogical reasoning We pick out the words that are most relevant in the dataset and take a look at their nearest neighbours.

```
Words similarity:

movie: [('film', 0.88180614), ('movies', 0.8750335), ('films', 0.8414647)]
actor: [('actress', 0.75105745), ('starring', 0.7424192), ('actors', 0.7176447)]
comedy: [('comedies', 0.7667165), ('sitcom', 0.74692607), ('drama', 0.738278)]
```

This time, we can observe that the neighbour words have become more semantically meaningful. GloVe embedding is widely used by researchers so the result does not come as a surprise. We can further evaluate the embeddings on the course's analogical reasoning task. The process is described as follows:

1. Take in 4 words. The embeddings for the 3 first words are denoted as (w1, w2, w3).
2. Compute: w = w2 - w1 + w3
3. Find the closest vector to w. The word associated with the resulting vector should be analogically close to the group, in this case, w4.

```
jordan - amman + baghdad = iraq
switzerland - bern + libreville = gabon
afghanistan - kabul + kigali = rwanda
slovenia - ljubljana + lusaka = zambia
sudan - khartoum + paramaribo = suriname
samoa - apia + baghdad = iraq
eritrea - asmara + bamako = mali
england - london + bern = fribourg
turkmenistan - ashgabat + astana = kazakhstan
belgium - brussels + lima = peru
thailand - bangkok + islamabad = pakistan
uganda - kampala + maputo = mozambique
hungary - budapest + libreville = gabon
greece - athens + oslo = norway
gambia - banjul + harare = zimbabwe
romania - bucharest + copenhagen = denmark
serbia - belgrade + berlin = germany
nigeria - abuja + apia = samoa
iraq - baghdad + belgrade = yugoslavia
vietnam - hanoi + athens = greece
```

We randomized the index and checked 20 random groups. The results were all analogically correct.

5 Document similarity

Another application of word embeddings is the ability to calculate similarity degree between documents. Some popular methods are described in this article. In this project, we will experiment with the cosine distance based method [4]. The idea is that, given the mean vector of the embeddings forming 2 documents, we can compute the cosine distance between them. A distance close to 1 means that the documents are similar in context and vice versa.

```
Sentence 1: I love it
Sentence 2: You hate the food
Similarity degree 1: 0.502

Sentence 1: The actor was great
Sentence 2: He was a superstar
Similarity degree 1: 0.441

Sentence 1: I went to bed yesterday
Sentence 2: I was sleeping the day before
Similarity degree 1: 0.709
```

Although the results were not entirely coherent, we still can observe some meaningful insights. For example, sentences with opposite meaning will not be considered similar. On the other hand, setences representing same context with different wordings have a quite feasible degree of similarity.

6 Conclusion

In this project, we have explained the process of training a word embeddings space with a dataset using an RNN. Our embeddings were also evaluated against the GloVe embddings with a sentiment analysis problem and an analogical reasoning tasks. As a result, our embeddings proved to perform better in the former task but did not fare well in the latter. We argue that the success of our embeddings in the sentiment analysis problem was due to the specificity of our corpus, which closely corressponds to the test data. Better result can be achieved with the pre-trained RNN if we carefully review the dataset pre-processing step and try to reduce the number of OOVs. On the other hand, we can improve the semantic meaningfulness of our embeddings with a much bigger corpus and probably a more complex network. Additionally, we did an experiment with estimating the documents similarity using consine distance between 2 documents. The results were not entirely feasible but did provide some good insight.

7 References

[1] Jeffrey Pennington, Richard Socher, Christopher D. Manning. GloVe: Global Vectors for Word Representation. URL

[2] IMDB Reviews with Keras. URL

[3] Machine Learning Mastery. How to use Word Embedding Layers for Deep Learning with Keras. URL

[4] Adrien Sieg: Text Similarities: Estimate the degree of similarity between two texts. URL

8 Appendix

[ ]:
```
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
from keras.datasets import imdb
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers import Embedding

import gensim
import numpy as np
import pandas as pd

from gensim import corpora
from gensim.utils import simple_preprocess
from gensim.parsing.preprocessing import preprocess_string, strip_short,␣
 ↪strip_tags, remove_stopwords, strip_punctuation,strip_numeric,␣
 ↪strip_multiple_whitespaces
from gensim import models

from pprint import pprint  # pretty-printer
```

```
from collections import defaultdict

from sklearn import preprocessing
from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_curve
from sklearn.metrics import roc_auc_score

import seaborn as sns
import matplotlib.pyplot as plt
from operator import itemgetter
from sklearn.feature_extraction.text import CountVectorizer

from numpy import asarray
from numpy import zeros
import random
```

[ ]:
```
# Loading dataset
imdb = pd.read_csv('imdb.csv')

print(imdb.sentiment.value_counts())
# converting type of columns to 'category'
imdb['sentiment'] = imdb['sentiment'].astype('category')
# Assigning numerical values and storing in another column
imdb['sentiment'] = imdb['sentiment'].cat.codes
```

[ ]:
```
# Figure plotting
pal = sns.dark_palette("seagreen")
sns.set_palette(pal)

plt.figure(figsize=(9, 6))
sns.distplot([len(sample) for sample in list(imdb['review'])], kde=False,␣
↪bins=40, color="black")
plt.xlabel('Length of samples')
plt.ylabel('Number of samples')
plt.title('Sample length distribution')
plt.show()
```

[ ]:
```
#Code taken from https://www.kaggle.com/irinaabdullaeva/
↪imdb-reviews-with-keras#Build-the-model

def counts_word_freq(corpus):

    # Note that `ngram_range=(1, 1)` means we want to extract Unigrams, i.e.␣
↪tokens.
    ngram_vectorizer = CountVectorizer(analyzer='word')
    # X matrix where the row represents sentences and column is our one-hot␣
↪vector for each token in our vocabulary
```

```
    X = ngram_vectorizer.fit_transform(corpus)

    # Vocabulary

    all_ngrams = ngram_vectorizer.get_feature_names()
    num_ngrams = min(30, len(all_ngrams))
    all_counts = X.sum(axis=0).tolist()[0]

    all_ngrams, all_counts = zip(*[(n, c) for c, n in sorted(zip(all_counts,␣
 ↪all_ngrams), reverse=True)])
    ngrams = all_ngrams[:num_ngrams]
    counts = all_counts[:num_ngrams]

    idx = np.arange(num_ngrams)

    return idx, ngrams, counts


idx, ngrams, counts = counts_word_freq(list(imdb["review"]))

# Let's now plot a frequency distribution plot of the most seen words in the␣
 ↪corpus.
plt.figure(figsize=(9, 9))
sns.barplot(idx, counts, color="black")
plt.xlabel('N-grams')
plt.ylabel('Frequencies')
plt.title('Frequency distribution of ngrams')
plt.xticks(idx, ngrams, rotation=45)
plt.show()
```

[ ]:
```
#Code taken and modified from https://machinelearningmastery.com/
 ↪use-word-embedding-layers-deep-learning-keras/
# Returns the padded corpus, tokenizer and size of vocabulary
FILTERS = [lambda x: x.lower(), strip_short, strip_tags, strip_punctuation,␣
 ↪remove_stopwords, strip_multiple_whitespaces]
def preprocessing(X, max_length=500):

    # Tokenize the corpus
    corpus = [preprocess_string(doc, FILTERS) for doc in X]

    # Create a toknenizer and fit on text
    tokenizer = Tokenizer()
    tokenizer.fit_on_texts(corpus)

    # Size of corpus' vocabulary
    vocab_size = len(tokenizer.word_index) + 1
    # integer encode the documents
```

```
    encoded_docs = tokenizer.texts_to_sequences(corpus)
    padded_corpus = pad_sequences(encoded_docs, maxlen=max_length,␣
 ↪padding='post')

    return padded_corpus, vocab_size, tokenizer

padded_corpus, vocab_size, tokenizer = preprocessing(imdb["review"].values)
```

[ ]:
```
# Example of padded sequence
sen = imdb["review"].values[3]
print("Review: {}\n".format(sen))
print("Preprocessed review: {}\n".format(preprocess_string(sen, FILTERS)))
print("Sequence {}\n".format(padded_corpus[3][:25]))
```

[ ]:
```
embeddings_index = dict()
f = open('glove.6B.200d.txt')
for line in f:
    values = line.split()
    word = values[0]
    coefs = asarray(values[1:], dtype='float32')
    embeddings_index[word] = coefs
f.close()
print('Loaded %s word vectors.' % len(embeddings_index))
```

[ ]:
```
print("{}: {}".format("film",embeddings_index["the"][0:5]))
print("{}: {}".format("wonderful",embeddings_index["the"][0:5]))
print("{}: {}".format("terible",embeddings_index["the"][0:5]))
```

[ ]:
```
# Assigning weights to our tokens

tokenizer.word_index.items()
vocab_size = len(tokenizer.word_index) +1
weights = np.zeros((vocab_size, 200), dtype=float)

for word, i in tokenizer.word_index.items():
    embedding_vector = embeddings_index.get(word)
    if embedding_vector is not None:
        weights[i] = embedding_vector
```

[ ]:
```
#Train test splitting

trains = padded_corpus
targets = imdb["sentiment"]

X_train,X_test, y_train, y_test = train_test_split(trains, targets, test_size=0.
 ↪2)
```

```
print("Training samples:", len(X_train))
print("Test samples:", len(X_test))
```

[ ]:
```
#Vanilla Sequence model
#Credit: https://machinelearningmastery.com/
↪use-word-embedding-layers-deep-learning-keras/
EMBEDDING_DIM = 200
MAX_LENGTH = 500

# define model
orig_model = Sequential()
e = Embedding(vocab_size, EMBEDDING_DIM, input_length=MAX_LENGTH)
orig_model.add(e)
orig_model.add(Flatten())
orig_model.add(Dense(1, activation='sigmoid'))
# compile the model
orig_model.compile(optimizer='adam', loss='binary_crossentropy',␣
↪metrics=['accuracy'])
# summarize the model
print(orig_model.summary())
# fit the model
history = orig_model.fit(X_train, y_train, epochs=50, verbose=0)

# serialize model to JSON
orig_model_json = orig_model.to_json()
with open("orig_model.json", "w") as json_file:
    json_file.write(orig_model_json)
# serialize weights to HDF5
orig_model.save_weights("orig_model.h5")
print("Saved model to disk")
```

[ ]:
```
# evaluate the model
loss, accuracy = orig_model.evaluate(X_train, y_train, verbose=0)
print('Train accuracy: %f' % (accuracy*100))
loss, accuracy = orig_model.evaluate(X_test, y_test, verbose=0)
print('Test accuracy: %f' % (accuracy*100))
```

[ ]:
```
train_preds = orig_model.predict_proba(X_train)
test_preds = orig_model.predict_proba(X_test)

ns_fpr, ns_tpr, _ = roc_curve(y_train, train_preds)
lr_fpr, lr_tpr, _ = roc_curve(y_test, test_preds)
baseline = ns_probs = [0 for _ in range(len(y_test))]

plt.figure(figsize=(9, 6))
# plot the roc curve for the model
plt.plot(ns_fpr, ns_tpr, linestyle='--', label='Train')
```

```
plt.plot(lr_fpr, lr_tpr, marker='.', label='Test')

# axis labels
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
# show the legend
plt.legend()
# show the plot
plt.show()
```

[ ]:
```
#Words similiarity
embeddings = e.get_weights()[0]
learned_embeddings = {w:embeddings[idx] for w, idx in tokenizer.word_index.
 ↪items()}

def cosine_similarity(src, dst):
    cosine_similarity = np.dot(src, dst)/(np.linalg.norm(src)* np.linalg.
 ↪norm(dst))
    return cosine_similarity

def most_similar(src, embeddings, top=3):
    similarities = []
    for word in embeddings:
        if word != src:
            cos_sim = cosine_similarity(embeddings[src], embeddings[word])
            similarities.append((word, cos_sim))
    return sorted(similarities,key=itemgetter(1), reverse=True)[:top]

def words_analogy(w1,w2,w3, embeddings, top=3):

    if w1 not in embeddings:
        print("Cant find {}".format(w1))
        return "<empty>"
    if w2 not in embeddings:
        print("Cant find {}".format(w2))
        return "<empty>"
    if w3 not in embeddings:
        print("Cant find {}".format(w3))
        return "<empty>"

    similarities = []
    v = embeddings[w1] - embeddings[w2] + embeddings[w3]
    for word in embeddings:
        if word != w1 and word != w2 and word != w3:
            cos_sim = cosine_similarity(v, embeddings[word])
            similarities.append((word, cos_sim))
    return sorted(similarities,key=itemgetter(1), reverse=True)[:top]
```

```
def most_similar_words(vector, embeddings, top=3):
    similarities = []
    for word in embeddings:
        cos_sim = cosine_similarity(vector, embeddings[word])
        similarities.append((word, cos_sim))
    return sorted(similarities,key=itemgetter(1), reverse=True)[:top]

print("Word similarities:\n")
print("movie: {}".format(most_similar("movie", learned_embeddings)))
print("actor: {}".format(most_similar("actor", learned_embeddings)))
print("comedy: {}".format(most_similar("comedy", learned_embeddings)))
```

[ ]:
```
#Pre-trained GloVe Sequence model
#Credit: https://machinelearningmastery.com/
 ↪use-word-embedding-layers-deep-learning-keras/

EMBEDDING_DIM = 200
MAX_LENGTH = 500

# define model
glove_model = Sequential()
e = Embedding(vocab_size, EMBEDDING_DIM, weights=[weights],␣
 ↪input_length=MAX_LENGTH, trainable=False)
glove_model.add(e)
glove_model.add(Flatten())
glove_model.add(Dense(1, activation='sigmoid'))
# compile the model
glove_model.compile(optimizer='adam', loss='binary_crossentropy',␣
 ↪metrics=['accuracy'])
```

[ ]:
```
# summarize the model
print(glove_model.summary())
# fit the model
history = glove_model.fit(X_train, y_train, epochs=50, verbose=0)

# serialize model to JSON
glove_model_json = glove_model.to_json()
with open("glove_model.json", "w") as json_file:
    json_file.write(glove_model_json)
# serialize weights to HDF5
glove_model.save_weights("glove_model.h5")
print("Saved model to disk")
```

[ ]:
```
#Pre-trained Evaluation
# evaluate the model
loss, accuracy = glove_model.evaluate(X_train, y_train, verbose=0)
```

```
print('Train accuracy: %f' % (accuracy*100))
loss, accuracy = glove_model.evaluate(X_test, y_test, verbose=0)
print('Test accuracy: %f' % (accuracy*100))

train_preds = glove_model.predict_proba(X_train)
test_preds = glove_model.predict_proba(X_test)

ns_fpr, ns_tpr, _ = roc_curve(y_train, train_preds)
lr_fpr, lr_tpr, _ = roc_curve(y_test, test_preds)
baseline = ns_probs = [0 for _ in range(len(y_test))]

plt.figure(figsize=(9, 6))
# plot the roc curve for the model
plt.plot(ns_fpr, ns_tpr, linestyle='--', label='Train')
plt.plot(lr_fpr, lr_tpr, marker='.', label='Test')

# axis labels
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
# show the legend
plt.legend()
# show the plot
plt.show()
```

[ ]:
```
# Analogical reasoning

print("Word similarities:\n")
print("movie: {}".format(most_similar("movie", embeddings_index)))
print("actor: {}".format(most_similar("actor", embeddings_index)))
print("comedy: {}".format(most_similar("comedy", embeddings_index)))
```

[ ]:
```
#Nearest neighbours

tasks = pd.read_csv('analogical_reasoning_questions-words.txt', sep=" ",␣
 ↪header=None)
tasks = tasks.apply(lambda x: x.astype(str).str.lower())
tasks.head()

def evaluate(tasks):
    for i in range(0,20):

        n = random.randint(1,3000)
        doc = tasks.values[n]
        w1 = doc[0]
        w2 = doc[1]
        w3 = doc[2]
        result = words_analogy(w2, w1, w3, embeddings_index)
```

```
        if isinstance(result, str):
            print("{} - {} + {} = {}".format(w2, w1, w3, result))
        else:
            print("{} - {} + {} = {}".format(w2, w1, w3, result[0][0]))

evaluate(tasks)
```

[ ]:
```
#Document similarity

def document_similarity(d1, d2, embeddings):

    v1 = np.mean([embeddings[w] for w in preprocess_string(d1, FILTERS) ],␣
 ↪axis=0)
    v2 = np.mean([embeddings[w] for w in preprocess_string(d2, FILTERS)],␣
 ↪axis=0)
    result = cosine_similarity(v1,v2)

    print("Sentence 1: {}".format(d1))
    print("Sentence 2: {}".format(d2))
    print("Similarity degree 1: {:.3f}\n".format(result))
    return result

r = document_similarity("I love it", "You hate the food", embeddings_index)
r = document_similarity("The actor was great", "He was a superstar",␣
 ↪embeddings_index)
r = document_similarity("I went to bed yesterday", "I was sleeping the day␣
 ↪before", embeddings_index)
```