Deep learning for NLP

# Word embeddings

Benoit Favre

21 Feb 2017

## 1 Introduction

Word embeddings are dense representations for words built by factoring a cooccurrence matrix through various means. It has been shown that in this space words that have the same meaning (or at least which occur in the same context) are close according to the cosine similarity, and that some analogy relationships are encoded in transitions.

In this tutorial, you will build word embeddings, and verify those properties. Since training on full scale corpora such as Wikipedia or Twitter can be quite long, you will work on a much smaller corpus extracted from Wikipedia, the text8 corpus which contains 100MB of text already normalized, tokenized and with punctuation removed.

You can download this corpus with the following command

```
wget http://mattmahoney.net/dc/text8.zip
```

## 2 Training GloVe embeddings

There are several implementations available for training word embeddings, one of the most popular is GloVe (for Global Vectors) which is trained so that the dot product between a pair of words is proportional to their probability to occur in the same context.

$$F\left((w_i - w_j)^T \tilde{w}_k\right) = \frac{P(w_k|w_i)}{P(w_k|w_j)} \tag{1}$$

The GloVe software source code can be downloaded at `nlp.stanford.edu/projects/glove/`. Just extract the archive and type `make` in it to build the programs. You can refer to `demo.sh` for an example of how to train a model.

First, we need to create a vocabulary file. We use the `-min-count` parameter to tell it to ignore words which appear less than 5 times:

```
./build/vocab_count -min-count 5 < text8 > text8.vocab
```

Then we build a sparse cooccurrence matrix for this vocabulary. We specify a window size of 15 words (so in the cooccurrence matrix, $m_{i,j}$ is increase by one everytime two words $i$ and $j$ occur in the same 15 word window). Larger window size create more topical representations, and smaller window sizes focus on syntactic aspects. At this point, it is a good idea to create several cooccurrence matrices (and apply the following steps to each of them) with a window of 5, 15, 40.

```
./build/cooccur -vocab-file text8.vocab -window-size 15 < text8 \
    > text8.cooc
```

The cooccurrence matrix needs to be shuffled before training. The memory limit sets how many gigabytes of memory can be used for processing.

```
./build/shuffle -memory 2 < text8.cooc > text8.cooc.shuf
```

Finally, we can train the model. To make a quick model, we will build vectors of size 50 with 5 iterations of training. One important parameter is x-max which calibrates the saturation of the weighting function. Since the corpus is small, words rarely cooccur, so this value needs to be lowered to 10 instead of the default of 100. Finally, the model we produce is a text model so that we can read it later in python.

```
./build/glove -save-file embeddings -threads 4 -input-file \
    text8.cooc.shuf -x-max 10 -iter 5 -vector-size 50 \
    -binary 0 -vocab-file text8.vocab
```

# 3 Loading embeddings

Embeddings are stored in the `embeddings.txt` file. Each line contains a word form, followed by 50 float values of the vector for that word. Loading the embeddings is not difficult in itself. To make computations fast, we store the whole set of embeddings in a numpy array of shape (num words, dimensions). We also build a dictionary (vocab) for mapping words to row index in this array, and another (rev_vocab) for mapping indexes back to word forms.

```
def load(filename):
    vocab = {}
    rev_vocab = []
    lines = open(filename).readlines()
    vectors = np.zeros((len(lines), len(lines[0].split()) - 1))
    for i, line in enumerate(lines):
        tokens = line.strip().split()
        vocab[tokens[0]] = i
        rev_vocab.append(tokens[0])
        vectors[i] = [float(value) for value in tokens[1:]]
    return vocab, rev_vocab, vectors
```

Then, we will compute *cosine* similarities between words:

$$cosine(x, y) = \frac{x^T y}{||x|| \times ||y||} \tag{2}$$

So this computation involves the dot product between the transpose of $x$ and $y$ divided by the product of their L2 norm. The vectors can be normalized so that their norm is 1. It can be done on the whole set of vectors at once with the following function:

```
def normalize(vectors):
    norm = np.linalg.norm(vectors, ord=2, axis=1)
    np.place(norm, norm==0, 1)
    vectors /= norm[:,None]
```

Once the vectors are normalized, computing the similarity between two words is as simple as:

```
    cosine = np.dot(vectors[vocab[word1]], vectors[vocab[word2]])
```

What is the cosine similarity between the vectors representing "dog" and "cat', and what about "dog" and "dentist"? GloVe embeddings do not account for polysemy (words that have multiple senses). So is "bank" closer to "river" or to "trade"? If the training corpus was mostly about nature, the story would be different.

# 4 Closest words

An interesting analysis is to look at the closest words to a given word. This involves computing the cosine similarity of all words in the vocabulary with the target word, and finding the $n$ highest scoring words. The similarities can be computed at once with the dot product between the matrix containing all vectors and the transpose of the target word vector (`np.dot(vectors, v.T)`), then we can use some numpy magic to recover the indices of the $n$ highest scores.

```
def closest(vectors, vector, n=10):
    scores = np.dot(vectors, vector.T)
    indices = np.argpartition(scores, -n)[-n:]
    indices = indices[np.argsort(scores[indices])]
    output = []
    for i in [int(x) for x in indices]:
        output.append((scores[i], i))
    return reversed(output)
```

Given the output of this function, you need to map indices back to words with `rev_vocab`. What are the closest words to "apple"? What about other words? Can you find words which have a strange neighborhood?

# 5 Analogy

Word analogies can be exposed by translating a word vector in a direction that corresponds to a linguistic or semantic relationship between two other words. So if we have $w_1$ and $w_2$ in relation $R(w_1, w_2)$, we can compute the translation $r = vec(w_2) - vec(W_1)$, and then apply this translation to the vector of another word $vec(w_3)$. The word closest to $W_3 + r$ should also exhibit the same relation. Therefore, the idea is to use the `closest` function to find vectors similar to $vec(w_2) - vec(w_1) + vec(w_3)$.

For instance, "paris" is to "france" what "delhi" is to X ? Try other analogies involving semantics (gates - microsoft + apple = ?), gender (king - man + woman = ?), morphological or syntactic relations. What is the impact of the window size on those analogies?

Note that analogies are not very clear here as the dataset the embeddings are trained on is small.

# 6 Evaluation

To evaluate the quality of a set of word embeddings, we will use data from the WS-353[1] dataset. The file `EN-WS-353-REL.txt` contains on each line a pair of words followed by the average of a similarity rating by human judges. The higher the rating, the more similar the judges though the words in the pair are. One way to evaluate the quality of word embeddings is to compute the correlation between the cosine similarity of the pairs and human ratings. Since cosine similarities and judgements are on a different scale, we will use a rank correlation (how similar two rankings are).

To compute Spearman's $\rho$ correlation, you can use `scipy`. The `spearmanr` function takes two lists of values as argument and returns two values. The first one is $\rho$, the second one is the significance of the estimation ($pval$). Here we are only interested in the first one.

```
from scipy import stats
serie1 = [1.3, 2.2, 3.1, 4.0]
serie2 = [4.5, 1.8, 3.2, 2.7]
print(stats.spearmanr(serie1, serie2)[0])
```

You can compute the $\rho$ correlation between human judgements and the word embeddings you have created for different window length. Which window length correlates best with human judgements?

# 7 Extensions

The most obvious extension is to download embeddings from the web (such as those on the GloVe website) trained with much larger datasets and see if the similarity and analogies work better.

[1] Available at `https://github.com/k-kawakami/embedding-evaluation`

In an industrial context, there exist many tricks to efficiently compute the $k$ nearest neighbors of a point according to the dot product, to make it scale to millions of words (or other objects such as images for that matter). Locality sensitive hashing could be a good starting point.