Word representations

A popular idea in modern machine learning is to represent words by vectors. These vectors capture hidden information about a language, like word analogies or semantic. It is also used to improve performance of text classifiers.

In this tutorial, we show how to build these word vectors with the fastText tool. To download and install fastText, follow the first steps of the tutorial on text classification.

Getting the data

In order to compute word vectors, you need a large text corpus. Depending on the corpus, the word vectors will capture different information. In this tutorial, we focus on Wikipedia's articles but other sources could be considered, like news or Webcrawl (more examples here). To download a raw dump of Wikipedia, run the following command:

```
wget https://dumps.wikimedia.org/enwiki/latest/enwiki-latest-pages-articles.xml.bz2
```

Downloading the Wikipedia corpus takes some time. Instead, lets restrict our study to the first 1 billion bytes of English Wikipedia. They can be found on Matt Mahoney's website:

```
$ mkdir data
$ wget -c http://mattmahoney.net/dc/enwik9.zip -P data
$ unzip data/enwik9.zip -d data
```

A raw Wikipedia dump contains a lot of HTML / XML data. We pre-process it with the wikifil.pl script bundled with fastText (this script was originally developed by Matt Mahoney, and can be found on his website)

```
$ perl wikifil.pl data/enwik9 > data/fil9
```

```
anarchism originated as a term of abuse first used against early working class
```

The text is nicely pre-processed and can be used to learn our word vectors.

Training word vectors

Learning word vectors on this data can now be achieved with a single command:

```
$ mkdir result
$ ./fasttext skipgram -input data/fil9 -output result/fil9
```

To decompose this command line: ./fastext calls the binary fastText executable (see how to install fastText here) with the 'skipgram' model (it can also be 'cbow'). We then specify the requires options '-input' for the location of the data and '-output' for the location where the word representations will be saved.

While fastText is running, the progress and estimated time to completion is shown on your screen. Once the program finishes, there should be two files in the result directory:

```
$ ls -l result
-rw-r-r-- 1 bojanowski 1876110778 978480850 Dec 20 11:01 fil9.bin
-rw-r-r-- 1 bojanowski 1876110778 190004182 Dec 20 11:01 fil9.vec
```

The `fil9.bin` file is a binary file that stores the whole fastText model and can be subsequently loaded. The `fil9.vec` file is a text file that contains the word vectors, one per line for each word in the vocabulary:

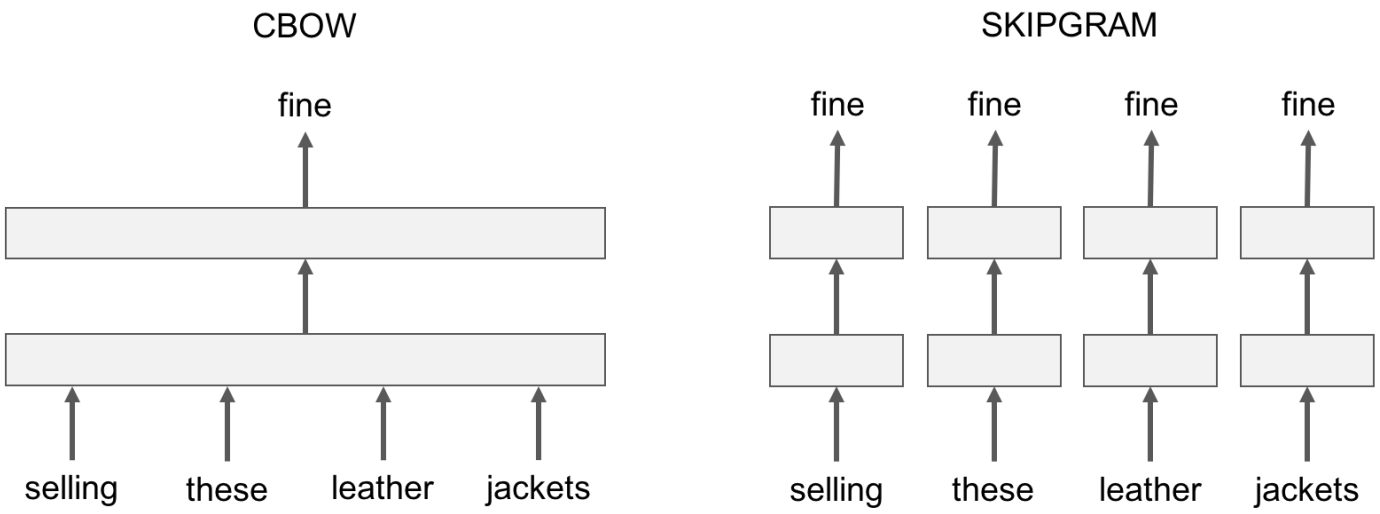
```
$ head -n 4 result/fil9.vec
218316 100
the -0.10363 -0.063669 0.032436 -0.040798 0.53749 0.00097867 0.10083 0.24829 ...
of -0.0083724 0.0059414 -0.046618 -0.072735 0.83007 0.038895 -0.13634 0.60063 ...
one 0.32731 0.044409 -0.46484 0.14716 0.7431 0.24684 -0.11301 0.51721 0.73262 ...
```

The first line is a header containing the number of words and the dimensionality of the vectors. The subsequent lines are the word vectors for all words in the vocabulary, sorted by decreasing

fastText provides two models for computing word representations: skipgram and cbow ('**c**ontinuous-**b**ag-**o**f-**w**ords').

The skipgram model learns to predict a target word thanks to a nearby word. On the other hand, the cbow model predicts the target word according to its context. The context is represented as a bag of the words contained in a fixed size window around the target word.

Let us illustrate this difference with an example: given the sentence *'Poets have been mysteriously silent on the subject of cheese'* and the target word '*silent*', a skipgram model tries to predict the target using a random close-by word, like '*subject*' or 'mysteriously'. *The cbow model takes all the words in a surrounding window, like {been, mysteriously, on, the*}, and uses the sum of their vectors to predict the target. The figure below summarizes this difference with another example.



To train a cbow model with fastText, you run the following command:

```
./fasttext cbow -input data/fil9 -output result/fil9
```

In practice, we observe that skipgram models works better with subword information than cbow.

Advanced readers: playing with the parameters

subwords. The dimension (*dim*) controls the size of the vectors, the larger they are the more information they can capture but requires more data to be learned. But, if they are too large, they are harder and slower to train. By default, we use 100 dimensions, but any value in the 100-300 range is as popular. The subwords are all the substrings contained in a word between the minimum size (*minn*) and the maximal size (*maxn*). By default, we take all the subword between 3 and 6 characters, but other range could be more appropriate to different languages:

```
$ ./fasttext skipgram -input data/fil9 -output result/fil9 -minn 2 -maxn 5 -dim 300
```

Depending on the quantity of data you have, you may want to change the parameters of the training. The *epoch* parameter controls how many time will loop over your data. By default, we loop over the dataset 5 times. If you dataset is extremely massive, you may want to loop over it less often. Another important parameter is the learning rate *-lr*). The higher the learning rate is, the faster the model converge to a solution but at the risk of overfitting to the dataset. The default value is 0.05 which is a good compromise. If you want to play with it we suggest to stay in the range of [0.01, 1]:

```
$ ./fasttext skipgram -input data/fil9 -output result/fil9 -epoch 1 -lr 0.5
```

Finally , fastText is multi-threaded and uses 12 threads by default. If you have less CPU cores (say 4), you can easily set the number of threads using the *thread* flag:

```
$ ./fasttext skipgram -input data/fil9 -output result/fil9 -thread 4
```

Printing word vectors

Searching and printing word vectors directly from the `fil9.vec` file is cumbersome. Fortunately, there is a `print-word-vectors` functionality in fastText.

For examples, we can print the word vectors of words *asparagus, pidgey* and *yellow* with the following command:

```
$ echo "asparagus pidgey yellow" | ./fasttext print-word-vectors result/fil9.bin
asparagus 0.46826 -0.20187 -0.29122 -0.17918 0.31289 -0.31679 0.17828 -0.04418 ...
```

A nice feature is that you can also query for words that did not appear in your data! Indeed words are represented by the sum of its substrings. As long as the unknown word is made of known substrings, there is a representation of it!

As an example let's try with a misspelled word:

```
$ echo "enviroment" | ./fasttext print-word-vectors result/fil9.bin
```

You still get a word vector for it! But how good it is? Let s find out in the next sections!

Nearest neighbor queries

A simple way to check the quality of a word vector is to look at its nearest neighbors. This give an intuition of the type of semantic information the vectors are able to capture.

This can be achieve with the *nn* functionality. For example, we can query the 10 nearest neighbors of a word by running the following command:

```
$ ./fasttext nn result/fil9.bin
Pre-computing word vectors... done.
```

Then we are prompted to type our query word, let us try *asparagus* :

```
Query word? asparagus
beetroot 0.812384
tomato 0.806688
horseradish 0.805928
spinach 0.801483
licorice 0.791697
lingonberries 0.781507
asparagales 0.780756
lingonberry 0.778534
celery 0.774529
beets 0.773984
```

Nice! It seems that vegetable vectors are similar. Note that the nearest neighbor is the word *asparagus* itself, this means that this word appeared in the dataset. What about pokemons?

```
pidgeon 0.787351
pok 0.781068
pikachu 0.758688
charizard 0.749403
squirtle 0.742582
beedrill 0.741579
charmeleon 0.733625
```

Different evolution of the same Pokemon have close-by vectors! But what about our misspelled word, is its vector close to anything reasonable? Let s find out:

```
Query word? enviroment
enviromental 0.907951
environ 0.87146
enviro 0.855381
environs 0.803349
environnement 0.772682
enviromission 0.761168
realclimate 0.716746
environment 0.702706
acclimatation 0.697196
ecotourism 0.697081
```

Thanks to the information contained within the word, the vector of our misspelled word matches to reasonable words! It is not perfect but the main information has been captured.

Advanced reader: measure of similarity

In order to find nearest neighbors, we need to compute a similarity score between words. Our words are represented by continuous word vectors and we can thus apply simple similarities to them. In particular we use the cosine of the angles between two vectors. This similarity is computed for all words in the vocabulary, and the 10 most similar words are shown. Of course, if the word appears in the vocabulary, it will appear on top, with a similarity of 1.

Word analogies

France) and outputs the analogy:

```
$ ./fasttext analogies result/fil9.bin
Pre-computing word vectors... done.
Query triplet (A - B + C)? berlin germany france
paris 0.896462
bourges 0.768954
louveciennes 0.765569
toulouse 0.761916
valenciennes 0.760251
montpellier 0.752747
strasbourg 0.744487
meudon 0.74143
bordeaux 0.740635
pigneaux 0.736122
```

The answer provides by our model is *Paris*, which is correct. Let's have a look at a less obvious example:

```
Query triplet (A - B + C)? psx sony nintendo
gamecube 0.803352
nintendogs 0.792646
playstation 0.77344
sega 0.772165
gameboy 0.767959
arcade 0.754774
playstationjapan 0.753473
gba 0.752909
dreamcast 0.74907
famicom 0.745298
```

Our model considers that the *nintendo* analogy of a *psx* is the *gamecube*, which seems reasonable. Of course the quality of the analogies depend on the dataset used to train the model and one can only hope to cover fields only in the dataset.

Importance of character n-grams

Using subword-level information is particularly interesting to build vectors for unknown words. For example, the word *gearshift* does not exist on Wikipedia but we can still query its closest existing

```
flywheels 0.779804
flywheel 0.777859
gears 0.776133
driveshafts 0.756345
driveshaft 0.755679
daisywheel 0.749998
wheelsets 0.748578
epicycles 0.744268
gearboxes 0.73986
```

Most of the retrieved words share substantial substrings but a few are actually quite different, like *cogwheel*. You can try other words like *sunbathe* or *grandnieces*.

Now that we have seen the interest of subword information for unknown words, let s check how it compares to a model that do not use subword information. To train a model without no subwords, just run the following command:

```
$ ./fasttext skipgram -input data/fil9 -output result/fil9-none -maxn 0
```

The results are saved in result/fil9-non.vec and result/fil9-non.bin.

To illustrate the difference, let us take an uncommon word in Wikipedia, like *accomodation* which is a misspelling of *accommodation*. Here is the nearest neighbors obtained without no subwords:

```
$ ./fasttext nn result/fil9-none.bin
Query word? accomodation
sunnhordland 0.775057
accomodations 0.769206
administrational 0.753011
laponian 0.752274
ammenities 0.750805
dachas 0.75026
vuosaari 0.74172
hostelling 0.739995
greenbelts 0.733975
asserbo 0.732465
```

The result does not make much sense, most of these words are unrelated. On the other hand, using subword information gives the following list of nearest neighbors:

```
accommodative 0.847751
accommodating 0.794353
accomodated 0.740381
amenities 0.729746
catering 0.725975
accomodate 0.703177
hospitality 0.701426
```

The nearest neighbors capture different variation around the word *accommodation*. We also get semantically related words such as *amenities* or *lodging*.

Conclusion

In this tutorial, we show how to obtain word vectors from Wikipedia. This can be done for any language and you we provide pre-trained models with the default setting for 294 of them.

← TEXT CLASSIFICATION

PYTHON MODULE →