Text classification

Text classification is a core problem to many applications, like spam detection, sentiment analysis or smart replies. In this tutorial, we describe how to build a text classifier with the fastText tool.

What is text classification?

The goal of text classification is to assign documents (such as emails, posts, text messages, product reviews, etc...) to one or multiple categories. Such categories can be review scores, spam v.s. non-spam, or the language in which the document was typed. Nowadays, the dominant approach to build such classifiers is machine learning, that is learning classification rules from examples. In order to build such classifiers, we need labeled data, which consists of documents and their corresponding categories (or tags, or labels).

As an example, we build a classifier which automatically classifies stackexchange questions about cooking into one of several possible tags, such as `pot`, `bowl` or `baking`.

Installing fastText

The first step of this tutorial is to install and build fastText. It only requires a c++ compiler with good support of c++11.

Let us start by downloading the most recent release:

```
$ wget https://github.com/facebookresearch/fastText/archive/v0.9.1.zip
$ unzip v0.9.1.zip
```

Move to the fastText directory and build it:

```
$ cd fastText-0.9.1
$ make
```

```
usage: fasttext <command> <args>

The commands supported by fasttext are:

  supervised              train a supervised classifier
  quantize                quantize a model to reduce the memory usage
  test                    evaluate a supervised classifier
  predict                 predict most likely labels
  predict-prob            predict most likely labels with probabilities
  skipgram                train a skipgram model
  cbow                    train a cbow model
  print-word-vectors      print word vectors given a trained model
  print-sentence-vectors  print sentence vectors given a trained model
  nn                      query for nearest neighbors
  analogies               query for analogies
```

In this tutorial, we mainly use the `supervised`, `test` and `predict` subcommands, which corresponds to learning (and using) text classifier. For an introduction to the other functionalities of fastText, please see the tutorial about learning word vectors.

Getting and preparing the data

As mentioned in the introduction, we need labeled data to train our supervised classifier. In this tutorial, we are interested in building a classifier to automatically recognize the topic of a stackexchange question about cooking. Let's download examples of questions from the cooking section of Stackexchange, and their associated tags:

```
>> wget https://dl.fbaipublicfiles.com/fasttext/data/cooking.stackexchange.tar.gz &&
>> head cooking.stackexchange.txt
```

Each line of the text file contains a list of labels, followed by the corresponding document. All the labels start by the `__label__` prefix, which is how fastText recognize what is a label or what is a word. The model is then trained to predict the labels given the word in the document.

Before training our first classifier, we need to split the data into train and validation. We will use the validation set to evaluate how good the learned classifier is on new data.

Our full dataset contains 15404 examples. Let's split it into a training set of 12404 examples and a validation set of 3000 examples:

```
>> head -n 12404 cooking.stackexchange.txt > cooking.train
>> tail -n 3000 cooking.stackexchange.txt > cooking.valid
```

Our first classifier

We are now ready to train our first classifier:

```
>> ./fasttext supervised -input cooking.train -output model_cooking
Read 0M words
Number of words:  14598
Number of labels: 734
Progress: 100.0%  words/sec/thread: 75109  lr: 0.000000  loss: 5.708354  eta: 0h0m
```

The `-input` command line option indicates the file containing the training examples, while the `-output` option indicates where to save the model. At the end of training, a file `model_cooking.bin`, containing the trained classifier, is created in the current directory.

It is possible to directly test our classifier interactively, by running the command:

```
>> ./fasttext predict model_cooking.bin -
```

and then typing a sentence. Let's first try the sentence:

Which baking dish is best to bake a banana bread ?

The predicted tag is `baking` which fits well to this question. Let us now try a second example:

Why not put knives in the dishwasher?

The label predicted by the model is `food-safety`, which is not relevant. Somehow, the model seems to fail on simple examples. To get a better sense of its quality, let's test it on the validation data by running:

```
Number of examples: 3000
```

The output of fastText are the precision at one (`P@1`) and the recall at one (`R@1`). We can also compute the precision at five and recall at five with:

```
>> ./fasttext test model_cooking.bin cooking.valid 5
N  3000
P@5  0.0668
R@5  0.146
Number of examples: 3000
```

Advanced readers: precision and recall

The precision is the number of correct labels among the labels predicted by fastText. The recall is the number of labels that successfully were predicted, among all the real labels. Let's take an example to make this more clear:

Why not put knives in the dishwasher?

On Stack Exchange, this sentence is labeled with three tags: `equipment` , `cleaning` and `knives` . The top five labels predicted by the model can be obtained with:

```
>> ./fasttext predict model_cooking.bin - 5
```

are `food-safety` , `baking` , `equipment` , `substitutions` and `bread` .

Thus, one out of five labels predicted by the model is correct, giving a precision of 0.20. Out of the three real labels, only one is predicted by the model, giving a recall of 0.33.

For more details, see the related Wikipedia page.

Making the model better

The model obtained by running fastText with the default arguments is pretty bad at classifying new questions. Let's try to improve the performance, by changing the default parameters.

crude normalization can be obtained using command line tools such as `sed` and `tr`:

```
>> cat cooking.stackexchange.txt | sed -e "s/\([.\!?,'/()]\)/ \1 /g" | tr "[:upper:]
>> head -n 12404 cooking.preprocessed.txt > cooking.train
>> tail -n 3000 cooking.preprocessed.txt > cooking.valid
```

Let's train a new model on the pre-processed data:

```
>> ./fasttext supervised -input cooking.train -output model_cooking
Read 0M words
Number of words:  9012
Number of labels: 734
Progress: 100.0%  words/sec/thread: 82041  lr: 0.000000  loss: 5.671649  eta: 0h0m h

>> ./fasttext test model_cooking.bin cooking.valid
N  3000
P@1  0.164
R@1  0.0717
Number of examples: 3000
```

We observe that thanks to the pre-processing, the vocabulary is smaller (from 14k words to 9k). The precision is also starting to go up by 4%!

more epochs and larger learning rate

By default, fastText sees each training example only five times during training, which is pretty small, given that our training set only have 12k training examples. The number of times each examples is seen (also known as the number of epochs), can be increased using the `-epoch` option:

```
>> ./fasttext supervised -input cooking.train -output model_cooking -epoch 25
Read 0M words
Number of words:  9012
Number of labels: 734
Progress: 100.0%  words/sec/thread: 77633  lr: 0.000000  loss: 7.147976  eta: 0h0m
```

```
P@1  0.501
R@1  0.218
Number of examples: 3000
```

This is much better! Another way to change the learning speed of our model is to increase (or decrease) the learning rate of the algorithm. This corresponds to how much the model changes after processing each example. A learning rate of 0 would mean that the model does not change at all, and thus, does not learn anything. Good values of the learning rate are in the range `0.1 - 1.0`.

```
>> ./fasttext supervised -input cooking.train -output model_cooking -lr 1.0
Read 0M words
Number of words:  9012
Number of labels: 734
Progress: 100.0%  words/sec/thread: 81469  lr: 0.000000  loss: 6.405640  eta: 0h0m

>> ./fasttext test model_cooking.bin cooking.valid
N  3000
P@1  0.563
R@1  0.245
Number of examples: 3000
```

Even better! Let's try both together:

```
>> ./fasttext supervised -input cooking.train -output model_cooking -lr 1.0 -epoch 2
Read 0M words
Number of words:  9012
Number of labels: 734
Progress: 100.0%  words/sec/thread: 76394  lr: 0.000000  loss: 4.350277  eta: 0h0m

>> ./fasttext test model_cooking.bin cooking.valid
N  3000
P@1  0.585
R@1  0.255
Number of examples: 3000
```

Let us now add a few more features to improve even further our performance!

such as sentiment analysis.

```
>> ./fasttext supervised -input cooking.train -output model_cooking -lr 1.0 -epoch 2
Read 0M words
Number of words:  9012
Number of labels: 734
Progress: 100.0%  words/sec/thread: 75366  lr: 0.000000  loss: 3.226064  eta: 0h0m

>> ./fasttext test model_cooking.bin cooking.valid
N  3000
P@1  0.599
R@1  0.261
Number of examples: 3000
```

With a few steps, we were able to go from a precision at one of 12.4% to 59.9%. Important steps included:

- preprocessing the data ;
- changing the number of epochs (using the option `-epoch`, standard range `[5 - 50]`) ;
- changing the learning rate (using the option `-lr`, standard range `[0.1 - 1.0]`) ;
- using word n-grams (using the option `-wordNgrams`, standard range `[1 - 5]`).

Advanced readers: What is a Bigram?

A 'unigram' refers to a single undividing unit, or token, usually used as an input to a model. For example a unigram can be a word or a letter depending on the model. In fastText, we work at the word level and thus unigrams are words.

Similarly we denote by 'bigram' the concatenation of 2 consecutive tokens or words. Similarly we often talk about n-gram to refer to the concatenation any n consecutive tokens.

For example, in the sentence, 'Last donut of the night', the unigrams are 'last', 'donut', 'of', 'the' and 'night'. The bigrams are: 'Last donut', 'donut of', 'of the' and 'the night'.

Bigrams are particularly interesting because, for most sentences, you can reconstruct the order of the words just by looking at a bag of n-grams.

Scaling things up

Since we are training our model on a few thousands of examples, the training only takes a few seconds. But training models on larger datasets, with more labels can start to be too slow. A potential solution to make the training faster is to use the hierarchical softmax, instead of the regular softmax. This can be done with the option `-loss hs`:

```
>> ./fasttext supervised -input cooking.train -output model_cooking -lr 1.0 -epoch 2
Read 0M words
Number of words:  9012
Number of labels: 734
Progress: 100.0%  words/sec/thread: 2199406  lr: 0.000000  loss: 1.718807  eta: 0h0m
```

Training should now take less than a second.

Advanced readers: hierarchical softmax

The hierarchical softmax is a loss function that approximates the softmax with a much faster computation.

The idea is to build a binary tree whose leaves correspond to the labels. Each intermediate node has a binary decision activation (e.g. sigmoid) that is trained, and predicts if we should go to the left or to the right. The probability of the output unit is then given by the product of the probabilities of intermediate nodes along the path from the root to the output unit leave.

For a detailed explanation, you can have a look on this video.

In fastText, we use a Huffman tree, so that the lookup time is faster for more frequent outputs and thus the average lookup time for the output is optimal.

Multi-label classification

When we want to assign a document to multiple labels, we can still use the softmax loss and play with the parameters for prediction, namely the number of labels to predict and the threshold for the predicted probability. However playing with these arguments can be tricky and unintuitive since the probabilities must sum to 1.

```
Read 0M words
Number of words:  14543
Number of labels: 735
Progress: 100.0% words/sec/thread:   72104 lr:  0.000000 loss:  4.340807 ETA:   0h 0
```

It is a good idea to decrease the learning rate compared to other loss functions.

Now let's have a look on our predictions, we want as many prediction as possible (argument `-1`) and we want only labels with probability higher or equal to `0.5` :

```
>> ./fasttext predict-prob model_cooking.bin - -1 0.5
```

and then type the sentence:

Which baking dish is best to bake a banana bread ?

we get:

```
__label__baking 1.00000 __label__bananas 0.939923 __label__bread 0.592677
```

We can also evaluate our results with the `test` command :

```
>> ./fasttext test model_cooking.bin cooking.valid -1 0.5
N 3000
P@-1  0.702
R@-1  0.2
Number of examples: 3000
```

and play with the threshold to obtain desired precision/recall metrics :

```
>> ./fasttext test model_cooking.bin cooking.valid -1 0.1
N 3000
P@-1  0.591
R@-1  0.272
Number of examples: 3000
```

← LIST OF OPTIONS

WORD REPRESENTATIONS →