TDT4305 - PROJECT PART 2

JØRGEN THORSNES, CARL JOHAN HAMBRO

Setup

Create a virtualenvironment with python >= 3.6 . Then run:
`pip install pyspark`

usage: `python part2.py [-h] -t training-file -i input-file -o output-file [-s]`

Use naive bayes to predict tweet location

optional arguments:

- -h, --help : show this help message and exit
- -t training-file, --training : training-file : Path to training data file
- -i input-file, --input input-file : Path to input data file
- -o output-file, --output output-file : Path to output data file
- -s, --sample : Enable sampling of training input

The code also comes with a simple integration test that passes the examples given in the handout. This can be run with `python test.py`.

About
The over all idea is to use the whole data set to train the model and then test some input against this. We chose to filter the training data based on the input data first, which greatly reduced runtime from 2-3 mins to < 30 secs. However, this is unsutable if Spark is queried many times for different tweets as the model is cherry picked to fit a single input.

Description
Count total tweets and tweets per city and store this for later use.

The training tweets are loaded and all columns except location (city) and tweet-content are discarded. The tweet content is then unified to lowercase and split into a list of words with a single space as divider, but each word from a tweet is only stored if it exists in the set of words from the **input tweet**.

Each tweet is flat mapped to (city - word) tuples by using flatMap on each tweet, along with the set (!) of words in each tweet. A set is used since 1 tweet may only vote for any word once, not multiple times. All (city, word) tuples are counted with a reduceByKey on the whole tuple. This gives the basis for calculating Tc,w_i . Some shuffeling is done so the result can be "reverse flatmapped". This latter operation uses combineByKey, which we pass a simple "list add function" (list.**add**). The result are city as keys, and tuples of (word, frequency) as values. We convert this list of tuples to a dictionary. City-tweet-count is joined in to calculate Tc,w_i / Tc .

location_prob_intermediate calculates the sum product of all Tc,w_i / Tc by looking up each word in the input tweet in the dictionary for a given city. 0 is returned as the default value if a word does not exist in the city-word dict. The probabilities are then calculated over all the words in the input tweet.

The location_prob is found by multiplying in (Tc / |T|) with the previous value. This is done in paralell for all the cities. Finally, the top two cities are picked out and stored to disk.