TDT4305 - BIG DATA: PROJECT

JØRGEN THORSNES, CARL JOHAN HAMBRO

About
The course had a two-part project. The code for each part is documented and commented. The documentation for both parts are listed here as well. We received 100% score on part one and 98% for part two.

Setup
Create a virtualenvironment (aka `mkproject bigdata`) with python >= 3.6 . Then run `pip install -r requirements.txt`.

The data set must be extracted by
`cat raw_data/geotweets.tsv.gz.part* | gunzip -c > geotweets.tsv`
Thats it!

The documentation can be generated by running `python documentation.py`.

PART 1

About
Simply run `python task_*.py` to run a task.

To speed up debugging: use kwarg `sample=True` when calling `load()`: `load(sample=True)`.

Usually, 1 coalesce is used in each task to store the final result.

Task 1

a: Use count() function on the RDD

b, c, d, e: Use distinct() to remove all duplicate users, before .count()

f, g, h, i: Find min and max of latitud. This is done by passing the function max(a,b) and min(a,b) to the reducer function in the RDD object. This runs max and min over the entire dataset, producing only 1 output.

j: Run len() on the tweet content, on all tweets, by passing the function to map() on an RDD object. This is afterwards summed up by a reduce() call, and finally, divided by the total amount of tweets found in a.

k: Almost the same as j, but instead of len() which gives total characters, we simply count the amount of spaces, and add 1 (because there are, for instance, only two spaces in a three word sentence). This gives the total amount of words instead.

Task 2

Extract country name from tweet with a simple map. We add 1 to each value. This lets us count all tweets per country. This is done with reduceByKey(f), which iterates over pairs of key-value elements from the RDD instance dataset. This is re-run until there is only 1 K-V per distinct key. The value is the aggregated value of the result by applying **f** over the dataset. Finally the result is sorted alphabetically, then by count.

Task 3

Map all tweets to country names as keys with values: 1, lat and lon. These values are row-wise summed, e.g. lat1 + lat2 + ... + latN (same for "1" and lon), per key. This gives a cumulated lat and lon value along with total count, for each key. This is a SQL equivalent of a "GROUP BY" + "SUM". Finally, we divide the lat and lon values on their corresponding count values to give an average.

Task 4

Map tweets to country and local time as key, and 1 as value. We can then reduce these values "down" for each key. The result is pairs of country-hour with tweet counts as values. The local hour has to be moved over from the key to the value, in order to reduce all country values to one country with the maximum hour as value. A simple custom lambda function returns the row with the highest count. This is nessecary because the value is (hour, count), while a regular max function doesn't know how to compare such tuples.

Map: https://s.ntnu.no/bigdata_coarto (expired).

Task 5

First we filter tweets to only tweets with countrycode 'US' and place_type = 'city', to only work with the tweets we need. Then we map the rdd of tweets from the US to new rdd with place_name as key and add 1 as value for each row. We then use reduceByKey and "add" together the values for each row where place_name == place_name, to get wanted result (place_name, tweet_count). First we sort by place_name, and then by negative tweet_count to sort the rdd descending by tweet_count, and for equal number of tweets, sorted by place_name in alphabetical order.

Task 6

First filter all tweets into rdd whith only tweets with country_code == 'US' to get rdd with only the tweets we need. Then we use flatmap on the rdd to convert list of values to individual keys and add value 1 to each word. To get words as key, we first convert tweet_text to lowercase and split the tweet_text into the words, and exclude words shorter than 2 characters. We then use reduceByKey to count the frequency of the words, by adding the values of the rows where the keys (words) are equal. We load in the stopwords into a new rdd, and map the stopword rdd to (stopword, Add None), because subtractByKey needs the data in the two sets to be of equal format. We then use subtractByKey on the rdd with the counted words, and use the stopwords rdd as parameter, to get an rdd with the counted words without the stopwords. Lastly, we sort by frequency by using sortBy and sort by the word count, decreasing, because that looks nice.

Task 7

First we map all tweets to an rdd with ('place_name', ('country_code', 'place_type', 'tweet_text')) to get only what we need. We then filter the rdd to a new rdd with only tweets from the US and only where place_type == 'city'. We then map to a new rdd(City, 1), so that we can count how many tweets there are in one city. To do this we use reduceByKey on the rdd so that we add the values for each row, where the keys(cities) are equal. We then use takeOrdered(5, sorted by tweet_count descending) to get a list of the 5 cities with the highest number of tweets. Since we only have a list we parallelize the list so we get an rdd to work with again. We then map the rdd with the tweets from US cities, and left outer join this rdd with the rdd that contains the top 5 cities to only get another rdd with only relevant information. We then flatmap this new rdd to an rdd('word', ('place_name', 'tweet_count')), to get the words we convert the tweet_text to lowercase, split it at whitespace and exclude words shorter than 2 characters. We then load the stopwords into an rdd('stopword', None) (we add the None to be able to use subtractByKey). Then we use subtractByKey on the rdd('word', ('place_name', 'tweet_count')) with the stopwords rdd as parameter to get an rdd without the stopwords.

With each word as a key, and (city, city-tweet-count) as value, we map it back to something useful. This lets us count every word, per city, as the key is a composite of the two. We wish to use ReduceByKey on cities, in order to get the top N-words. Therefore, the word is mapped out of the key, to the value. Since the word-count is sorted, we can use a nifty custom add function to get the top-N results. The add function is run over each city, by starting at the top row, adding (word, word-count) as a list to the returned value. Once a keys value is a list of more than N results, we stop adding more words. Finally, we sort the result on the city-tweet count that we've kept all along and store the result.

PART 2

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

Example:

```
python part2.py -i input_tweet -t ../geotweets.tsv -o my_output.dat
```

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.