DCSA project report

Catrina Bodean

January 2021

1 Prerequisites for running the code

Before running any task, run the following command:

```
pip install -r requirements.txt
```

This will install all the additional packages used in the project.

Note: the results of each task are not mentioned in the report, but they are included as output files for each task.

2 Task 1 - Top 10 keywords for each movie genre

2.1 Short description of the code

The task is done in three steps.

Step 1. Grouping the input data by movie genre.

Mapper: It reads the input file line by line and returns a (genre, partial_title) pair for each line. The partial title is the original title without numerals, adverbs, conjunctions etc.

Reducer: It groups the pairs by genre and returns (genre, [titles]).

Step 2. Calculating the word count for each word pertaining to a genre.

Mapper: For each word in each title of each genre, it returns a (genre, [word, 1]) pair. This will allows me to count each word occurrence later on.

Combiner: It groups the [word, 1] lists by genre. Then, it goes through each [word, 1] element and creates a dictionary of form word: total_number_of_occurrences. At the end it returns a (genre, dictionary) pair.

Reducer: For each genre, it sorts the dictionary in decreasing order by number of word occurences and returns the ten most used words as a (genre, [word, counts]) pair.

Step 3. Displaying the top 10 keywords

Mapper: It takes the (genre, [word, counts]) pair and returns (genre, words) for each word in the list, removing the count number.

Reducer: It groups by genre the pairs given by the mapper and returns a (genre, [top_ten_keywords) pair for each genre.

2.2 Running the code

If we want to see the output in the console, we simply run

```
python task1.py movies.csv
```

If we wish to have the output saved in an output file, like "task1_output.txt", we use the command

```
python task1.py movies.csv > task1_output.txt
```

One such output file is included in the archive.

3 Task 2 - Reverse web-link graph

3.1 Short description of the code

This task is done in one step. First, the mapper reads the input file, "web-Google.txt", and returns a (to_id, from_id) pair for each relevant line of the file. Then, the reducer groups all the pairs by the to_id key.

Note: both the combiner and the reducer produce the desired output, so I chose to only use a reducer.

3.2 Running the code

If we want to see the output in the console, we simply run

```
python task2.py web-Google.txt
```

However, because the output is extremely long, it would be better to save it to an output file; if the name of our output file is "task2_output.txt", for example, we use the command

```
python task2.py web-Google.txt > task2_output.txt
```

One such output file is included in the archive.

4 Task 3 - k-nearest neighbour (k = 15)

4.1 Short description of the code

First and foremost, we need to normalize the features in the input file. In order to do this, we read the file, "Iris.csv", using the `pandas` library and the formula

$$x_{normalized} = \frac{x}{|x_{maximum}|} \tag{1}$$

Then, we create a new file, "Iris_normalized.csv", with the normalized values. I chose to create a new file so that the original values are not lost.

The KNN task itself is done in three steps:

Step 1. Grouping the input data by species.

Mapper: It reads the input file line by line and returns a (species, [id, features]) key-values pair for each line. This will allow us to group the features corresponding to each species, and also to identify the flowers that had not been categorised yet, who will have the species "".

Reducer: It groups the pairs by species and returns the labeled data, while saving the unlabeled data (species = "") in the class variable `unlabeled_data`.

Step 2. Calculating the distance between the unlabeled and the labeled data.

Mapper: It takes the (labeled_species, [id, features]) output of the previous reducer and calculates the distance between each set of features

and every set of unlabeled features. It returns a (unlabeled_set_id, [[distance, label]]) for each unlabeled and each labeled set of features. The distance is calculated using the formula

$$d(x, y) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + (x_3 - y_3)^2} \tag{2}$$

Combiner: Partially groups the pairs outputted by the mapper by unlabeled_set_id.

Reducer: Finishes grouping the pairs by unlabeled_set_id, then it sorts the values of the (key, value) pairs by the distance and returns (unlabeled_set_id, [[distance, label][0:15]]); the first 15 elements of the (distance, label) list correspond to the 15 nearest neighbours.

Step 3. Identify the most common label and return the prediction for each unlabeled feature set.

Mapper: It takes the (unlabeled_set_id, [[distance, label]]) pairs outputted by the reducer and, for each element of the values list, return a (unlabeled_set_id, label) pair.

Reducer: It groups by unlabeled_set_id the pairs given by the mapper and returns a (unlabeled_set_id, most_common_label) pair for each unlabeled feature set. It determines the most common label using the `Counter` function of the `collection` package; since all the label lists have only 15 elements, it makes no sense to use another MapReduce step just to count them.

4.2 Running the code

If we want to see the output in the console, we simply run

```
python task3.py Iris_normalized.csv
```

If we wish to have the output saved in an output file, like "task3_output.txt", we use the command

```
python task3.py Iris_normalized.csv > task3_output.txt
```

One such output file is included in the archive.

Note: Although there is no "Iris_normalized file in the project folder, it will be generated by the Python file.

5 Task 4 - The Frobenius Norm of a given matrix

5.1 Short description of the code

This task is resolved in two ways: the first one uses the index of the matrix line each element is on as a key; the second method does not use any key. The rationale for the second method is that (a) the equation of the Frobenius norm is

$$\|A\|_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}|a_{ij}|^2} \tag{3}$$

and (b) addition is commutative; therefore, as long as all the elements of the matrix are summed before applying the square root, the order in which they are added doesn't matter.

Method 1

Step 1. Calculating the sum of the elements of each matrix line.

Mapper: It reads the input file line by line and returns a (line_id, |number|^2) pair for each number in the line. The line index is generated by increasing a class variable by 1 each time a new line is read.

Reducer: It groups the pairs by line_id and returns a (_, row_sum) pair, where row_sum = the sum of all the elements of one line. We don't return a key because we need to add all the row sums, so we don't need to group them by anything.

Step 2. Calculating the Frobenius norm.

Reducer: It calculates the sum of all the row sums outputted by the previous reducer and outputs the square root of the final sum.

Method 2

Step 1. Calculating the sum of the elements of each matrix line.

Mapper: It reads the input file line by line and returns a (_, |number|^2) pair for each number in the line.

Reducer: It groups the pairs randomly and returns a (_, sum) pair, where sum = the sum of all the elements grouped by the same "key". We don't return a key because we need to add all the sums, so we don't need to group them by anything.

Step 2. Calculating the Frobenius norm.

Reducer: It calculates the sum of all the sub-sums outputted by the previous reducer and outputs the square root of the final sum.

In practice, both methods output approximately the same value, with small differences caused by the different propagation of the floating point error (e.g. 128.4506152933251 and 128.45061529332497).

5.2 Running the code

The code can be run using the command

```
python task4.py A.txt
```

If we wish to have the output saved in an output file, like "task4_output.txt", we use the command

```
python task4.py A.txt > task4_output.txt
```

One such output file is included in the archive.