BIOINFORMATICS

Vol. 00 no. 00 2015
Pages 1–3

Report for the Project in Medical Bioinformatics

Stefan Jacob [1,*], Zarin Shakibaei [1,*]

[1]BigDaddyAG, Fachbereich Mathematik und Informatik, Freie Universität Berlin, Takustraße 9, 14195 Berlin, Germany

Received on XXXXX; revised on XXXXX; accepted on XXXXX

Associate Editor: Tim Conrad

ABSTRACT

Motivation: As the amount of biological data has recently increased due to new and enhanced techniques like next-generation sequencing, the analysis of this data also needs to improve and especially speed up. As a result, new frameworks and platforms for distributed stream and batch data processing are evolving. In a university project, we developed a pipeline for the analysis of biological data from *The Cancer Genome Atlas (TCGA)* data portal, which provides - among others - the access to data from high level sequence analysis of tumor genomes. In this report we present the used methods and discuss their results.

Results: n/a

Availability: All data we used is available in our GitHub repository under: https://github.com/BigDaddyAG/MedBioProject

Contact: stefan.jacob@fu-berlin.de and zarin.shakibaei@fu-berlin.de

1 INTRODUCTION

As a summarizing project of our course "Big Data in Life Sciences" we were asked to develop a pipeline that can analyze data from the *The Cancer Genome Atlas (TCGA)* (with a focus on lung adenocarcinoma) and classify as a result between to different subgroups of cancer or between an ill and healthy patient.

This is our story of the failures, frustrations and all issues we faced...

2 APPROACH

Following an overview of how we thought the pipeline could look like:

- get appropriate data from TCGA (either one or both)
  - transcriptomic data (from GCC[1])
  - tissue/patient specific information (from BCR[2])
- preprocess the data (filter out unnecessary information and join all patients into one single data table)
- train flink machine learning API using above created data
- take new data and classify it based on rendered model

The generated pipeline should be based on *Apache Flink*, an open source platform for distributed stream and batch data processing that is in an early development state (beta status).

The final version of the pipeline should also be able to run on a computer cluster in order to speed up calculation.

To achieve all this and simplify collaboration (and also because we were "forced" to do it) we set up a GitHub repository that contains all used data and the code written so far.

3 DATA

Two different data archives have been downloaded from *The Cancer Genome Atlas*.

The first archive comes from the *Genome Characterization Center (GCC)* and contains text files with gene samples and their corresponding expression level (log2 lowess normalized).

The second archive which is provided by the *Biospecimen Core Resource (BCR)* contains carefully cataloged tissue and sample information with important medical information about the patient. Each text file from the BCR contains other information e.g. the gender of the patient or the treatment, used drugs and some information about the tumor (size, site, location, etc.)

As mentioned previously, the downloaded GCC-Data folder contains 35 text files, each referring to one patient. The first column contains all Genes from a microarry chip, the second column describes expression levels corresponding to each gene.

4 METHODS

4.1 Various approaches for data preparation

4.1.1 Transcriptomic data After mature consideration, we decided to take the above-mentioned transcriptomic data from TCGA's Genome Characterization Center (GCC) and join it in a big table with Flinks TableAPI (see below for the code).

The goal here was to read all text files in the GCC folder that contain about 17,000 lines with two columns (one for the gene symbol and one for the logarithmized expression level for each patient) and join them so that in the end, we get one single file, which contains one gene column and 35 gene expression level columns of all 35 patients.

Here is the used scala code to read in all text files iteratively:

```
1 // function to list all files in a directory
2 def getListOfFiles(dir: String): List[File] = {
3   val d = new File(dir)
```

*to whom correspondence should be addressed

[1] Genome Characterization Center

[2] Biospecimen Core Resource

```
  if (d.exists && d.isDirectory) {
    d.listFiles.filter(_.isFile).toList
  } else {
    List[File]()
  }
}
```

We generated a list of strings that each contains the path to a single (text) file. Next, we split up the list at each unique path and iterated over them.

```
// extract the filename of absolute path to file
val files = getListOfFiles(dataGCCFilePath)
val filenameArray = files.toString.split(",")
val sizeOfFilenameArray = filenameArray.size
```

Then, we tried to iterate over our newly created array of file paths and join them into a table.

```
val firstFile = getDataSetFile(env,
        filenameArray(1)).as('firstFileCol1,
        'firstFileCol2)
for (i <- 2 to filenameArray.size-2) {

// Read a file but only includes the 1st,
// 2nd column - returns DataSet[MyLineitem]
val CurrentFile = getDataSetFile(env,
        filenameArray(i)).as('col1, 'col2)

val items =
        firstFile.join(CurrentFile)
                .where('firstFileCol1 === 'col1)
                .select()
}
items.writeAsCsv("file://path", "\n", "\t")
```

This method has been failed due to a `writeAsCsv()` method problem. Somehow this method didn't recognize the `items` variable out of the for-loop. In addition how joining in Flink works would never let us to join tables in this way. Refering to the documentation, Flink can always join two tables with fixed column sizes but as one of our tables was growing over the time, Flink wasn't able to handle that.

As the iteration turned out not to work properly, we ended up in joining every two tables manually until all 35 patients were in one big table[3].

The problem now was that we had a table with 17,816 lines - one for each gene and its expression level. We came up with the idea to preprocess the data in that way, that we only wanted to keep genes that showed a significant change in their expression value over the different patients. Therefore, we tried to use the standard deviation for each line of the table and only take lines into account, that had a SD bigger than for example 3. The occurring problem was that Flinks TableAPI was only capable of taking the SD of columns into account, but not the one of lines. Thats why we transposed the dataset with the expression levels resulting in a set that had 17,816 columns. Now the problem was that we had to enumerate these 17,816 columns in order to be able to read in the dataset with Flinks TableAPI as each unique column had to the mentioned (we found no way to simply read in all columns without telling Flink how many columns there exists). So that's where we got stuck in the end.

4.1.2 Tissue/patient specific information data a.k.a. the only Flink alternative approach To continue the project, we needed a smaller input data (with less columns). So looking into the BCR data inspired us to preprocess these files and prepare an input file for Flink-ML (machine learning). The file mentioned below contained interesting information, which could be used to classify patients into two groups regarding their smoking behavior:

`nationwidechildrens.org_clinical_patient_luad.txt`

[3] That was also the proposed method by the Flink developers!

After reconsidering all available approaches, it remained only one way to handle our input data, using

```
Interface InputFormat<OT,T extends InputSplit>
        (text, file InputFormat)
```

and then overwrite the default `createInputSplits` function with an own one that reads in every file as an input split record and then saves it in an own specific file format of 2-tuples containing the gene symbols (1st tuple) and all the expression values (2nd tuple) as `<String><<double, double, doule, ...>>`

Due to lack of time (and because we met the Flink developers one day before the project's deadline) we weren't able to finish this approach.

4.2 Machine Learning Methods

Referring the definition of SVM:

> In machine learning, support vector machines (SVMs, also support vector networks) are supervised learning models with associated learning algorithms that analyze data and recognize patterns, used for classification and regression analysis. Given a set of training examples, each marked for belonging to one of two categories, an SVM training algorithm builds a model that assigns new examples into one category or the other, making it a non-probabilistic binary linear classifier. *(citated from Wikipedia)*

The goal here was to use an appropriate input data and train Flink ML (SVM) and to render a model which could classify any given data (test data).

We could present the ML-API with example inputs of patients who smoked longer than 30 years and died of lung cancer (death as a desired output) and the ML could learn a general rule that maps patients to appropriate consent status.

5 DISCUSSION

6 CONCLUSION

This project was harsh!

Not knowing exactly which data to take, going through unreadable and not well curated Flink documentation and struggling with unresolvable error warnings and crude design concepts of Flink, we had a really frustrating time of one and a half months. If showing us how undelightful "Big Data" could be was one of the aims of the project, then it definitively predominated!

Having spent countless hours without any significant progress was even more frustrating as our instructor told us that he managed to bring up a pipeline in half a day.

But we couldn't count on his consultation as he was completely unreachable during the whole time of the project. That's why we used the opportunity to talk to the Flink developers directly, which - in the end - didn't solve any of our problems but showed us that they also had no real clue how to realize this project with Flink.

Furthermore, while talking to the Flink developers, we got to know that their changed their focus more on stream data processing and not on machine learning algorithms, which will stay in a beta status.

This all sums up in the conclusion that we wouldn't recommend Flink for the creation of a pipeline for analyzing genomic or medical data (quod erat demonstrandum?). The Flink developers even encouraged us to do the pipeline completely without Flink and only in Scala.