Please upload your solutions on or before the due date to the provided Moodle assignment as a single zip file using your group id as the file name. Provide some brief instructions on how to run your solution to each problem in a file called `Problem_X.txt`*, and report also the most important results (such as number of results, runtimes, etc.) of your solutions to that problem within this file. Remember that all solutions may be submitted in groups of up to 3 students.*

Word Counting via the Java API of Hadoop — 12 Points

Problem 1. Consider the three Java classes `HadoopWordCount.java`, `HadoopWordPairs.java` and `HadoopWordStripes.java` provided on Moodle to solve this exercise. Start by creating a new project in your Eclipse-Java IDE by following the steps described in the "Getting Started" guide (or alternatively follow the steps described in "`run-intro-examples.sh`" to do the same from the command-line shell). Note that the solutions of this exercise may be measured either by using your local Hadoop installation or directly via the Iris HPC cluster.

(a) Modify the three Java classes above, such that they extract only *consecutive sequences of word characters (*`\w`*) and digits (*`\d`*)* from the `value` received by the `map` function as tokens (instead of just splitting by *spaces* as it is currently the case).

That is, all tokens emitted by the `map` function should only consist of sequences of alphanumeric characters. Any combination of word characters and digits is allowed per token, but no other characters may be contained within a token. All word-characters should also be turned to lower cases. **2 points**

See, e.g., `http://www.tutorialspoint.com/java/java_regular_expressions.htm` for a tutorial on using regular expressions in Java.

(b) Compare the performance of the `HadoopWordCount.java`, `HadoopWordPairs.java` and `HadoopWordStripes.java` classes (with the improved tokenizer from (a)) by

(i) enabling the given `Reduce` class for being used as a `Combiner` function, and by

(i) disabling the given `Reduce` class from being used as a `Combiner` function.

Report the runtimes for both cases along with your solution when using the three classes on the `AA` subdirectory of the `Wikipedia-En-41784-Articles.tar.gz` archive. **2 points**

(c) Further improve the performance of the `HadoopWordCount.java`, `HadoopWordPairs.java` and `HadoopWordStripes.java` classes (again with the improved tokenizer from (a)) by implementing an explicit form of *local aggregation* (e.g., by using an appropriate `HashMap` in Java)

(i) inside the `map` method, and

(ii) inside the `Map` class.

Report the runtimes for both cases along with your solution when using the three classes on the `AA` subdirectory of the `Wikipedia-En-41784-Articles.tar.gz` archive. **4 points**

(d) Perform a "strong scalability" test for the best version (using all improvements from (a)–(c)) of your `HadoopWordCount.java`, `HadoopWordPairs.java` and `HadoopWordStripes.java` classes by running each of the three classes on the `AA`, `AB`, `AC` and `AD` subdirectories of the `Wikipedia-En-41784-Articles.tar.gz` archive.

That is, run the three classes first on the `AA` directory (25% input size) and note the runtime of the programs; next run the programs on the `AA` and `AB` directories (50% input size) and again note the runtime of the programs; continue this until you also have the 75% and 100% measurements. **4 points**

Word Counting via the Scala API of Spark **12 Points**

Problem 2. Consider the Scala object `SparkWordCount.scala` that is provided on Moodle. Ideally create a new Scala project in your Eclipse-Scala IDE by following the steps described in the "Getting Started" guide (or alternatively follow the steps described in "`run-intro-examples.sh`" to do the same from the command-line shell). Note that the solutions of this exercise may be measured either by using your local Spark installation or directly via the Iris HPC cluster.

(a) Also here, modify the Scala object above, such that it extracts only *consecutive sequences of word characters (`\w`) and digits (`\d`)* from each `line` received by the `flatMap` transformation (instead of just splitting by *spaces* as it is currently the case).

That is, all tokens emitted by the `flatMap` function should only consist of sequences of alphanumeric characters. Any combination of word characters and digits is allowed per token, but no other characters may be contained within a token. All word-characters should also be turned to lower cases. **2 points**

See, e.g., `http://www.tutorialspoint.com/scala/scala_regular_expressions.htm` for a tutorial on using regular expressions in Scala. Note that you need to make yourself familiar with the more functional programming style of Scala to modify the tokenizer as described above.

(b) Compare the performance of your Scala program (with the improved tokenizer from (a)) by replacing the current `reduceByKey` *transformation* into a `countByValue` *action*. Please also save the result of your `countByValue` action into a single file (e.g., using a `BufferedWriter` from Java) rather than using the current `saveAsTextFile` action for RDDs. **3 points**

(c) Further improve the performance of your Scala program by *caching* one or more of the underlying RDD objects, such that they can be efficiently re-used for counting the *number of lines* (in addition to the *word counts*) that are contained in your input documents. Your resulting program should print both the *total number of lines* and the *total number of words* contained in your input documents in addition to the previous word counts. **3 points**

See, e.g., `http://spark.apache.org/docs/latest/rdd-programming-guide.html` for a documentation on how to use RDD persistency in Scala.

(d) Implement a `combineByKey` aggregation function in your Scala program in order to compute the *average amount of times each word occurs among the input documents*. That is, for each distinct word contained in the input documents, your program should return the word together with the sum of times this word occurs in all documents; this value then needs to be divided by the number of documents in which the word occurs in order to obtain the correct average.
4 points

You may use the `Wikipedia-En-41784-Articles.zip` from Moodle or any other reasonably large document collection for testing your solutions.

Processing Twitter Streams via the Scala API of Spark **12 Points**

Problem 3. Finally, consider the Scala program `SparkTwitterCollector.scala` provided on Moodle. It implements a simple connection to Twitter to retrieve a stream of JSON documents that contain a random selection of real-time tweets – including data and metadata – and saves these JSON documents to your local file system. Add the additional jar files provided on Moodle to your Eclipse-Scala project to compile and run the Twitter collector example (or alternatively follow the steps described in "`run-intro-examples.sh`" to do the same from the command-line shell). Also here the solutions of this exercise may be measured either by using your local Spark installation or directly via the Iris HPC cluster.

The command to submit the `SparkTwitterCollector` object as a new job in Spark is:

```
spark-submit --class SparkTwitterCollector SparkTwitterCollector.jar \
<output-path> <time-window-secs> <timeout-secs> <partitions-per-interval>\
<twitter-consumer-key> <twitter-consumer-secret> \
<twitter-access-token> <twitter-access-token-secret>
```

(a) First, you will need to generate credentials to access the Twitter API via the given `OAuth` authorization protocol (you should previously create a Twitter account for you or your group). After obtaining your twitter keys, you can already test the `SparkTwitterCollector.scala` object. You may try different parameter configurations and have a look at the output data. **4 points**

See, e.g., `https://www.prophoto.com/support/twitter-api-credentials/` for detailed instructions on getting Twitter API credentials.

(b) Adapt the `SparkTwitterCollector.scala` example in order to retrieve only those tweets that include a given keyword which is provided as an extra command-line argument to your `spark-submit` call. In addition, instead of saving the full JSON documents – called `statuses` – per tweet, extract only the text content of the tweet as output data. To do so, implement a simple regular expression that extracts the value of the corresponding `text` field from the JSON format. **4 points**

See `https://spark.apache.org/docs/1.6.0/api/java/org/apache/spark/streaming/twitter/TwitterUtils.html` and `http://twitter4j.org/javadoc/twitter4j/Status.html` for the API documentations.

(c) Include the code from `SparkWordCount.scala` to implement a *word counting* version for our Twitter streaming example, such that it counts the number of times that a term appears in a given RDD created from the incoming stream of tweets (thus selecting only those tweets that match your given set of keywords). The output words should be sorted in descending order of their counts. Please use the improved tokenizer described in Problems 1 & 2 of this exercise sheet to solve this step. **4 points**

Please upload your solutions on or before the due date to the provided Moodle assignment as a single zip file using your group id as the file name. Provide some brief instructions on how to run your solution to each problem in a file called `Problem_X.txt`*, and report also the most important results (such as number of results, runtimes, etc.) of your solutions to that problem within this file. Remember that all solutions may be submitted in groups of up to 3 students.*

***Note:** Since this exercise sheet imposes intensive computational workloads for the hyper-parameter tuning and cross-validation steps of your machine learning approaches, it is highly recommended to run the following experiments on the Iris HPC cluster by using either the interactive or batching modes of the Spark launcher scripts (see once more the "Getting Started" provided on Moodle for detailed instructions).*

Predicting Forest Covers via Decision Trees/Forests **12 points**

Problem 1. Consider the CoverType dataset (available at: `https://archive.ics.uci.edu/ml/machine-learning-databases/covtype/`) that was introduced in Chapter 2 of the lecture to solve this problem. Also consider the sample shell script "`RunDecisionTrees-shell.scala`" provided on Moodle as basis to solve this exercise, and focus on the improved encoding of the training data which transforms the original 1-hot encoding of the "wilderness-area" and "soil-type" attributes into two (single-dimensional) categorical attributes.

(a) In the first step, suppose we wish to include the form of *cross-validations* as they were introduced in the lecture (Chapter 3) in order to more systematically learn the hyper-parameters of a Decision Tree classifier. To do so, extend the given shell script by the following steps:

(i) Start from the original `rawData` RDD of the provided shell script but, instead of applying an 80%/10%/10% split, split this RDD into a 90%/10% split. As before, the 90% split will be used for hyper-parameter tuning, while the 10% split will be used for the final evaluation of your model under the best hyper-parameters you found.

(ii) Consider the `CrossValidator` Scala API of Spark 2.2.3 to implement a fully automatic hyper-parameter tuning step in your shell script. Specifically, use the 90% split of (i) to initialize a 5-fold cross-validation loop over a fixed hyper-parameter setting provided in the script (using `impurity="entropy"`, `depth=10` and `bins=300` as fixed parameters). Use the classification *accuracy* over all of the 7 tree types (obtained as before from the `MulticlassMetrics` package of Spark) to measure the performance of your model for the given hyper-parameters.

(iii) Finally measure the *accuracy* of your model and hyper-parameters over the 10% split you obtained in (i).

Please report the Spark runtime for steps (i)–(iii) of this experiment. **4 points**

(Refer to `https://spark.apache.org/docs/2.2.3/ml-tuning.html` *for the usage of the* `CrossValidator` *API.)*

(b) Next, extend the cross-validation setup you developed in (a) by investigating actual *ranges of hyper-parameters*. Specifically, consider the following ranges of hyper-parameters for your experiments:

- `impurity <- Array("gini", "entropy");`
- `depth <- Array(10, 25, 50);`
- `bins <- Array(10, 100, 300));`

Also here, use the built-in `CrossValidator` API for automatic parameter tuning over the 90% split of the data. In addition, provide a final evaluation of your model and hyper-parameter choice by using the 10% testing set of (a) over the 7 classes of trees given by the CoverType dataset. Report the resulting parameters and the Spark runtime of this experiment. **4 points**

(c) Finally, repeat the methodology described in (b) by training a Random Forest classifier instead of a single Decision Tree. Investigate the following range of parameters to also find the *best amount of trees* (using the "automatic" training mode) in your random forest:

- `numTrees <- Array(5, 10, 20);`

Once more, identify the best choice of all hyper-parameters you obtain from the cross-validations as described in (a)–(c) and measure the final *accuracy* you obtain for these hyper-parameters with the one that was obtained by the simpler 80%/10%/10% split of the original script. Also here, use the 10% testing set to measure the accuracy of your final model over the 7 classes of trees given by the CoverType dataset. Report the resulting parameters and the Spark runtime of this experiment. **4 points**

Weather Prediction via Regression Trees/Forests **12 points**

Problem 2. The Integrated Surface Data (ISD) (available at: `http://www1.ncdc.noaa.gov/pub/data/noaa/`) of the National Oceanic and Atmospheric Administration (NOAA) provides a rich collection of weather measurements from around the world. The Air Force station in Luxembourg is also included with all its recordings between 1949 and 2020 (these are all files in the subdirectories that start with the USAF identifier `065900`). For a detailed file format and field explanation, read also the specifications provided in `ish-format-document.pdf`.

Note that, also here, you should consider the sample script "`RunDecisionTrees-shell.scala`" provided on Moodle as basis to solve this exercise; the respective NOAA data dumps and a parsing function for the particular data format are also provided on Moodle.

(a) Write a Spark shell script for parsing the input files into an appropriate RDD structure in order to then train a *Regression Tree* that can predict the temperature in Luxembourg for a given date and time of the day. Following the instructions from the NOAA specification file, you will only need to extract the following fields from the input files

4. `GEOPHYSICAL-POINT-OBSERVATION date` (split into three fields: `year`, `month` and `day`)
5. `GEOPHYSICAL-POINT-OBSERVATION time` (extract only the hour without the minutes)
7. `GEOPHYSICAL-POINT-OBSERVATION latitude coordinate`
8. `GEOPHYSICAL-POINT-OBSERVATION longitude coordinate`
10. `GEOPHYSICAL-POINT-OBSERVATION elevation dimension`
13. `WIND-OBSERVATION direction angle`
16. `WIND-OBSERVATION speed rate`
18. `SKY-CONDITION-OBSERVATION ceiling height dimension`
22. `VISIBILITY-OBSERVATION distance dimension`
26. `AIR-TEMPERATURE-OBSERVATION air temperature`
28. `AIR-TEMPERATURE-OBSERVATION dew point temperature`

thus considering the `air temperature` as your target attribute. Be cautious about parsing floating point values and scaling factors properly. Save your input RDD to avoid the need of re-processing and parsing all of the input files every time you run your application. **3 points**

(b) Implement a Scala function to *optimize the parameters* and to train a *Random-Forest Regressor* (based again on $n = 10$ Regression Trees and automatic training mode) by splitting the dataset into a (i) *training set* with all records from 1949 until 2018 and a (ii) *test set* with all measurements for

2019. You may use the suggested combination of `for` loops provided in the original script or the built-in `CrossValidator` API of Spark for this step.

Based on your resulting model, predict the temperature for Luxembourg on `December-24-2019 at 18:00` and on `April-22-2020 at 8:00` and compare your results with the actually measured values (see, e.g., `AccuWeather.com` for some archived recordings of weather data). **3 points**

Hint: See the Spark API for a reference on how to use the regression-based variant of Decision Trees: `https://spark.apache.org/docs/2.2.3/mllib-decision-tree.html`

(c) Using your previously trained model, predict the temperatures for the Luxembourg weather station along the whole year of 2019, for all days and times at which data is available from NOAA for 2019. Calculate the *Mean Squared Error* (MSE) (see the above URL for an example) between the predicted temperatures and the actual measurements for 2019 (on those dates available from NOAA for 2019). **3 points**

(d) Which of the above attributes do you think has the *highest correlation* with our target attribute, i.e., the `air temperature`?

To answer this question, find compute the *Spearman's correlation coefficient*

$$\rho(X,Y) = 1 - \frac{6\sum_{i=1}^{n}\delta_i^2}{n(n^2-1)}$$

where X is the distribution of observations under the `air temperature` attribute and Y is the distribution of values under each of the remaining attributes (using the entire NOAA dataset for Luxembourg as input).

Note that, in the above formula, $\delta_i := rank(X_i) - rank(Y_i)$ denotes the difference in ranks between two observations X_i and Y_i, while n is total number of observations. You may consider the natural order of values for the categorical attributes to apply this coefficient. You may also want to appropriately discretize the numerical attributes to calculate the coefficient. **3 points**

Hint: You may directly use the implementation of the Spearman coefficient described in `https://spark.apache.org/docs/2.2.3/mllib-statistics.html` *to solve this exercise.*

Recommender Systems via Matrix Factorization **12 points**

Problem 3. Consider the AudioScrobbler dataset (available from Moodle) that was introduced in Chapter 3 of the lecture to solve this problem. Also consider the sample shell script "`RunRecommender-shell.scala`" provided on Moodle as basis to solve this exercise. Also here, we will follow up on the idea of extending the provided shell script via hyper-parameter tuning and cross-validations.

(a) Before we can invoke the actual hyper-parameter tuning steps, we need to define a proper quality measure for our resulting recommender engine. To do so, we will extend the given script by computing the *AUC measure* as it was introduced in the lecture.

(i) Instead of using the entire dataset to create the `trainData` RDD (including the resolved artist aliases), create a 90%/10% split into two separate `trainData` and `testData` RDDs. As before, the 90% split will be used for hyper-parameter tuning, while the 10% split will be used for the final evaluation of your model under the best hyper-parameters you found. To evaluate the recommendations properly, make sure that for each user, you put about 90% of the artists this user has listened to into `trainData` while the remaining 10% are put into `testData`. Follow the steps provided in the script to train a first recommender model by using the `trainData` RDD.

(ii) Next, take 1000 random users (similarly to the `someUsers` RDD toward the end of the script) from the `testData` RDD and compute the top 100 most recommended artists for each of them. For each of the 1000 lists of recommendations, create a new RDD consisting of $(\hat{y}, y)$ pairs, where $\hat{y}$ is the recommendation value of the respective artist for the current user, and y is a binary value indicating whether the current user has indeed listened to that artist ($y = 1$) or not ($y = 0$) based on the provided `actualArtistsForUser` Scala function.

This RDD should be used as input to the `BinaryClassificationMetrics` API of Spark which then computes the AUC value for each user automatically. Finally, compute the *average AUC* over all of the 1000 previously selected users and compare this value also with a similarly computed AUC value when using the `predictMostPopular` baseline recommendation function provided in the script. **4 points**

(Refer to `https://spark.apache.org/docs/2.2.3/mllib-evaluation-metrics.html` *for the usage of the* `BinaryClassificationMetrics` *API.)*

(b) Next, consider a "manual" search for the best hyper-parameters of your Recommender System. To do so, implement a triple `for` loop (as shown in Chapter 3, Slide 27) in the provided shell script to evaluate your system over the following ranges of hyper-parameters:

- `rank <- Array(10, 25, 100);`
- `lambda <- Array(1.0, 0.01, 0.0001);`
- `alpha <- Array(1.0, 20.0, 40.0);`

For each selection of hyper-parameters, compute the AUC measure of your model over the 10% split contained in the `testData` RDD you created in (a). That is, for each of the 1000 random users you selected, check whether they indeed listened to a newly recommended artist and use this information for the AUC computation under each choice of hyper-parameters. Print the best choice of hyper-parameters and report the overall runtime in Spark for this experiment. **4 points**

(c) Finally, consider the `CrossValidator` Scala API of Spark 2.2.3 to implement a fully automatic hyper-parameter tuning step in your shell script. Use the same setting as in (a)–(b): first, create a 90%/10% split into `trainData` and `testData`; next, initialize a 5-fold cross-validation loop over the same range of parameters as specified in (b) to find the best model.

Provide a final evaluation of your model and hyper-parameter choice by using the 10% testing set over the 1000 users you selected in (a) and by checking whether they indeed listened to the newly recommended artists among those contained in the 10% `testData` split. Again, print the best choice of hyper-parameters and report the overall runtime in Spark for this experiment. **4 points**

(Refer to `https://spark.apache.org/docs/2.2.3/ml-tuning.html` *for the usage of the* `CrossValidator` *API.)*

Big Data Analytics

Prof. Dr. Martin Theobald & Dr. Vinu Venugopal
Summer Semester 2020 – University of Luxembourg

Exercise Sheet #3 – Due on May 13, 2020 – 8:00am

Please upload your solutions on or before the due date to the provided Moodle assignment as a single zip file using your group id as the file name. Provide some brief instructions on how to run your solution to each problem in a file called `Problem_X.txt`*, and report also the most important results (such as number of results, runtimes, etc.) of your solutions to that problem within this file. Remember that all solutions may be submitted in groups of up to 3 students.*

***Note:* Since this exercise sheet imposes intensive computational workloads for processing and hyper-parameter tuning, it is highly recommended to run the following experiments on the Iris HPC cluster by using either the interactive or batching modes of the Spark launcher scripts (see the "Getting Started" and recent forum posts provided on Moodle for detailed instructions).**

Latent Semantic Analysis of Wikipedia Articles — 12 Points

Problem 1. Consider the Wikipedia dataset which is available from Moodle under the link `Wikipedia-En-41784-Articles.tar.gz` of Chapter 1. Also consider the sample script `RunLSA-shell.scala` provided on Moodle as a basis to solve this exercise.

(a) Translate the `RunLSA-shell.scala` shell script with the provided Scala functions into an actual Scala object called `RunLSA.scala`, and compile this object into a jar file called `RunLSA.jar` which can then also be executed via `spark-submit` on the HPC cluster.

Create two versions of your code (and corresponding jar files), namely (i) one that performs an NLP pipeline for parsing the articles, and (ii) one that uses a simple tokenizer which parses only for alphanumeric sequences of characters as tokens (using again the `w` and `d` characters classes). Also turn all characters into lower cases and remove all stopwords for both (i) and (ii). **4 Points**

(b) Implement two basic `save(.)` and `load(.)` functions in order to save (and later load) the three matrices you obtain from an SVD in Spark into three files in order to avoid the need for recomputing these matrices for further experiments. **4 Points**

(c) For the entire dataset consisting of all the 41,784 Wikipedia articles, run your jar file containing the `RunLSA.scala` object with the following configurations of hyper-parameters and measure their runtimes:

- $numFreq \in \{10000, 25000, 50000\}$ for the number of frequent terms extracted from all Wikipedia articles, and
- $k \in \{100, 250, 1000\}$ for the different numbers of latent dimensions used for the SVD.

Based on the examples provided in the `RunLSA-shell.scala` shell script, compute the *top-25 words* and the *top-25 documents* under the *top-10 latent concepts* for each of the above choices of hyper-parameters and by either enabling or disabling the NLP pipeline. Manually inspect the results and determine which of the parameter settings you think works best. Save all matrices into corresponding files. **4 Points**

Note: The setting with $numFreq = 50000$ *and* $k = 1000$ *should run in less than one hour for all of the 41,784 articles on the HPC cluster.*

Implementing a Search Engine via Latent Semantic Analysis **12 Points**

Problem 2. Consider the Wikipedia dataset which is available from Moodle under the link `Wikipedia-En-41784-Articles.tar.gz` of Chapter 1 and the sample script `RunLSA-shell.scala` also to solve this exercise.

(a) By using the NLP pipeline provided in the `RunLSA-shell.scala` script, implement a basic parsing function that creates a *sparse vector* representation of each Wikipedia article in the collection. The dimension in each of the vectors should represent a *word lemma*, while the value at each dimension should contain the *term frequency* (TF) of that word in the Wikipedia article represented by the vector. Make sure that you keep all of your document vectors in a corresponding RDD that is distributed within your Spark environment.

You may refer to the `LinearRegression-shell.scala` script provided on Moodle to implement this part based on the `HashingTF` API of Spark. **4 Points**

(b) Implement a basic search engine for the Wikipedia articles by computing the *Cosine measure* between a document vector $\boldsymbol{d}$ and a similarly translated query vector $\boldsymbol{q}$:

$$Cosine(\boldsymbol{d}, \boldsymbol{q}) = \frac{\boldsymbol{d} \cdot \boldsymbol{q}}{\|\boldsymbol{d}\|_2 \cdot \|\boldsymbol{q}\|_2}$$

Make sure that the Cosine function is evaluated in parallel over your RDD containing the document vectors for each query. Then, sort the documents in descending order of Cosine similarities, and finally return the top-25 document vectors (and corresponding article titles) with the highest similarities to each such keyword query. **4 Points**

(c) Implement an analogous form of a search engine for the Wikipedia articles by using the SVD functions provided in the `RunLSA-shell.scala` (under the given parameters). Use the latter part of the script to transform a query vector into a corresponding representation of latent concepts and return also here the top-25 document vectors (and corresponding article titles) that match a given keyword query.

Think of 5–10 interesting keyword queries and compare the results you obtained in (b) with the ones you obtain for your SVD. Which one do you think is the better approach? **4 Points**

Note: You may also use the `save(.)` *and* `load(.)` *functions of Problem 1 to save and load a number of precomputed SVD decompositions with different parameters for this part, but this is not required.*

Social Network Analysis in GraphX **12 Points**

Problem 3. Consider `RunGraphX-shell.scala` as a basis to solve this exercise. Also, download and extract the *YouTube Online Social Network* open dataset available at: `https://snap.stanford.edu/data/com-Youtube.html` to solve the following parts of this exercise. This dataset includes the ground-truth communities which are basically user defined groups that other users can join.

(a) Parse and load the `com-youtube.ungraph.txt.gz` file obtained from the above URL into an initial RDD that contains all pairs of (*sourceId, targetId*) vertex ids as edges. Next, perform an initial *in-* and *out-degree analysis* by counting: (1) for each distinct *targetId* how many incoming links this vertex has; and (2) for each distinct *sourceId* how many outgoing links this vertex has. Rank the vertices based on the average of the in- and out-degrees. Print the 100 vertex ids with the highest scores, respectively. **2 Points**

(b) Consider the `SparkPageRank.scala` implementation of the PageRank algorithm provided on Moodle to compute the PageRank value of all vertices in the graph. This time, print the 100 vertex ids with the highest PageRank values and compare them to the results you obtained in (a). **2 Points**

(c) Compare the PageRank computation you performed in (a) and (b) with the built-in PageRank API of GraphX. Again, print the 100 vertex ids with the highest PageRank values and compare them to the results you obtained in both (a) and (b). Compute *Spearman's Rank Correlation Coefficient* (refer Exercise Sheet #2 for the definition) among the complete rankings you obtain in the three cases (i.e., this time really consider the rankings of all vertices as input for the computation of this coefficient). **4 Points**

(d) Transform the RDD you created in (a) into Spark's GraphX libraries by creating two respective RDDs (one for the distinct vertices and one for the distinct edges) to initialize a new Graph instance in the GraphX APIs.

Next, perform the following principle analyses (from the lecture script) over this graph: (1) find all connected components, (2) compute the degree distribution among all vertices in this graph, (3) compute the average clustering coefficient among all vertices in the graph, and (4) finally compute the average path length among all pairs of vertices in the graph (note that you may need to sample a sufficiently small subset of vertices from the original graph to be able to run this step). **4 Points**

Big Data Analytics
Prof. Dr. Martin Theobald & Dr. Vinu Venugopal
Summer Semester 2020 – University of Luxembourg
Exercise Sheet #4 – Due on June 3, 2020 – 8:00am

Please upload your solutions on or before the due date to the provided Moodle assignment as a single zip file using your group id as the file name. Provide some brief instructions on how to run your solution to each problem in a file called `Problem_X.txt`*, and report also the most important results (such as number of results, runtimes, etc.) of your solutions to that problem within this file. Remember that all solutions may be submitted in groups of up to 3 students.*

Geo-Spatial & Temporal Data Analysis — 12 Points

Problem 1. Consider the NYC taxi trips and NYC boroughs GeoJson datasets which are available from Moodle. Also consider the sample script `RunTaxiTrips.scala` (and related classes) on Moodle as a basis to perform the following analytical tasks.

(a) Compute the number of taxi trips which (i) *started* and (ii) *ended* in *each of the NYC boroughs* over the entire period of time recorded in the data set. **2 Points**

(b) Compute the number of taxi trips which (i) *started* and (ii) *ended* in *each of the NYC boroughs* and at *each day of the week* over the entire period of time recorded in the data set. **2 Points**

(c) Modify the provided script such that it computes the *average duration between two subsequent trips* conducted by the same taxi driver *per borough* and *per hour-of-the-day* (at which the second trip starts). **2 Points**

(d) Detect potential outliers by finding taxi trips whose duration is longer than the 95% quantile of the durations of all taxi trips. **2 Points**

(e) Detect potential outliers by finding taxi trips whose duration is longer than the 95% quantile of all taxi trips normalized by the distances of the respective trips.

That is, for each taxi trip, compute its duration (e.g. in seconds) and divide this duration by the direct distance (e.g. in miles or kilometres) between the start and end point of this trip. Then sort all trips in ascending order of these ratios and cut-off all trips which are above 95% of this list to find the outliers. **4 Points**

Please make sure that you properly make use of the Spark infrastructure by loading the taxi trips and their various transformations into RDDs and by computing the requested tasks as much as possible via parallel RDD transformations.

Linking Traffic Safety & Taxi Trips Datasets — 12 Points

Problem 2. For this exercise, besides the two data sets used in Problem 1, we will additionally consider the "Motor Vehicle Collisions" open data collection (available from `https://data.cityofnewyork.us/` and via Moodle) from the New York Police Department, a CSV file which consists of records with information about collisions that occurred in New York City in 2013. Each line of this data set contains (amongst others) the following fields:

```
DATE, TIME, BOROUGH, LATITUDE, LONGITUDE, LOCATION, ON STREET NAME, CROSS
STREET NAME, OFF STREET NAME, NUMBER OF PERSONS INJURED, NUMBER OF PERSONS KILLED,
CONTRIBUTING FACTOR VEHICLE 1, CONTRIBUTING FACTOR VEHICLE 2,
VEHICLE TYPE CODE 1, VEHICLE TYPE CODE 2
```

(a) Implement a parser for the collisions data that extracts the lines (including all of the above fields) of this file into an initial RDD. Filter out useless or meaningless lines from your data set: remove all records which have no coordinates information or where the `ON STREET NAME` or `CROSS STREET NAME` is empty. **2 Points**

(b) Find the most dangerous street crossings according to the number of people that are injured or even killed in collisions which are recorded within the dataset. Return pairs of `(ON STREET NAME, CROSS STREET NAME)` together with the number of people involved (injured or killed) and a list of up to the 10 most common contributing factors (of either one of the two vehicles involved in a collision) for each such crossing. Sort all crossings in descending order of the total number of people involved in accidents. **4 Points**

(c) Based on the results of (b), analyze whether vehicles of type `SPORT UTILITY` are more frequently among the top contributing factors of accidents than vehicles of type `PASSENGER VEHICLE`. **2 Points**

(d) Finally, let us aim to find taxi trips that likely were affected by (or even possibly involved in) a collision. To do so, we first define the *affecting area* of a collision to be the area in a radius of 50 meters around the collision. Since we are not getting information about the exact trip directions, we will assume that trips follow linear surface trajectories. Thus, a taxi trip is assumed to be affected by a collision if (1) a *linear surface trajectory* from the start to the end point of the trip crosses an affecting area of a collision, and (2) the *time* of the collision also interleaves with the start to end time of the taxi trip.

The result of this query should contain all taxi trips which are found to likely be affected by a collision, together with the date, time, coordinates and contributing factors of each affecting collision. **4 Points**

Hint: See the Esri geometry API (`http://esri.github.io/geometry-api-java/javadoc/`) for a reference on the necessary distance operators.

Financial Risk Analysis **12 Points**

Problem 3. Consider the stocks and factors time-series dataset available from Moodle to solve this exercise. Also consider the sample script `RunMontoCarlo.scala` on Moodle as a basis to perform the following analytical tasks.

(a) Compute the Var and CVaR for each of the four stocks in the provided dataset by directly computing the two-week returns over only the *historical recordings* of these stocks. Compare your results with the simulated VaR and CVaR values for each of the four stocks individually.

- Which stock do you think is the safest investment according to each of the two methods?
- What is the advantage of simulating the VaR and CVar values as it is done in the provided Scala class?

4 Points

(b) Based on the provided script, analyze how the Var and CVaR values change when we consider (i) *1-day*, (ii) *1-week*, (iii) *2-week* and (iv) *1-month* returns (instead of just the 2-week returns computed by the script).

Print the respective Var and CVaR values, and plot the underlying distributions of all four series of returns you obtain from this setting via the `breeze-viz` package (similarly to how the second plot of the provided script is created). **4 Points**

(c) Assume we wish to *drop the analysis of correlations among market factors* (which is currently implemented via the Multivariate Normal Distribution in the provided script), and instead assume that all market factors are independent of each other.

Modify the provided script to accommodate this independence assumption, i.e., sample the factors f_j from each of the three distributions of historical factor returns *independently* from each other,

and feed these factors as features into the linear-regression model to predict the respective stock returns r_i.

Based on the averages of the predicted returns among the four stocks in our portfolio, compute the VaR and CVaR values now under this independence assumption. **4 Points**