

AIM-3 Scalable Data Science

WS 2020/21

Assignment 2



Technische Universität Berlin

2021

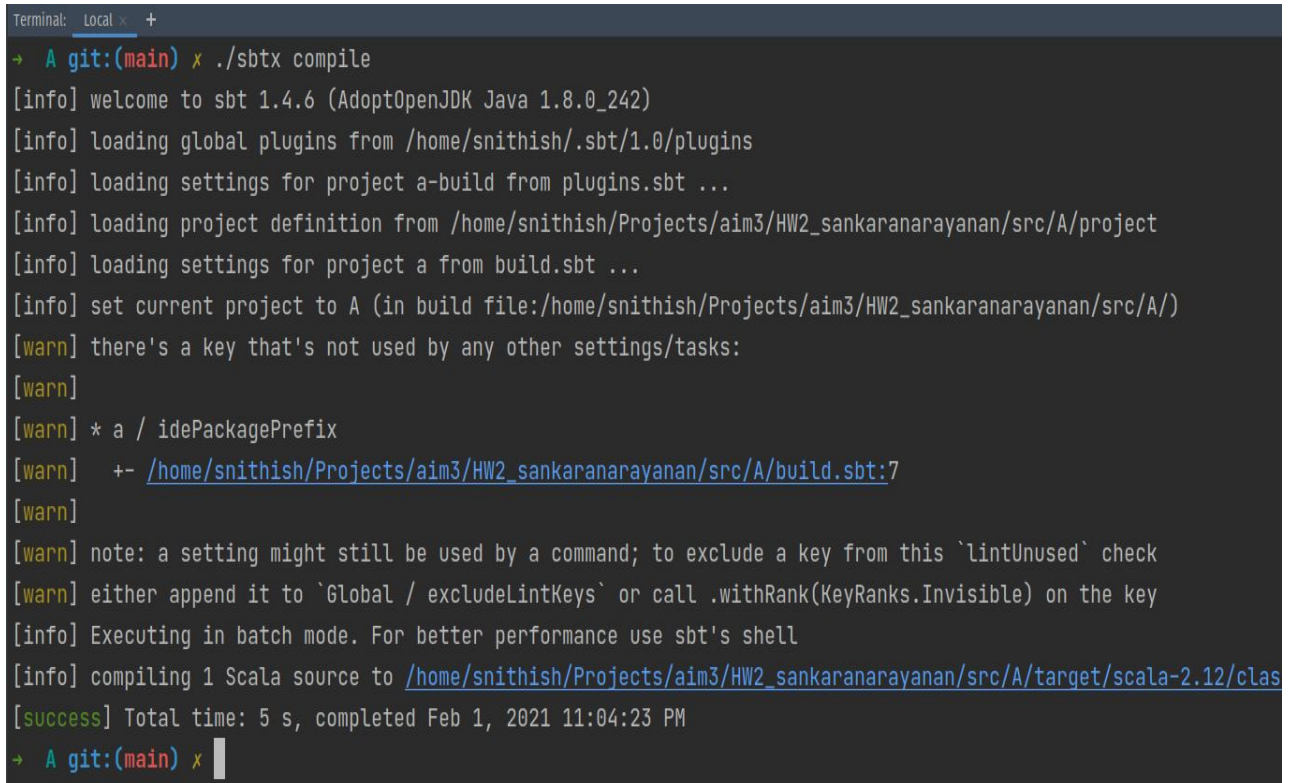
A: Classification

The task for spam classification is implemented in Spark using *Scala 2.12.13* and *sbt* is used as a build tool. Input files are also present in the data folder for convenience.

The accuracy metrics are outputted to console on invocation.

README.md file found under *src/A* contains instructions for running code.

Following screenshot shows successful compilation of code



```
Terminal: Local x +
→ A git:(main) x ./sbt compile
[info] welcome to sbt 1.4.6 (AdoptOpenJDK Java 1.8.0_242)
[info] loading global plugins from /home/snithish/.sbt/1.0/plugins
[info] loading settings for project a-build from plugins.sbt ...
[info] loading project definition from /home/snithish/Projects/aim3/HW2_sankaranarayanan/src/A/project
[info] loading settings for project a from build.sbt ...
[info] set current project to A (in build file:/home/snithish/Projects/aim3/HW2_sankaranarayanan/src/A/)
[warn] there's a key that's not used by any other settings/tasks:
[warn]
[warn] * a / idePackagePrefix
[warn] +- /home/snithish/Projects/aim3/HW2_sankaranarayanan/src/A/build.sbt:7
[warn]
[warn] note: a setting might still be used by a command; to exclude a key from this `lintUnused` check
[warn] either append it to `Global / excludeLintKeys` or call .withRank(KeyRanks.Invisible) on the key
[info] Executing in batch mode. For better performance use sbt's shell
[info] compiling 1 Scala source to /home/snithish/Projects/aim3/HW2_sankaranarayanan/src/A/target/scala-2.12/classes
[success] Total time: 5 s, completed Feb 1, 2021 11:04:23 PM
→ A git:(main) x
```

Approach:

1. We read the CSV files as dataframe and add a column “spam” to denote if the read record is spam or not.
2. We union training records for spam and no spam into a single dataframe, the same operation is done for the training set too.
3. We normalize the email text, we apply normalization as we term frequency as the basis for classification and having discrete email addresses, numbers and urls don’t add any value; Hence we generalize them. Normalization is done by the means of a Spark UDFs
 - a. All email address are converted into “normalizedEmail” string
 - b. All URLs are converted into “normalizedURL” string
 - c. Currency symbols £, \$, € and ₹ are converted into “normalizedCurrency” string

- d. Numbers are converted into “normalizedNumbers” string
4. We construct a pipeline to apply the transformation we perform.
 - a. We tokenize the words based on spaces and punctuations, this is referred to as “Bag of words”
 - b. We remove stop words like “a”, “an”, “the” as they don’t contribute any value to the text
 - c. We use a hashing term frequency transform to count the word occurrences
 - d. We calculate the inverse document frequency and this forms our feature vector
5. We train a Logistic Regression and SVM model with transformed training data
6. We use the testing set to make predictions using the model
7. We compute RoC, Accuracy, Precision and Recall metrics and display a confusion matrix for the benefit of the user.

Output:

Logistic Regression:

RoC: 0.8350036512849045

Actual \ Predicted	spam	ham
spam	104	45
ham	27	938

Accuracy: 0.9353680430879713

Precision: 0.7938931297709924

Recall: 0.697986577181208

SVM:

RoC: 0.9114441701151025

Actual \ Predicted	spam	ham
spam	124	25
ham	9	956

Accuracy: 0.9694793536804309

Precision: 0.9323308270676691

Recall: 0.8322147651006712

B: Clustering

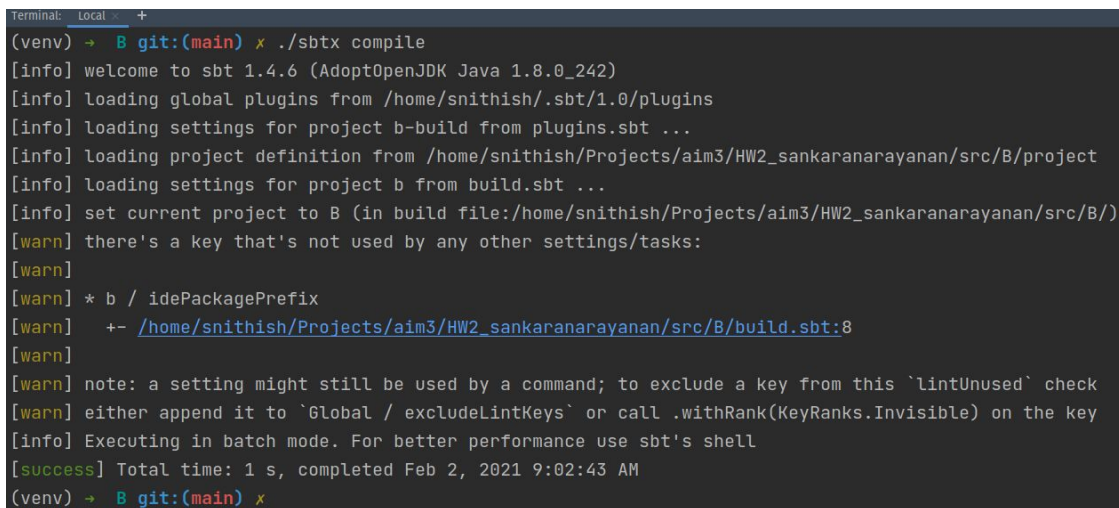
The task for k means clustering on MNIST handwritten numbers dataset is implemented in Spark using *Scala 2.12.13* and *sbt* is used as a build tool. Input files are also present in the data folder for convenience.

The centroids are saved in *kmeans_centroid.csv* file within the data folder.

The script to visualize all the centroids is present inside folder *viz_scripts* the results of visualization are also saved there.

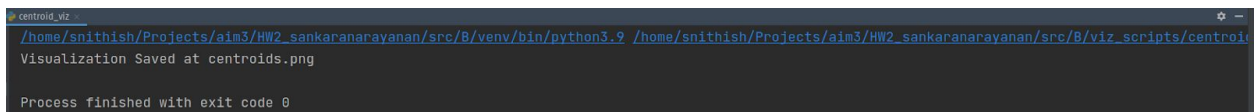
README.md file found under *src/B* contains instructions for running code.

Following screenshot shows successful compilation of code.



```
Terminal: Local +
(venv) → B git:(main) x ./sbt compile
[info] welcome to sbt 1.4.6 (AdoptOpenJDK Java 1.8.0_242)
[info] loading global plugins from /home/snithish/.sbt/1.0/plugins
[info] loading settings for project b-build from plugins.sbt ...
[info] loading project definition from /home/snithish/Projects/aim3/HW2_sankaranarayanan/src/B/project
[info] loading settings for project b from build.sbt ...
[info] set current project to B (in build file:/home/snithish/Projects/aim3/HW2_sankaranarayanan/src/B/)
[warn] there's a key that's not used by any other settings/tasks:
[warn]
[warn] * b / idePackagePrefix
[warn] +- /home/snithish/Projects/aim3/HW2_sankaranarayanan/src/B/build.sbt:8
[warn]
[warn] note: a setting might still be used by a command; to exclude a key from this `lintUnused` check
[warn] either append it to `Global / excludeLintKeys` or call .withRank(KeyRanks.Invisible) on the key
[info] Executing in batch mode. For better performance use sbt's shell
[success] Total time: 1 s, completed Feb 2, 2021 9:02:43 AM
(venv) → B git:(main) x
```

Following screenshot shows the successful execution of the visualization script.



```
centroid_viz
/home/snithish/Projects/aim3/HW2_sankaranarayanan/src/B/venv/bin/python3.9 /home/snithish/Projects/aim3/HW2_sankaranarayanan/src/B/viz_scripts/centroid_viz.py
Visualization Saved at centroids.png
Process finished with exit code 0
```

Approach:

Clustering

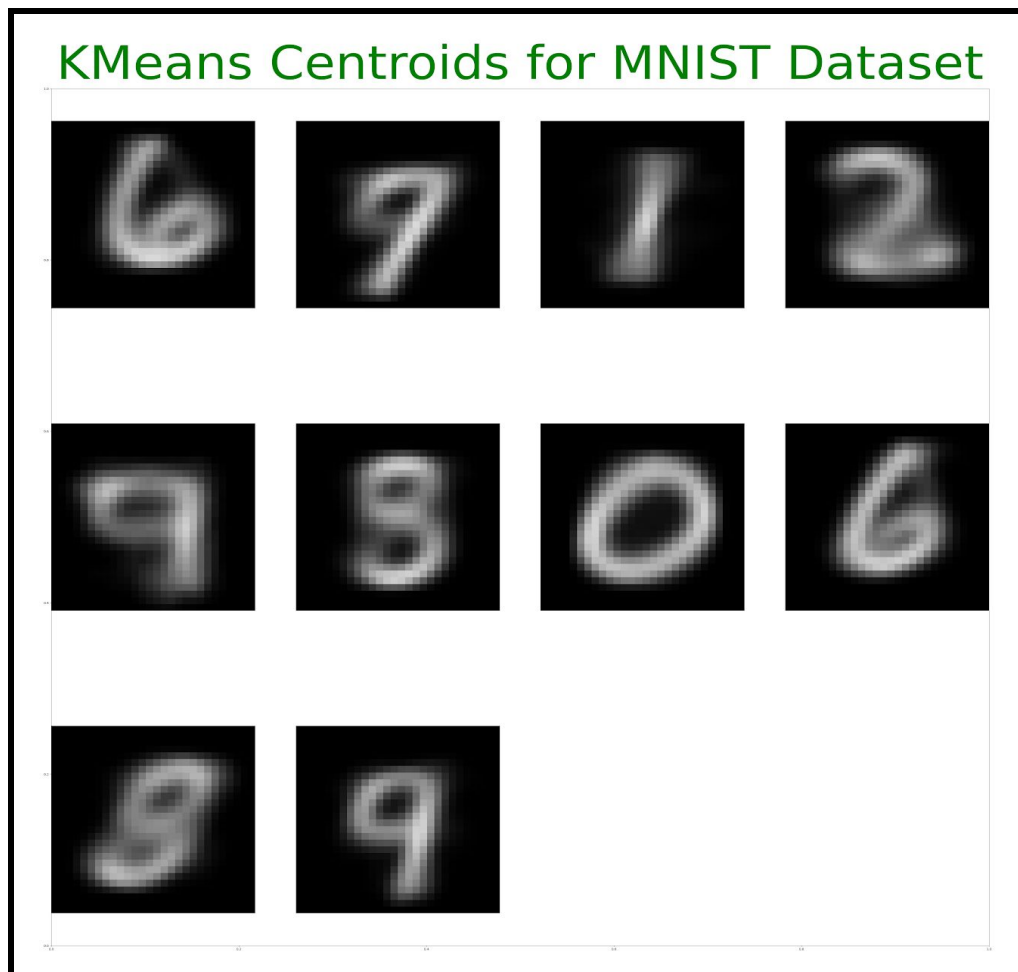
1. Read the mnist CSV files as raw text files
2. Convert each line into a Row of dense vectors
3. Create a dataframe from RDD from step 2

4. Since the grayscale values can range from 0 to 255 and we don't want to give undue weightage during distance computation, we apply mean scaling
5. Scaled dataset is used to training KMeans model
6. We set the number of clusters as 10 based on intuition, there are 10 numbers (0-9) and the aim was to see if vectors with the same image are clustered together.
7. The final centroid values are scaled back and written to a CSV file

Centroid Visualization

1. We read the centroids CSV file
2. Create a matplotlib plot with 10 subplots within it, one for each centroid
3. We reshape the flattened MNIST data into 28x28 numpy array
4. This array is then projected as a grayscale image
5. The plot is saved

Output



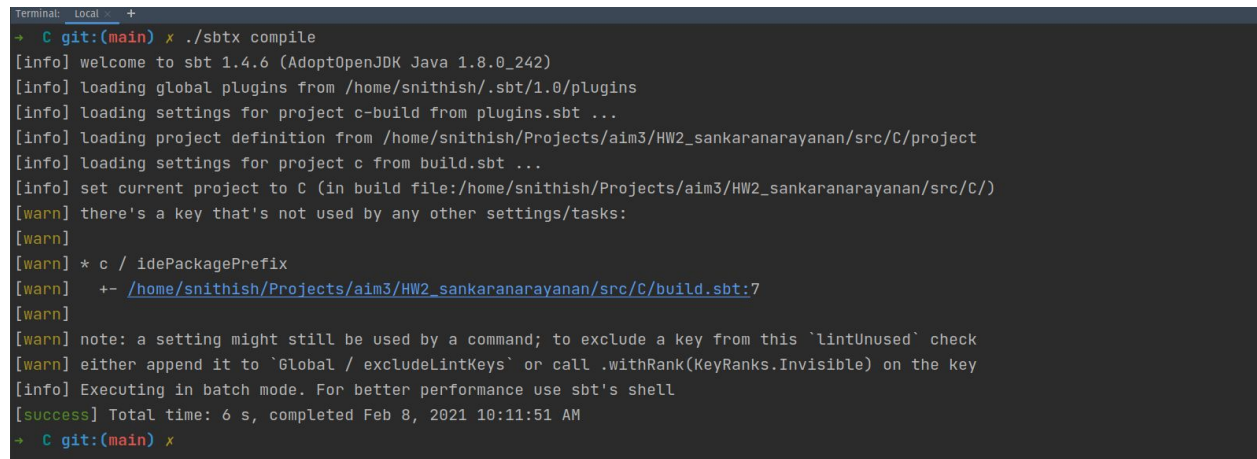
C: Recommendation Systems

The task for creating a movie recommender system is implemented in Spark using *Scala 2.12.13* and *sbt* is used as a build tool. Input files are included in the data folder for convenience. The output for exploration and final recommendations are written back to the data folder.

The root mean square error metrics are outputted to console on invocation.

README.md file found under *src/C* contains instructions for running code.

Following screenshot shows successful compilation of code.



```
terminal: Local +
+ C git:(main) x ./sbt compile
[info] welcome to sbt 1.4.6 (AdoptOpenJDK Java 1.8.0_242)
[info] loading global plugins from /home/snithish/.sbt/1.0/plugins
[info] loading settings for project c-build from plugins.sbt ...
[info] loading project definition from /home/snithish/Projects/aim3/HW2_sankaranarayanan/src/C/project
[info] loading settings for project c from build.sbt ...
[info] set current project to C (in build file:/home/snithish/Projects/aim3/HW2_sankaranarayanan/src/C/)
[warn] there's a key that's not used by any other settings/tasks:
[warn]
[warn] * c / idePackagePrefix
[warn] +- /home/snithish/Projects/aim3/HW2_sankaranarayanan/src/C/build.sbt:7
[warn]
[warn] note: a setting might still be used by a command; to exclude a key from this 'lintUnused' check
[warn] either append it to 'Global / excludeLintKeys' or call .withRank(KeyRanks.Invisible) on the key
[info] Executing in batch mode. For better performance use sbt's shell
[success] Total time: 6 s, completed Feb 8, 2021 10:11:51 AM
+ C git:(main) x
```

Approach:

Exploration

Common Steps:

1. We read the movies and ratings csv files into dataframes
2. Join movies and rating dataframes on movieId column to get movieRating dataframe

01. Printing the names of the top-10 movies with the largest number of ratings

- a. Group movieRating dataframe on movieId column
- b. Apply aggregation to select the first movie title and count of ratings received
- c. Sort descending based on the number of ratings received
- d. Select the title column and limit results to 10
- e. Write results to *data/c1.csv*

02. Printing the names of the top-10 movies with the highest average rating grouped by genre

- a. Since genres is a multi-valued column which genres separated using '|', we first by creating one row for each movie, genre combination

- b. We group by genre and movielf
- c. We choose first movie title and compute average rating
- d. Sort descending based on average rating and choose top 10
- e. Write results to *data/c2.csv*

03. Finding the common support for all pair of the first 100 movies

- a. Select movielf of the first 100 movies
- b. Cross join 100 movielf with itself to generate all possible pairs
- c. Co-Pairs are joined with ratings such that, first and second movielf are matched and such that users have rated both movies
- d. We group by first movielf and the second movielf and count ratings
- e. Write results to *data/c3.csv*

Baseline Predictor

We build a model that implements Spark's model abstraction.

1. We compute the mean ratings for all movie
2. For each user, user bias is the difference between overall average ratings and average user rating
3. For each movie, movie bias is the difference between overall average ratings and average movie rating
4. Baseline model is created with overall means, user and movie bias

Collaborative Filtering

1. Spark supports Alternating Least Squares (ALS) matrix factorization for collaborative filtering
2. We initialize ALS with 50 maximum iterations and 25 latent factors
3. We instruct spark to checkout frequently to avoid StackOverflow exception resulting from deep recursions performed by ALS

Evaluation

1. We randomly split the rating dataset into train (70%) and test sets (30%)
2. Training the models using training set
3. We use the test set to make predictions
4. We use regression evaluator to calculate root mean square error

Root Mean Square Error (RMSE)	
Baseline	0.9226554416827888
Collaborative Filtering - ALS	0.8953246814811533

Recommendations

1. Since the ALS model has the lowest RMSE we use it for recommending best movie for all users
2. Results are written to *data/recommendations.csv*

D: Spatial Data Analysis

The task for counting tweets per neighbourhood is implemented in Spark using *Scala 2.12.13* and *sbt* is used as a build tool. The output for all possible combinations are written back to the data folder.

README.md file found under *src/D* contains instructions for running code.

Following screenshot shows successful compilation of code.

```
* D git:(main) x ./sbt compile
[info] welcome to sbt 1.4.6 (AdoptOpenJDK Java 1.8.0_242)
[info] loading global plugins from /home/snithish/.sbt/1.0/plugins
[info] loading settings for project d-build from plugins.sbt ...
[info] loading project definition from /home/snithish/Projects/aim3/HW2_sankaranarayanan/src/D/project
[info] loading settings for project d from build.sbt ...
[info] set current project to D (in build file:/home/snithish/Projects/aim3/HW2_sankaranarayanan/src/D/)
[warn] there's a key that's not used by any other settings/tasks:
[warn]
[warn] * d / idePackagePrefix
[warn] +- /home/snithish/Projects/aim3/HW2_sankaranarayanan/src/D/build.sbt:7
[warn]
[warn] note: a setting might still be used by a command; to exclude a key from this `lintUnused` check
[warn] either append it to `Global / excludeLintKeys` or call .withRank(KeyRanks.Invisible) on the key
[info] Executing in batch mode. For better performance use sbt's shell
[info] compiling 1 Scala source to /home/snithish/Projects/aim3/HW2_sankaranarayanan/src/D/target/scala-2.12/classes ...
[success] Total time: 5 s, completed Feb 8, 2021 12:22:51 PM
* D git:(main) x
```

Approach:

1. Since our objective is to count the number of tweets, and we notice that there are multiple tweets from the same location, we read the location coordinates and aggregate

them to get the total number of tweets for each point. This reduces the number of points that will be involved in the joint operation

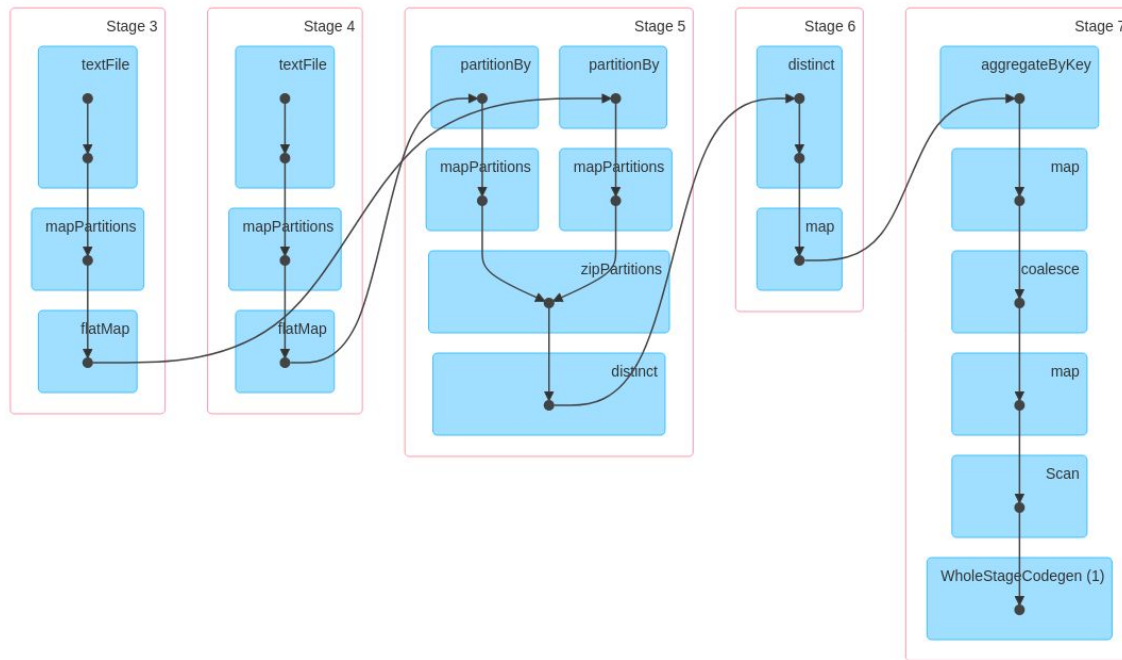
2. The number of tweets is treated as user data and we convert point into WKT representation for easy loading to geometric RDD
3. We read the neighborhood WKT file and assign line number as user data
4. We partition tweetRDD using RTree partitioning type, we use this since we are performing a join with polygons; RTree will construct enveloping polygons
5. We partition neighborhoodRdd using RTree too
6. Depending on the scenario, we construct indexes on either tweetRdd or neighborhoodRdd, both or none. We treat D(1) as a special case of D(2) where no indexes are applied
7. Since we are looking at a ST_CONTAINS inequivalent query, we sent join param to not consider boundary intersection and be eliminate duplicates
8. After join, we extract the line number user data from neighborhoodRdd and count from tweetRdd
9. We apply reduce operator to find total number of tweets per neighborhood
10. Results are written to appropriate csv's in *data* folder
11. We compute time taken for each combination and save it inside *data/metrics.csv*
12. We present screenshots from spark web UI below to flow operations involved

The screenshot displays the Spark Web UI interface. At the top, there's a navigation bar with tabs: Jobs, Stages, Storage, Environment, Executors, and SQL. The 'Jobs' tab is active. Below the navigation bar, the page title is 'Spark Jobs (?)'. On the left, there's a sidebar with links: 'User: snithish', 'Total Uptime: 9.9 min', 'Scheduling Mode: FIFO', 'Active Jobs: 1', 'Completed Jobs: 3', 'Event Timeline', and 'Active Jobs (1)'. The main content area shows a table of jobs. The first job is '3', with a description 'csv at SpatialWithoutIndex.scala:50', submitted at '2021/01/26 00:27:43', duration '7.7 min', and '2/5' stages. The tasks bar shows '77/195 (4 running)'. Below this, there's a section for 'Completed Jobs (3)' with a table showing three jobs: '2' (take at RddReader.java:29), '1' (collect at SpatialRDD.java:258), and '0' (aggregate at SpatialRDD.java:520). Each job has its submission time, duration, and stage progress (1/1). The tasks bar for each completed job shows '1/1', '62/62', and '62/62' respectively. The bottom of the page shows pagination controls: 'Page: 1', '1 Pages. Jump to 1', 'Show 100 items in a page. Go'.

Job id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
3	csv at SpatialWithoutIndex.scala:50 (kill)	2021/01/26 00:27:43	7.7 min	2/5	77/195 (4 running)

Job id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
2	take at RddReader.java:29	2021/01/26 00:27:40	15 ms	1/1	1/1
1	collect at SpatialRDD.java:258	2021/01/26 00:26:55	44 s	1/1	62/62
0	aggregate at SpatialRDD.java:520	2021/01/26 00:25:37	1.3 min	1/1	62/62

▼ DAG Visualization



Result:

	No Index	R-Tree	Quad-Tree	Location	Time
Points	x			d_one.csv	886299.70ms
Polygon	x				
Points	x			d_neighborhood_rtree.csv	785361.71ms
Polygon		x			
Points		x		d_tweet_rtree.csv	771302.71ms
Polygon	x				
Points		x		d_rtree.csv	786113.89ms
Polygon		x			
Points			x	d_tweet_quadtree.csv	803977.03ms
Polygon	x				

Points	x			d_neighborhood_quadtrees.csv	790071.20ms
Polygon			x		
Points			x	d_quadtrees.csv	771422.97ms
Polygon			x		

We observe from the results that RTree index on points had reduced execution times the most and without any indexes execution times are significantly faster.

After analyzing spatialJoin logic from Sedona, we can observe that even if both spatial geometries are indexed, join makes use of only one of them for index lookups.