Big Data Processing 2017/2018 2nd Assignment Report

BDP-05: LARGE-SCALE CLUSTERING

Part A - Data Extraction.
Part B - Data Normalisation+Feature selection.
Part C - Clustering in pyspark.
Part D - Benchmarking against mllib.
Part E - Conclusion

Group AI:

Liev Garcia
Eurico Covas
Nadir Badji
Ines Nolasco
John Flynn

Individual Contributions:

- Proposal submission (everybody)
- Project Start discussions and ideas (everybody)
- Data Munging (Liev Garcia)
- Data Normalisation and initial statistical review (Eurico Covas)
- K-Means custom implementation (Ines Nolasco & Nadir Badji)
- K-Means implementation in Mllib (John Flynn)
- Experiments and Result analysis (Ines Nolasco & Eurico Covas)
- Report writing (everybody based on their section)
- Document review (everybody)

Objectives

1. To successfully extract features from raw unstructured big data;
2. To successfully do data analysis;
3. To successfully do feature engineering and feature pruning;
4. To successfully use pyspark and dataframes;
5. To successfully implement K Means explicitly;
6. To successfully obtain the same results as mllib (except performance);
7. To successfully identify the K means elbow shape;
8. Given a K means elbow shape, to project successfully the clusters on a 2D subspaces whereby the clusters are clearly recognizable;
9. To meet the original coursework deadline of 15 December 2017.

Files structure

All the produced code and results are gathered in the main *GroupAI-submission* folder following the structure:

GroupAI-submission:

GroupAI_report.pdf: this report

resultsAnalysis_comparisons.xlsx: spreadsheet where the main analysis for part C and D are done.

- PartA-Feature Extraction:
  - Code to run part A
- PartB-DataNormalization_FeatureSelection:
  - Code to run partB: ***get_all_features_histograms.py***
  - **output_users_fields_what_to_do.xlsx**: Spreadsheet resuming what was done to each feature.
- PartC-ClusteringSpark
  - Code to run part C: ***Kmeans_ForStackFile.py***
  - Results:
    - Output files for each value of K
    - ClusterPlots2D: code ***plot_clusters.py*** and resulting plots of clusters found for k=4 and k=3
- PartD-Mlib
  - Code to run part D: ***Kmeans_withMlib.py***
  - Results: output files for each value of K

Introduction and Structure

Clustering is a very interesting technique to find patterns in data and specially when dealing with a lot of features, where natural groups are not evident, clustering is very attractive. But clustering high dimensionality data and big datasets is not without problems. Here we will use Spark to analyse Stackoverflow data and implement K-Means algorithm to cluster this data. In part A we extract the data with pyspark dataframes, we extract data based on user id. After that, in part B, we analyse the data and in particular we decide on the form of scaling per feature extracted. We use python to create histograms of all features in both semilog y format and histograms of the log of the actual feature. We also analyse cross correlation to avoid collinear variables. We then scale and pass the data to two sets of pyspark code (part C and part D), one where we implement explicitly the K means clustering in pyspark code, and another where we run against mllib, for checking our results and for benchmarking the performance of our code. We conclude in part E.

Part A - Data Extraction

Stack Overflow (SO) makes its data available through this site:
https://archive.org/details/stackexchange

The files published, however, will require some work to make them useful for a k-means clustering exercise.

We've decided that Part A would need to, at a high level do the following:

- Read files
- Extract useful features
- Add some summary features through joins
- Output for the next step in a readable format

Easy! They're XML files and there's a library available for us: **Spark-XML**. This library will allow us to load the files into dataframes, from there we can add/remove columns easily and create joins.

Oh Wait! There's a problem. SO uses self-closing XML tags, this is not standard compliant XML and therefore not supported by the library. and that's not supported by the library.
In fact, there's a couple open issues: and a some work done to support this, but this has not been merged into the master branch of the library.

https://github.com/databricks/spark-xml/issues/92
https://github.com/databricks/spark-xml/issues/190

And a pull request waiting to be merged.
https://github.com/databricks/spark-xml/pull/149

Anyway, since we cannot use this 'shortcut' something else will need to be done.

As a result the following repo was created (see README for run instruction and acknowledgments):
https://github.com/lievcin/pystack-injest

There're a few jobs this program can run:

- badges_to_csv
- answers_to_csv
- comments_to_csv
- posthistory_to_csv
- postlinks_to_csv
- questions_to_csv
- users_to_csv

and a bigger consolidation job:

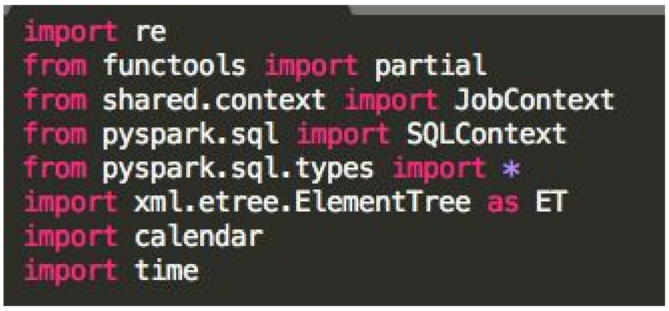
- combine_from_csv

We'll cover here the details of one of the parsing to csv jobs to illustrate (as they all work very similarly) and then cover the combination job.

***_to_csv jobs:**

These jobs can be illustrated by the file in **/src/jobs/answers_to_csv/__init__.py**

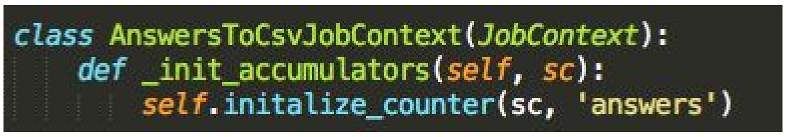
First import specific libraries and dependencies the job will need:



```
import re
from functools import partial
from shared.context import JobContext
from pyspark.sql import SQLContext
from pyspark.sql.types import *
import xml.etree.ElementTree as ET
import calendar
import time
```

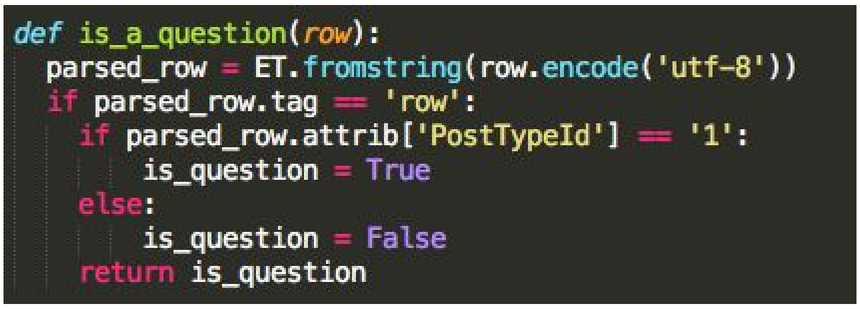
Notice the **shared.context** line, what we're designing here is not a one-off run task, but something that can be broken down into several components to be run when/if required, that can be updated and changed independently as well. But they're all jobs inside our application and therefore share 'stuff'.

Initialise our JobContext:



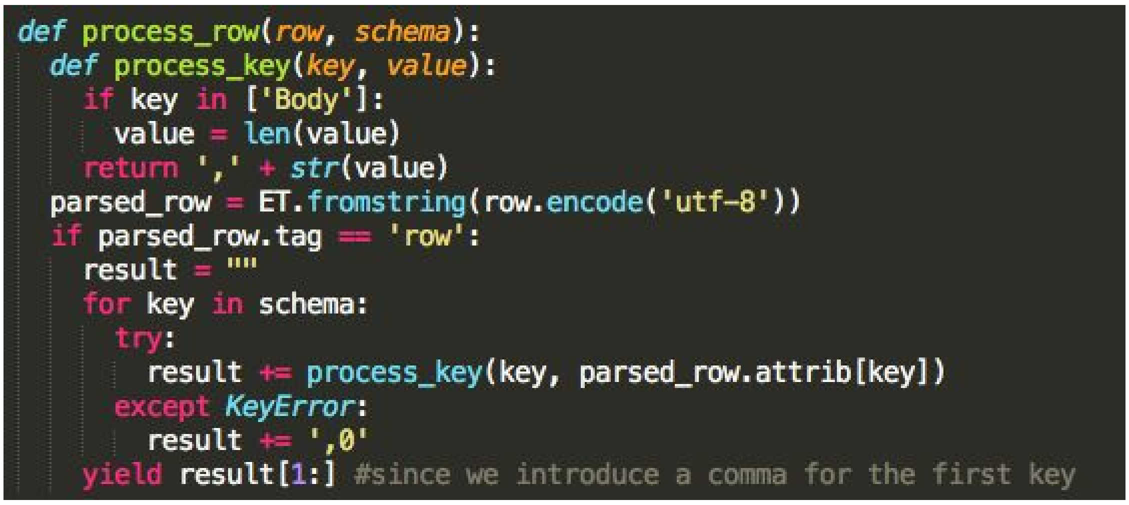
```
class AnswersToCsvJobContext(JobContext):
    def _init_accumulators(self, sc):
        self.initalize_counter(sc, 'answers')
```

We will also require additional methods for the processing of our RDDs. We define a method that tells us if something is a question or not (only required for questions and answers since they'll in the same Posts file)



```python
def is_a_question(row):
  parsed_row = ET.fromstring(row.encode('utf-8'))
  if parsed_row.tag == 'row':
    if parsed_row.attrib['PostTypeId'] == '1':
        is_question = True
    else:
        is_question = False
    return is_question
```

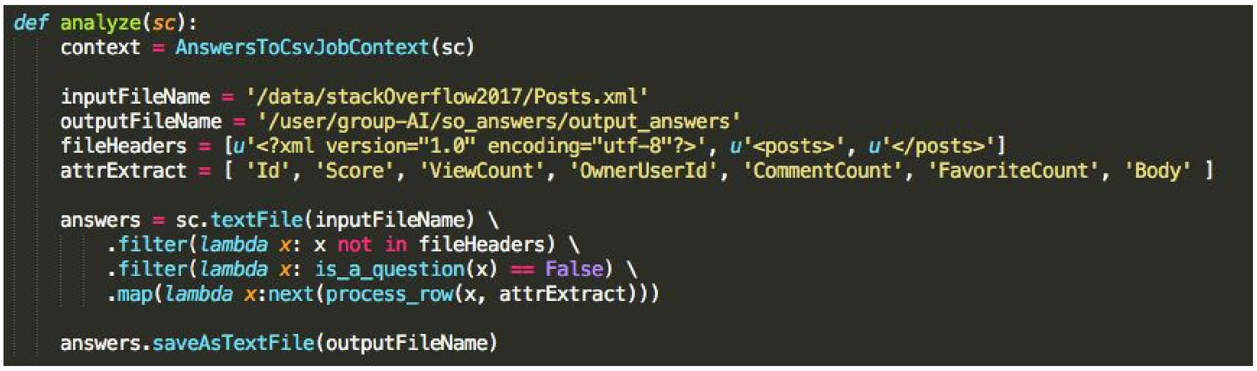
And the method to process each row and extract its attributes:



```python
def process_row(row, schema):
  def process_key(key, value):
    if key in ['Body']:
      value = len(value)
    return ',' + str(value)
  parsed_row = ET.fromstring(row.encode('utf-8'))
  if parsed_row.tag == 'row':
    result = ""
    for key in schema:
      try:
        result += process_key(key, parsed_row.attrib[key])
      except KeyError:
        result += ',0'
    yield result[1:] #since we introduce a comma for the first key
```

Since we cannot use the Spark-XML library we've had to revert back to **xml.etree.ElementTree** standard Python library for XML processing. The idea is that we can fee each row to this library (which does understand self-closing tags) in order to retrieve our attributes.

And then our master method:



```python
def analyze(sc):
    context = AnswersToCsvJobContext(sc)

    inputFileName = '/data/stackOverflow2017/Posts.xml'
    outputFileName = '/user/group-AI/so_answers/output_answers'
    fileHeaders = [u'<?xml version="1.0" encoding="utf-8"?>', u'<posts>', u'</posts>']
    attrExtract = [ 'Id', 'Score', 'ViewCount', 'OwnerUserId', 'CommentCount', 'FavoriteCount', 'Body' ]

    answers = sc.textFile(inputFileName) \
        .filter(lambda x: x not in fileHeaders) \
        .filter(lambda x: is_a_question(x) == False) \
        .map(lambda x:next(process_row(x, attrExtract)))

    answers.saveAsTextFile(outputFileName)
```

Here we define a few simple variables and the follow a simple flow:
- Load file
- Remove headers and tail
- Remove questions (if we're dealing with answers)
- Process each row's XML
  - For each attribute defined in attrExtract list retrieve from XML
  - Comma separate the output.
- Save the result as file (which now is comma-separated)

With small variations this is what is done for each of the other files.

combine_from_csv job:

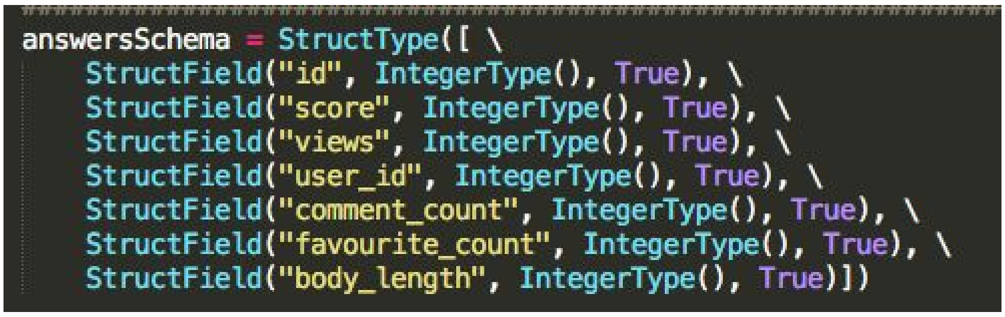
The code for this job is under **`/src/jobs/combine_from_csv`**

When launching this job we will need to load the Spark-CSV library as well.

```
spark-submit --executor-memory 7G --packages com.databricks:spark-csv_2.10:1.5.0 --py-files jobs.zip main.py --job combine_from_csv
```

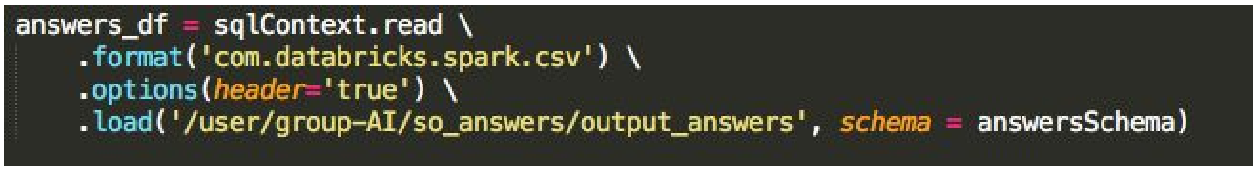
This module is a bit different from the others and relies on there being outputs to the previous jobs. The idea is to load the CSV files we now have with interesting attributes into dataframes and consolidate the files by user_id.

When defining a dataframe we will need to specify the schema (unless we want the library to infer from data, but a decision was made to be explicit here).



```
answersSchema = StructType([ \
    StructField("id", IntegerType(), True), \
    StructField("score", IntegerType(), True), \
    StructField("views", IntegerType(), True), \
    StructField("user_id", IntegerType(), True), \
    StructField("comment_count", IntegerType(), True), \
    StructField("favourite_count", IntegerType(), True), \
    StructField("body_length", IntegerType(), True)])
```

With this schema we can now load a dataframe for answers.

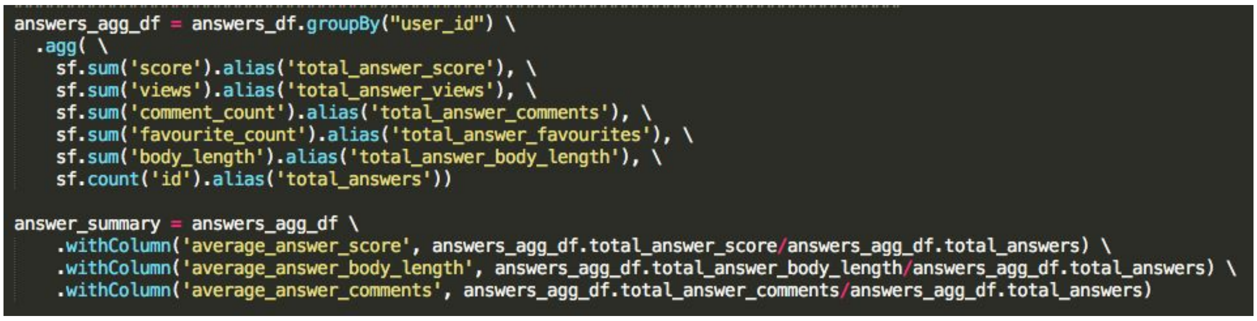


```
answers_df = sqlContext.read \
    .format('com.databricks.spark.csv') \
    .options(header='true') \
    .load('/user/group-AI/so_answers/output_answers', schema = answersSchema)
```

We do the same for the other objects like Users, Comments, Badges, Questions, etc.

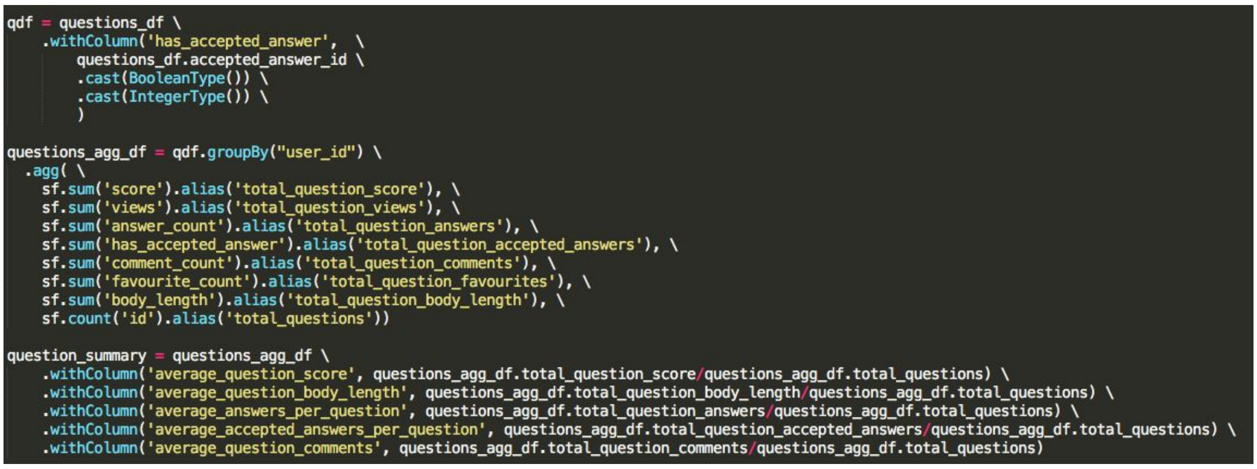
But before joining them all together it would be good to perform some aggregation on the answers and questions dataframes, that way we can summarize those two dataframes per user_id, since until then they have both user_id and post_id.
We could skip this and join to the users dataframe and perform the aggregation then, but this would be an unnecessarily expensive join with no additional gain.

In the case of the answers_df we can further process it to create a summarized view with more helpful attributes.



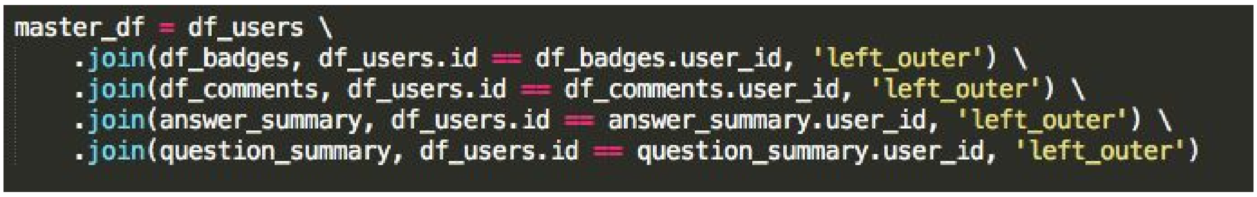
```
answers_agg_df = answers_df.groupBy("user_id") \
  .agg( \
    sf.sum('score').alias('total_answer_score'), \
    sf.sum('views').alias('total_answer_views'), \
    sf.sum('comment_count').alias('total_answer_comments'), \
    sf.sum('favourite_count').alias('total_answer_favourites'), \
    sf.sum('body_length').alias('total_answer_body_length'), \
    sf.count('id').alias('total_answers'))

answer_summary = answers_agg_df \
    .withColumn('average_answer_score', answers_agg_df.total_answer_score/answers_agg_df.total_answers) \
    .withColumn('average_answer_body_length', answers_agg_df.total_answer_body_length/answers_agg_df.total_answers) \
    .withColumn('average_answer_comments', answers_agg_df.total_answer_comments/answers_agg_df.total_answers)
```

And the same for questions



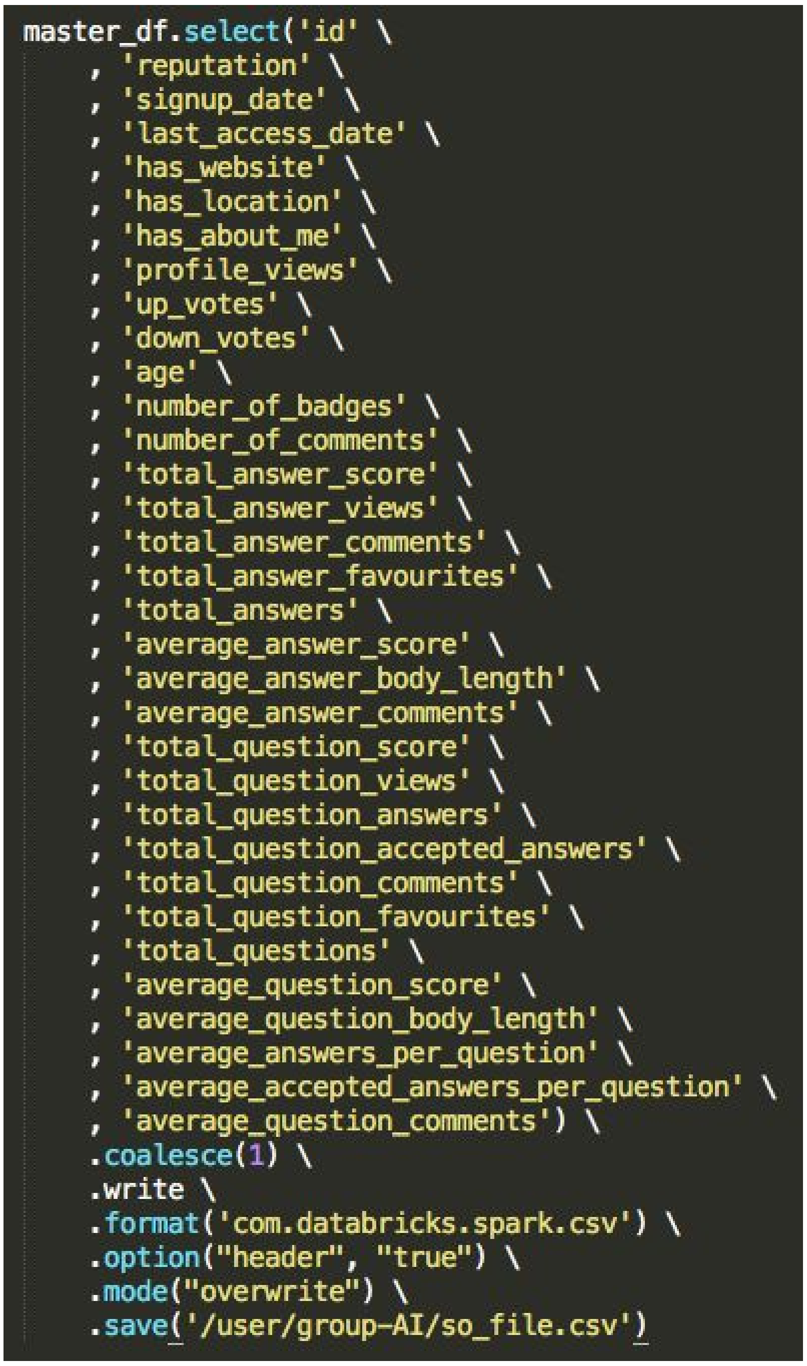
```
qdf = questions_df \
    .withColumn('has_accepted_answer',  \
        questions_df.accepted_answer_id \
        .cast(BooleanType()) \
        .cast(IntegerType()) \
        )

questions_agg_df = qdf.groupBy("user_id") \
  .agg( \
    sf.sum('score').alias('total_question_score'), \
    sf.sum('views').alias('total_question_views'), \
    sf.sum('answer_count').alias('total_question_answers'), \
    sf.sum('has_accepted_answer').alias('total_question_accepted_answers'), \
    sf.sum('comment_count').alias('total_question_comments'), \
    sf.sum('favourite_count').alias('total_question_favourites'), \
    sf.sum('body_length').alias('total_question_body_length'), \
    sf.count('id').alias('total_questions'))

question_summary = questions_agg_df \
    .withColumn('average_question_score', questions_agg_df.total_question_score/questions_agg_df.total_questions) \
    .withColumn('average_question_body_length', questions_agg_df.total_question_body_length/questions_agg_df.total_questions) \
    .withColumn('average_answers_per_question', questions_agg_df.total_question_answers/questions_agg_df.total_questions) \
    .withColumn('average_accepted_answers_per_question', questions_agg_df.total_question_accepted_answers/questions_agg_df.total_questions) \
    .withColumn('average_question_comments', questions_agg_df.total_question_comments/questions_agg_df.total_questions)
```

Our final step is to perform our join with the other dataframes under a new dataframe:



```
master_df = df_users \
    .join(df_badges, df_users.id == df_badges.user_id, 'left_outer') \
    .join(df_comments, df_users.id == df_comments.user_id, 'left_outer') \
    .join(answer_summary, df_users.id == answer_summary.user_id, 'left_outer') \
    .join(question_summary, df_users.id == question_summary.user_id, 'left_outer')
```

As a last step we want to select relevant attributes and save them back to HDFS. Optionally we can consolidate the file into one, this takes considerably longer, although the final output is a bit nicer for the next step as is single file with headers.



```
master_df.select('id' \
    , 'reputation' \
    , 'signup_date' \
    , 'last_access_date' \
    , 'has_website' \
    , 'has_location' \
    , 'has_about_me' \
    , 'profile_views' \
    , 'up_votes' \
    , 'down_votes' \
    , 'age' \
    , 'number_of_badges' \
    , 'number_of_comments' \
    , 'total_answer_score' \
    , 'total_answer_views' \
    , 'total_answer_comments' \
    , 'total_answer_favourites' \
    , 'total_answers' \
    , 'average_answer_score' \
    , 'average_answer_body_length' \
    , 'average_answer_comments' \
    , 'total_question_score' \
    , 'total_question_views' \
    , 'total_question_answers' \
    , 'total_question_accepted_answers' \
    , 'total_question_comments' \
    , 'total_question_favourites' \
    , 'total_questions' \
    , 'average_question_score' \
    , 'average_question_body_length' \
    , 'average_answers_per_question' \
    , 'average_accepted_answers_per_question' \
    , 'average_question_comments') \
    .coalesce(1) \
    .write \
    .format('com.databricks.spark.csv') \
    .option("header", "true") \
    .mode("overwrite") \
    .save('/user/group-AI/so_file.csv')
```

Part B - Data Normalisation + Feature Selection.

In this section we analyse the raw features extracted from Part A and assess if and what normalisation we need to apply to the data.

The raw data from Part A consists of a single csv file with 7,617,190 lines and the following 33 fields:

- id
- reputation
- signup_date
- last_access_date
- has_website
- has_location
- has_about_me
- profile_views
- up_votes
- down_votes
- age
- number_of_badges
- number_of_comments
- total_answer_score
- total_answer_views
- total_answer_comments
- total_answer_favourites
- total_answers
- average_answer_score
- average_answer_body_length
- average_answer_comments
- total_question_score
- total_question_views
- total_question_answers
- total_question_accepted_answers
- total_question_comments
- total_question_favourites
- total_questions
- average_question_score
- average_question_body_length
- average_answers_per_question
- average_accepted_answers_per_question
- average_question_comments

The main field (the unique key) is the id field,which is the stackoverflow user id.

From these 32 fields (excluding id) we created an engineered feature:

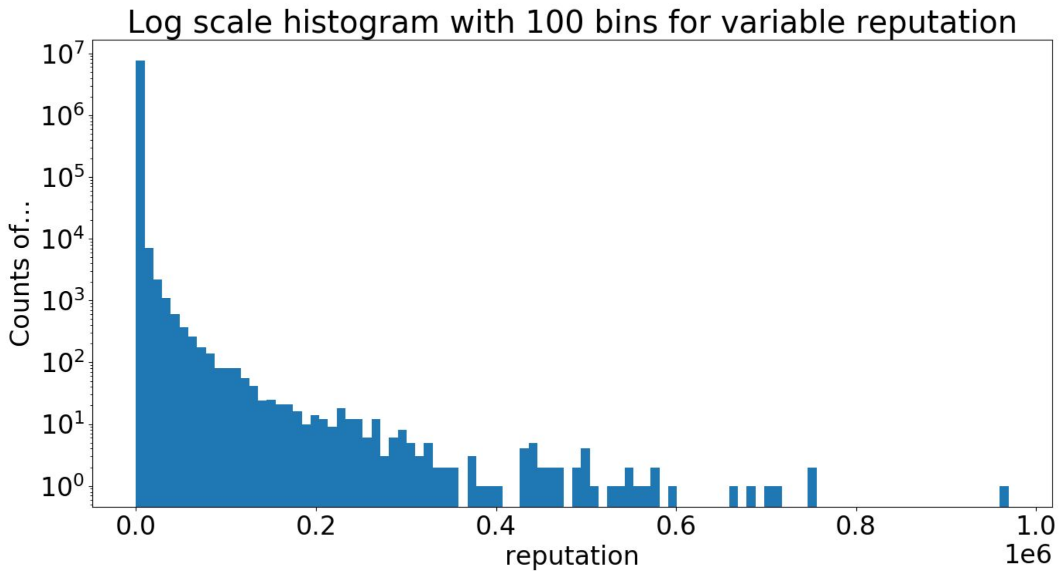
```
been_user_number_days = np.array(list(map(lambda x,y: (y-x).days,
signup_date,last_access_date)))
```

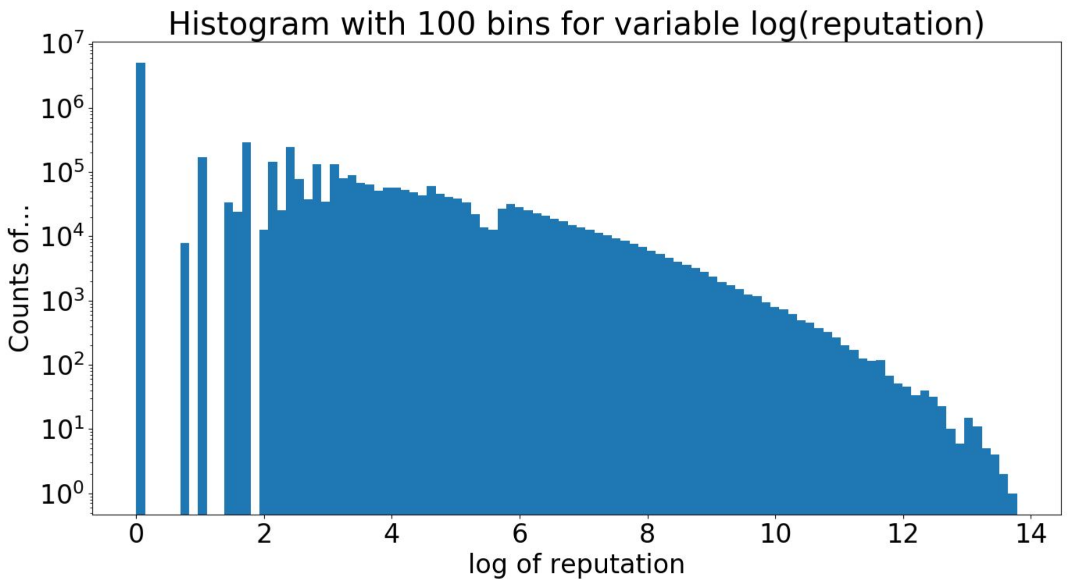
We then remove the **signup_date** and **last_access_date**. We now have 32-2+1 = 31 fields.

We then plot all histograms of this data. In order to be able to see clearly, we plot 3 histograms per field using matplotlib and numpy:

- A histogram using plt.hist(x, bins=mybins);
- A histogram using the above plus plt.yscale('log', nonposy='clip') to do a semilogy plot, in case the variable follows a power law distribution, and so it spans many order of magnitude in value;
- A histogram using `plt.hist(ma.log(x).filled(0), bins=mybins)` to get the distribution of the logarithm of the feature.

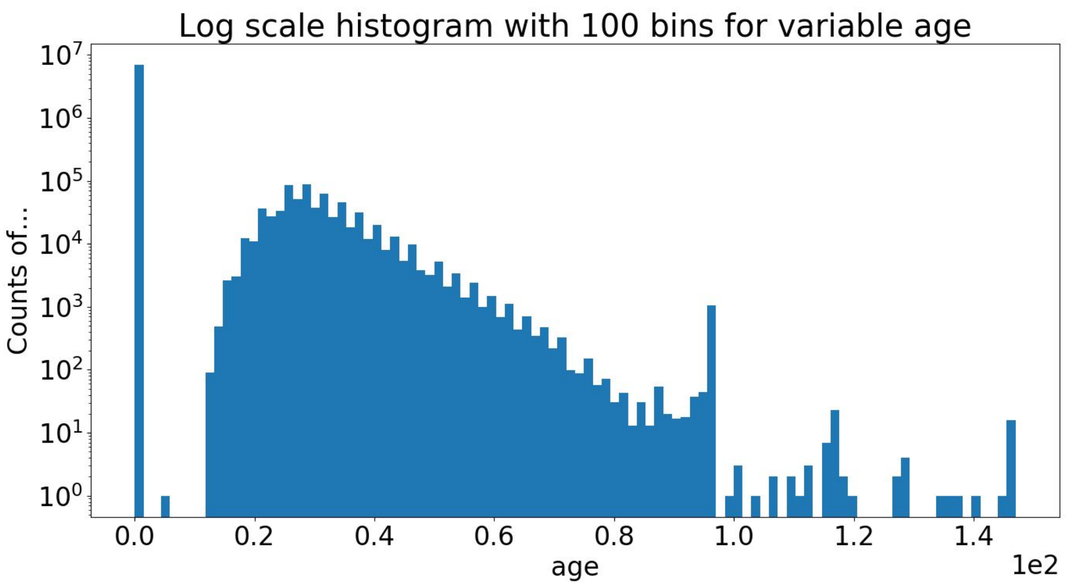
From that analysis we see most variables follow a power law distribution and we believe it is reasonable to apply first the logarithm before doing a z-score scaling [4].
An example of a variable that follows a power law, e.g. we have the following two histograms of the variable reputation (first a semilogy plot of the histogram and second histogram of the logarithm of reputation):





It gives some indication that we should apply logarithm to this variable before scaling. Notice that there are some indications of possible clustering at low end and at high end too.

On the other hand, for the feature age, we can see it does not make sense to apply any logarithmic scaling at all:



In its raw format, age has already some clear clusters formed.

For the variables to which we apply the logarithm, we ensure we get no NaN by doing the following: `my_list_features[i]+=-np.min(my_list_features[i])+1,` for all i.

To summarize, the variables to which we apply the logarithm are : (spreadshhet available in PartB folder: **output_users_fields_what_to_do.xlsx)**

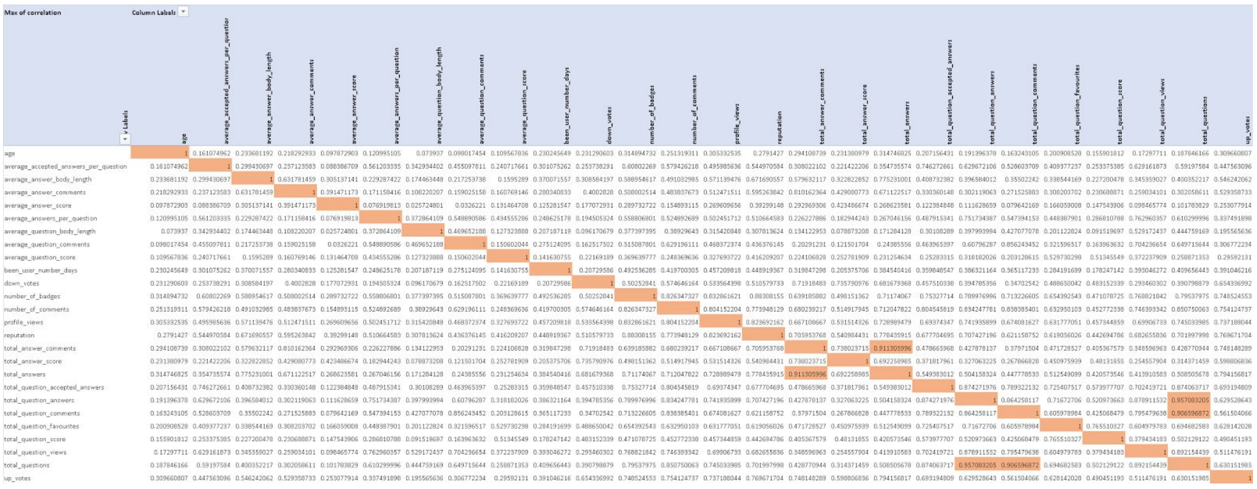
id on raw data	id on my_list_features	variable/feature/field	use?	shall I apply logarithm before z-score?	min	max	mean	variance
1	0	id	FALSE					
2	0	reputation	TRUE	TRUE	1	969,386	115	4,989,499
3	1	signup_date	FALSE					
4	1	last_access_date	FALSE					
	1	been_user_number_days	TRUE	TRUE	0	3,313	404	372,568
5	2	has_website	FALSE					
6	2	has_location	FALSE					
7	2	has_about_me	FALSE					
8	2	profile_views	TRUE	TRUE	0	1,446,764	15	430,415
9	3	up_votes	TRUE	TRUE	0	73,498	12	20,531
10	4	down_votes	TRUE	TRUE	0	50,505	1	7,706
11	5	age	TRUE	FALSE	0	147	3	84
12	6	number_of_badges	TRUE	TRUE	0	15,626	3	242
13	7	number_of_comments	TRUE	TRUE	0	78,265	8	21,682
14	8	total_answer_score	TRUE	TRUE	-67	360,955	8	66,012
15	9	total_answer_views	FALSE		0	0	0	0
16	9	total_answer_comments	TRUE	TRUE	0	97,108	4	11,502
17	10	total_answer_favourites	FALSE					
18	10	total_answers	TRUE	TRUE	0	37,988	3	2,825
19	11	average_answer_score	TRUE	TRUE	-49	3,298	0	28
20	12	average_answer_body_length	TRUE	TRUE	0	52,547	151	199,872
21	13	average_answer_comments	TRUE	TRUE	0	51	0	0
22	14	total_question_score	TRUE	TRUE	-161	27,535	4	2,870
23	15	total_question_views	TRUE	TRUE	0	16,585,817	3,566	2,158,697,206
24	16	total_question_answers	TRUE	TRUE	0	5,757	3	401
25	17	total_question_accepted_answers	TRUE	TRUE	0	1,991	1	54
26	18	total_question_comments	TRUE	TRUE	0	6,099	4	423
27	19	total_question_favourites	TRUE	TRUE	0	11,897	1	390
28	20	total_questions	TRUE	TRUE	0	2,286	2	108
29	21	average_question_score	TRUE	TRUE	-36	6,310	0	30
30	22	average_question_body_length	TRUE	FALSE	0	47,633	540	1,594,133
31	23	average_answers_per_question	TRUE	FALSE	0	158	0	1
32	24	average_accepted_answers_per_question	TRUE	FALSE	0	1	0	0
33	25	average_question_comments	TRUE	TRUE	0	56	1	2

Notice we also remove the fields has_website, has_location, has_about_me, as these were categorical and so they don't make much sense to add to a K-Means algorithm (see however Huang 1998 [5]). We also remove total_answer_views and total_answer_favourites, as they are empty for all fields. So we end up with a total of 26 fields.

We then apply z-score normalization via:

```
for i in arange(0,len(my_list_features),1):
       mean = np.mean(my_list_features[i])
       std = np.std(my_list_features[i])
       my_list_features[i]=(my_list_features[i]-mean)/std
```

We finally calculate the cross-correlation of all variable to determine if any has a strong correlation with another (in case there is a linear combination on one of of features):



Max of correlation	Column Labels												
Row Labels	age	average_accepted_answers_per_question	average_answer_body_length	average_answer_comments	average_answer_score	average_answers_per_question	average_question_body_length	average_question_comments	average_question_score	been_user_number_days	down_votes	number_of_badges	number_of_comments
age	1	0.161074962	0.233681192	0.218292933	0.097872903	0.120995105	0.073937	0.098017454	0.109567836	0.230245649	0.231290603	0.314894732	0.251319311
average_accepted_answers_per_question	0.161074962	1	0.299430697	0.237123583	0.088386709	0.561203335	0.342934402	0.455097811	0.240717661	0.301075262	0.253738291	0.60802269	0.579426218
average_answer_body_length	0.233681192	0.299430697	1	0.631781459	0.305137141	0.229287422	0.174463448	0.217253738	0.1595289	0.370071557	0.308584197	0.588954617	0.491032985
average_answer_comments	0.218292933	0.237123583	0.631781459	1	0.391471173	0.171158416	0.108220207	0.159025158	0.160769146	0.280340833	0.4002828	0.508002514	0.483837673
average_answer_score	0.097872903	0.088386709	0.305137141	0.391471173	1	0.076919813	0.025724801	0.0326221	0.131464708	0.125281547	0.177072931	0.289732722	0.154893115
average_answers_per_question	0.120995105	0.561203335	0.229287422	0.171158416	0.076919813	1	0.372864109	0.548890586	0.434555286	0.248625178	0.194505324	0.558806801	0.524892689
average_question_body_length	0.073937	0.342934402	0.174463448	0.108220207	0.025724801	0.372864109	1	0.469652188	0.127323888	0.207187119	0.096170679	0.377397395	0.38929643
average_question_comments	0.098017454	0.455097811	0.217253738	0.159025158	0.0326221	0.548890586	0.469652188	1	0.150602044	0.275124095	0.162517502	0.515087801	0.629196111
average_question_score	0.109567836	0.240717661	0.1595289	0.160769146	0.131464708	0.434555286	0.127323888	0.150602044	1	0.141630755	0.22169189	0.369639777	0.248369636
been_user_number_days	0.230245649	0.301075262	0.370071557	0.280340833	0.125281547	0.248625178	0.207187119	0.275124095	0.141630755	1	0.20729586	0.492536285	0.419700305
down_votes	0.231290603	0.253738291	0.308584197	0.4002828	0.177072931	0.194505324	0.096170679	0.162517502	0.22169189	0.20729586	1	0.50252841	0.574646164
number_of_badges	0.314894732	0.60802269	0.588954617	0.508002514	0.289732722	0.558806801	0.377397395	0.515087801	0.369639777	0.492536285	0.50252841	1	0.826347327
number_of_comments	0.251319311	0.579426218	0.491032985	0.483837673	0.154893115	0.524892689	0.38929643	0.629196111	0.248369636	0.419700305	0.574646164	0.826347327	1
profile_views	0.305332535	0.495985636	0.571139476	0.512471511	0.269609656	0.502451712	0.315420848	0.468372374	0.327693722	0.457209818	0.533564398	0.832861621	0.804152204
reputation	0.2791427	0.544970584	0.671690557	0.595263842	0.39299148	0.510664583	0.307813624	0.436376145	0.416209207	0.448919367	0.510579733	0.88308155	0.773948129
total_answer_comments	0.294108739	0.308022102	0.579632117	0.810162364	0.292969306	0.226227886	0.134122953	0.20291231	0.224106828	0.319847298	0.71918483	0.639185882	0.680239217
total_answer_score	0.231380979	0.221422206	0.322822852	0.429080773	0.423486674	0.182944243	0.078873208	0.121501704	0.252781909	0.205375706	0.735790976	0.498151362	0.514917945
total_answers	0.314746825	0.354735574	0.775231001	0.671122517	0.268623581	0.267046156	0.171284128	0.24385556	0.231254634	0.384540416	0.681679368	0.71174067	0.712047822
total_question_accepted_answers	0.207156431	0.746272661	0.408732382	0.330360148	0.122384848	0.487915341	0.30108289	0.463965397	0.25283315	0.359848547	0.457510338	0.75327714	0.804545819
total_question_answers	0.191396378	0.629672106	0.396584012	0.302119063	0.111628659	0.751734387	0.397993994	0.60796287	0.318182026	0.386321164	0.394785356	0.789976996	0.834247781
total_question_comments	0.163243105	0.528603709	0.35502242	0.271525883	0.079642169	0.547394153	0.427077078	0.856243452	0.203128615	0.365117233	0.34702542	0.713226605	0.838385401
total_question_favourites	0.200908528	0.409377237	0.338544169	0.308203702	0.166059008	0.448387901	0.201122824	0.321596517	0.529730298	0.284191699	0.488650042	0.654392543	0.632950103
total_question_score	0.155901812	0.253375385	0.227200478	0.230688871	0.147543906	0.286810788	0.091519697	0.163963632	0.51345549	0.178247142	0.483152339	0.471078725	0.452772338
total_question_views	0.17297711	0.629161873	0.345359027	0.259034101	0.098465774	0.762960357	0.529172437	0.704236654	0.372237909	0.393046272	0.293460302	0.768821842	0.746393342
total_questions	0.187846166	0.59197584	0.400352217	0.302058611	0.101783829	0.610299996	0.444759169	0.649715644	0.258871353	0.409656443	0.390798879	0.79537975	0.850750063
up_votes	0.309660807	0.447563096	0.546242062	0.529358733	0.253077914	0.337491898	0.195565636	0.306772234	0.29592131	0.391046216	0.654336992	0.748524553	0.754124737

Row Labels	profile_views	reputation	total_answer_comments	total_answer_score	total_answers	total_question_accepted_answers	total_question_answers	total_question_comments	total_question_favourites	total_question_score	total_question_views	total_questions	up_votes
age	0.305332535	0.2791427	0.294108739	0.231380979	0.314746825	0.207156431	0.191396378	0.163243105	0.200908528	0.155901812	0.17297711	0.187846166	0.309660807
average_accepted_answers_per_question	0.495985636	0.544970584	0.308022102	0.221422206	0.354735574	0.746272661	0.629672106	0.528603709	0.409377237	0.253375385	0.629161873	0.59197584	0.447563096
average_answer_body_length	0.571139476	0.671690557	0.579632117	0.322822852	0.775231001	0.408732382	0.396584012	0.35502242	0.338544169	0.227200478	0.345359027	0.400352217	0.546242062
average_answer_comments	0.512471511	0.595263842	0.810162364	0.429080773	0.671122517	0.330360148	0.302119063	0.271525883	0.308203702	0.230688871	0.259034101	0.302058611	0.529358733
average_answer_score	0.269609656	0.39299148	0.292969306	0.423486674	0.268623581	0.122384848	0.111628659	0.079642169	0.166059008	0.147543906	0.098465774	0.101783829	0.253077914
average_answers_per_question	0.502451712	0.510664583	0.226227886	0.182944243	0.267046156	0.487915341	0.751734387	0.547394153	0.448387901	0.286810788	0.762960357	0.610299996	0.337491898
average_question_body_length	0.315420848	0.307813624	0.134122953	0.078873208	0.171284128	0.30108289	0.397993994	0.427077078	0.201122824	0.091519697	0.529172437	0.444759169	0.195565636
average_question_comments	0.468372374	0.436376145	0.20291231	0.121501704	0.24385556	0.463965397	0.60796287	0.856243452	0.321596517	0.163963632	0.704236654	0.649715644	0.306772234
average_question_score	0.327693722	0.416209207	0.224106828	0.252781909	0.231254634	0.25283315	0.318182026	0.203128615	0.529730298	0.51345549	0.372237909	0.258871353	0.29592131
been_user_number_days	0.457209818	0.448919367	0.319847298	0.205375706	0.384540416	0.359848547	0.386321164	0.365117233	0.284191699	0.178247142	0.393046272	0.409656443	0.391046216
down_votes	0.533564398	0.510579733	0.71918483	0.735790976	0.681679368	0.457510338	0.394785356	0.34702542	0.488650042	0.483152339	0.293460302	0.390798879	0.654336992
number_of_badges	0.832861621	0.88308155	0.639185882	0.498151362	0.71174067	0.75327714	0.789976996	0.713226605	0.654392543	0.471078725	0.768821842	0.79537975	0.748524553
number_of_comments	0.804152204	0.773948129	0.680239217	0.514917945	0.712047822	0.804545819	0.834247781	0.838385401	0.632950103	0.452772338	0.746393342	0.850750063	0.754124737
profile_views	1	0.823692162	0.667108667	0.531514326	0.728989479	0.69374347	0.741935899	0.674081627	0.631777051	0.457344859	0.69906733	0.745033985	0.737188044
reputation	0.823692162	1	0.705953768	0.540984431	0.778435915	0.677704695	0.707427196	0.621158752	0.619056026	0.442694786	0.682655836	0.701997998	0.769671704
total_answer_comments	0.667108667	0.705953768	1	0.738023715	0.911305996	0.478665968	0.427878137	0.37971504	0.471728527	0.405367579	0.348596963	0.428770944	0.748148289
total_answer_score	0.531514326	0.540984431	0.738023715	1	0.692258985	0.371817961	0.327063225	0.267866828	0.450975939	0.48131855	0.254557904	0.314371459	0.598806836
total_answers	0.728989479	0.778435915	0.911305996	0.692258985	1	0.549383012	0.504158324	0.447778533	0.512549099	0.420573546	0.413910583	0.508505678	0.794156817
total_question_accepted_answers	0.69374347	0.677704695	0.478665968	0.371817961	0.549383012	1	0.874271976	0.789322132	0.725407517	0.573977707	0.702419721	0.874063717	0.693194809
total_question_answers	0.741935899	0.707427196	0.427878137	0.327063225	0.504158324	0.874271976	1	0.864258117	0.71672706	0.520973663	0.878911532	0.957083205	0.629528643
total_question_comments	0.674081627	0.621158752	0.37971504	0.267866828	0.447778533	0.789322132	0.864258117	1	0.605978984	0.425068479	0.795479638	0.906596872	0.561504066
total_question_favourites	0.631777051	0.619056026	0.471728527	0.450975939	0.512549099	0.725407517	0.71672706	0.605978984	1	0.765510327	0.604979783	0.694682583	0.628142028
total_question_score	0.457344859	0.442694786	0.405367579	0.48131855	0.420573546	0.573977707	0.520973663	0.425068479	0.765510327	1	0.379434183	0.502129122	0.490451193
total_question_views	0.69906733	0.682655836	0.348596963	0.254557904	0.413910583	0.702419721	0.878911532	0.795479638	0.604979783	0.379434183	1	0.892154439	0.511476191
total_questions	0.745033985	0.701997998	0.428770944	0.314371459	0.508505678	0.874063717	0.957083205	0.906596872	0.694682583	0.502129122	0.892154439	1	0.630151985
up_votes	0.737188044	0.769671704	0.748148289	0.598806836	0.794156817	0.693194809	0.629528643	0.561504066	0.628142028	0.490451193	0.511476191	0.630151985	1

So from these we remove total_answer_comments, total_question_answers and total_question_comments as their cross correlation with other variables are > 0.9. After removing that, we end up with a total of 23 fields to pass to the mllib and our own pyspark implementation of k-means code.

The file processing the data from part A and outputting the clean data for part C is saved in HDFS under `hdfs:/user/group-AI/output_user_clean.csv`.

The code created to perform this analysis is the python file get_all_features_histograms.py

Part C - Clustering pyspark.

This section outlines our k-means implementation in PySpark.

k-Means is an iterative algorithm that finds clusters in data in an unsupervised way (see [6-9]).
The way it is designed, k-means is able to find clusters that minimize the overall distance of the points to the assigned cluster centroid (cost).

The algorithm can be structured in 3 main parts:

1- Initialization of centroids
2-Assignment of each point to clusters
3-Update cluster centroids.

Parts 2 and 3 are run until the convergence is achieved, i,e situation where the cost doesn't change more than a defined value of epsilon from the previous iteration to the next.

Implementation of K-means

This algorithm was implemented in Pyspark and the code can be found in the file: *Kmeans_ForStackFile.py*.

The program receives as input:

- A csv file to be clustered.
- The number of clusters we want to find (k)
- The epsilon value
- The portion of the file that it will use (in order to realize tests).

The code starts by reading and parsing the csv file from *HDFS* to a collection RDD`_part_parsed`. To allow fast testing it is possible to use only a portion of the file.
Since the file to use has a heading, the easiest way to read it is by creating a dataframe from the csv, this dataframe is then transformed into an Rdd that we parse using the defined function:
`parsePoint_fromStringArray(st)`.
This RDD`_part_parsed` is cached and the number of points are counted using the pyspark method `count()`

This Implementation follows the same structure mentioned above, after the processing of the csv file the actual k-means algorithm can be applied to the data.

1. Initialization of centroids

For the Initialization of the centroids we opted for a simple initialization strategy that randomly picks k points from the initial file:

```
iniCentroids = rdd_part_parsed.takeSample(False, k, rseed)
```

Keeping the seed will allow us to make a comparison with the MLlib algorithm using the same initial centroids.

As the UserId is not included, and some points have the same values, there is a chance that this method will pick identical points. To guarantee that we don’t have repeated centroids we check the uniqueness of the initial centroids and we keep taking new samples until all the centroids are different from each other.

```
mset = [list(x) for x in set(tuple(x) for x in iniCentroids)]
while (len(iniCentroids) is not len(mset)):
    print("will take another sample1")
    # take new samples until length are the same!
    iniCentroids = rdd_part_parsed.takeSample(False,k, 0)
    mset = [list(x) for x in set(tuple(x) for x in iniCentroids)]
```

We create a local variable that will contain the centroids:

```
centroids_table=[]  #Create dictionary of centroids from inicentroids
for c in range(k):
      centroids_table.append((c, iniCentroids[c]))
```

As we will need the indexes of the centroids for the algorithm, we create a dictionary based on the previous table:

```
dict_centroids=create_dict(centroids_table)
```

The create_dict function is implemented as the following:

```
def create_dict(table):
      dict_table={}
      for t in table:
            dict_table[t[0]]=t[-1]
      return dict_table
```

Having the initial centroids and the parsed file the iterative process can start, this is done by defining a main cycle that evaluates the difference in cost from the previous iteration to the next.

```
while (delta_cost>epsilon)
```
Both assignment of points to clusters and centroid updates happen inside this cycle.

2. <u>Assignment of each point to clusters</u>

To assign a point to a cluster, we need to compute the distances to all the centroïds and pick the smallest one:

```
rdd_centroids =
rdd_part_parsed.map(lambda row :
(row,calculate_closest_centroid(row,dict_centroids, k))).cache()
```

We create a new RDD file containing the previous points and a new column containing a tuple of the closest centroid index for a given point and the corresponding distance. The following function is used in the process:

```
def calculate_closest_centroid(row, dict_centroids, k):
        dist = 100000000000
        index = k+1
        for i in range(k):
        aa=array(row)
        bb = array(dict_centroids[i])
        new_distance=linalg.norm(aa-bb)**2
        if new_distance < dist:
              dist = new_distance
              index = i
        return (index, dist)
```

In fact, we return the squared distances that will be used in the computation of the cost function. We square the output of the linalg.norm method that has been imported in the numpy package. We loop over all the centroids updating the distance and the index each time we find the closest centroid.

The RDD is cached to be kept in memory for the next operations.

3. <u>Update cluster centroids</u>

Now that all the points have been assigned to a centroid, we need to compute the new centroids averaging all the points with the same centroid index.

We create a new RDD with first column having the index of the centroid and the second having the coordinates of the point. A ReduceByKey will allow us to count the number of points per cluster that will be stored in the RDD rdd_count:

```
rdd_temp = rdd_centroids.map(lambda row : (int(row[-1][0]),
row[0])).cache()
rdd_count = rdd_temp.map(lambda row : (row[0],1)).reduceByKey(lambda a,b :
(a+b))
```

We sum the points contained in rdd_temp (note that the RDD has been kept in memory) using another ReduceByKey:

```
rdd_sum_computation = rdd_temp.reduceByKey(lambda a,b :[x + y for x, y in
zip(a, b)])
```

To average, we still need to divide each centroid by the number of points of the corresponding cluster. This is done with the pyspark Join function:

```
rdd_index_sum_counts = rdd_sum_computation.join(rdd_count)
```

The Join merges the RDDs containing the sum and the number of points using as a key the IDs of the centroids.

Now that we have an RDD containing all the information needed, we can easily calculate the mean of each cluster:

```
rdd_centroids_update = rdd_index_sum_counts.map(lambda row :
(row[0],compute_newCentroids(row[-1][0], row[-1][-1])))
```

We call a function named compute_newCentroids that will take as input the sum and the number of points of each cluster and return the averaged points dividing each coordinate:

```
def compute_newCentroids(sum_coord, count):
	c=[]
	for x in sum_coord:
		t=x/count
		c.append(t)
	return c
```

Finally, we update the centroids table and the dictionary collecting the results of rdd_centroids_update:

```
centroids=rdd_centroids_update.collect()
dict_centroids=create_dict(centroids)
```

Computation of the Cost Function

The criterion of convergence is set regarding the evolution of the cost function. While the cost does not decrease more than a certain user_defined value (epsilon) from a step to another, the updating of the centroids continues.

We initialize the costs and the delta_cost:

```
delta_cost=10000.0
list_cost=[]
cost=100000
```

At each iteration, we store the previous cost in cost_prev and we calculate the new cost summing over all the squared distances calculated in rdd_centroids. Note that we first add the distances for each cluster using a ReduceByKey on the centroid index:

```
rdd_dist= rdd_centroids.map(lambda row :
(int(row[-1][0]),float(row[-1][1]))).reduceByKey(lambda a,b: (a+b))
```

Then, we collect the results and add together the costs associated to each centroïds:

```
dist = rdd_dist.collect();
cost=sum(x[1] for x in dist)/count_points  #cost=dist/count_points
```

We append the calculated cost the list and compute the delta_cost to check if the value exceeds the user_defined threshold epsilon. If not, we stop the algorithm.

```
list_cost.append(cost)
delta_cost=abs(cost_prev-cost)
```

Outputs:

This code creates 3 local files with initial centroids coordinates, the cost at each iteration and the coordinates of the final centroids found.
Optionally it can also output all the points and assigned cluster in HDFS, but since this operation has shown to be expensive this is only performed in the final experiments.

Spark particularities:

To implement k-Means in spark some aspects regarding memory and particular operations had to be taken into consideration.

- sortByKey():

A previous version of this code was using SortByKey() operations in RDDs concerning the k clusters as a way ro be able to use the index as the cluster id. However, when the first experiments were ran, the sortByKey operations appeared to be taking a long time.
To overcome this dictionaries are used instead which allow for a direct access of the value associated to each cluster from the cluster id.

- Collect():

The use of collects was carefully analysed and this operation is only used in small RDDs, in this case the larger Rdd we are collecting has 10 rows and 26 columns.

- Cache():

This operation was used whenever the RDD was going to be used more than once.
The most significant cache() is done on the Rdd_part_file since it is used in every iteration to compute the distances and the RDD never changes. This allowed a significant improvement of the code execution time.

Experiments:

The code is run by calling the following command:

```
spark-submit --master yarn --packages com.databricks:spark-csv_2.10:1.4.0
Kmeans_ForStackFile.py --k 2 --epsilon 0.001 --maxiter 25 --fileportion
1.0
```

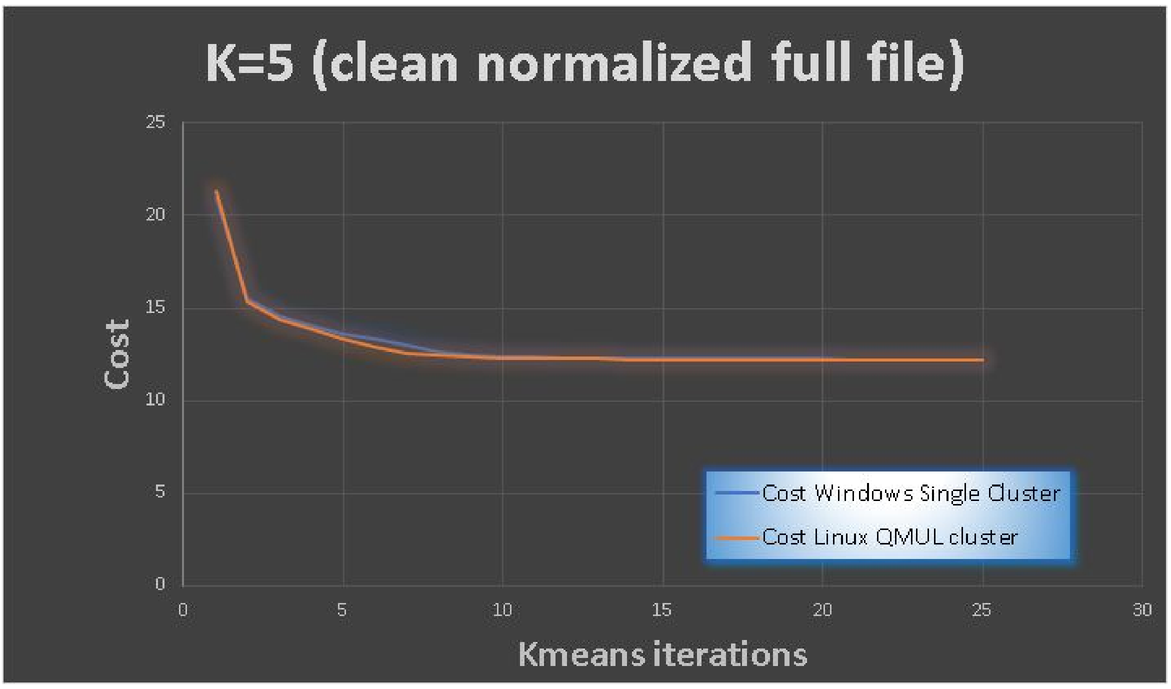
This command should have defined as arguments:
Number of cluster (k), threshold for stopping criteria (epsilon), maximum number of iterations we allow the program to run (maxiter) and finally the portion of the file we are going to use.

The k-means implementation explained before uses as data to cluster the file that contains 7.5 million lines of user features (in **/user/group-AI/output_user_clean.csv)**

The first experiments with this code were made using increasingly bigger portions of the file. These helped us to verify the convergence of the algorithm, have an idea about the time it would take to cluster the whole file and also,it allowed us to adjust the values of epsilon.
In parallel the initial experiments were ran on a Windows single cluster installation of Spark in order to gain further understanding of the possibilities of the code and to compare performance between windows pyspark and linux pyspark.

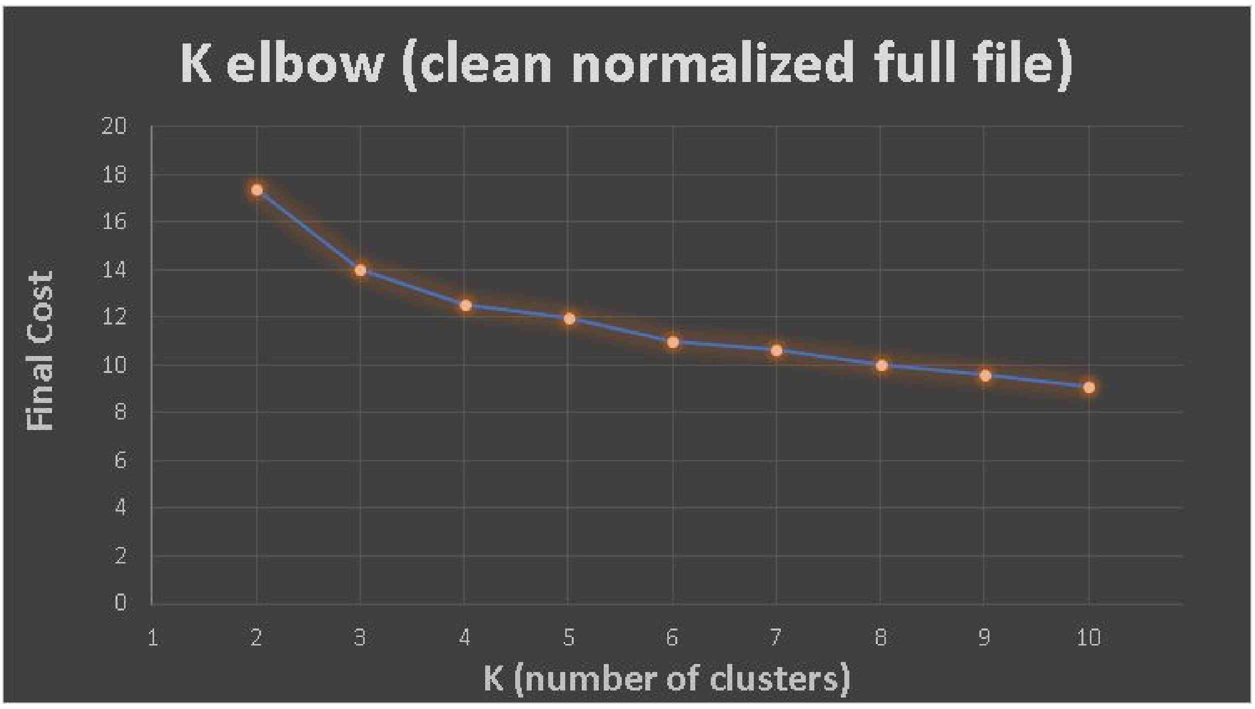
The decision regarding epsilon is a balance between number of iterations the algorithm will run versus the amount of error we are willing to take. The final value for epsilon was set to 0.001 which is small enough to guarantee that the cost converged but not so small that it will run a huge number of iterations.
Having verified the convergence, the maximum number of iterations is set to a higher value than the iterations the algorithm needs to converge, and can indeed be ignored.

The convergence can be shown by plotting the cost value at each iteration of the algorithm, below is an example for k=5:

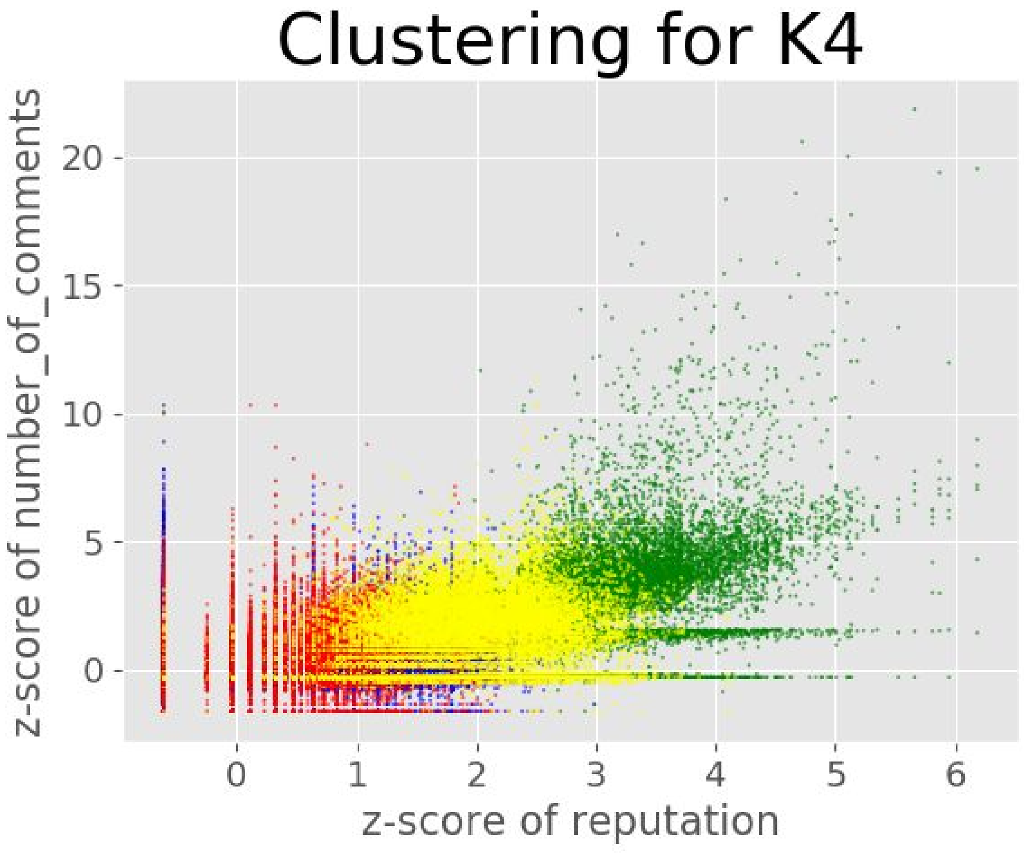


As an unsupervised learning problem, we don’t know exactly what is the number of clusters that best apply to the data. Here we follow the elbow method (see[10]) that says that the best number of clusters is the highest number until adding another cluster does not yield evident better results.
In practice this can be seen by plotting the cost over different values of k and choose the k where the line’s slope starts to decrease.

To do this the k-means implementation was applied to the whole dataset with different number of clusters (k) from k=2 to k=10.
The resulting plot of final cost over K is presented below:



From here we concluded that the best number of clusters to cluster stackoverflow users is 4.
To have a visualization of the 4 clusters obtained, (since we cannot plot clusters in high dimensionality), different combinations of 2 features were chosen. Then 1% of the clustered user points using only the selected 2 features were plotted. The image below is the best visualization of the 4 clusters in 2 dimensions:



Part D - Benchmark and verify against MLlib.

To evaluate our own clustering algorithm, we test against a known implementation: PySpark's MLlib [1]. Any randomisation must be pseudo random seeded to ensure repeatability across algorithms. Python's random package uses the Mersenne Twister [2] generator, which we can force pseudorandom by seeding. Specifically, we ensure that any calls to sample() or takeSample() performed on an RDD type will be seeded by the third parameter:

```
randomseed=0
#...
iniCentroids = rdd_parsed.takeSample(False, k, randomseed)
```

This gives full control over the random initialisation and we are free to change the seed at any time. Since the output of K-means depends on the choice of initial centroids, this ensures that both MLlib and our implementation start with the same initial centroids.

<u>Running K-means with MLlib.</u>

Apache Spark's MLlib provides K-means clustering routines under pyspark.mllib.clustering. The two classes we will use are KMeans and KMeansModel which we can import for use as follows:

```
from pyspark.mllib.clustering import KMeans, KMeansModel
```

For our purposes, performing K-means clustering with MLlib requires that we specify the initial centroids up front. This bypasses MLlib's initialisation randomisations (which may or may not match our initial centroid choices). We do this by defining a model from the initial centroids we have created in our own implementation.

```
# iniCentroids created in our own algorithm above
init_model = KMeansModel(iniCentroids)
```

From here we run K-means to train a model using the same parameters as our own algorithm.

```
# rdd_part_parsed is our parsed input dataset
model = ml_lib.KMeans.train(rdd_part_parsed,
                            k,
                            maxIterations = maxiter,
                            epsilon,
                            initialModel = init_model)
```

<u>Stopping criteria and cost differences</u>.

Note that MLlib works slightly different to our code. Stopping criteria for MLlib is a threshold epsilon based on centroid movement. Stopping criteria for our own implementation is a threshold epsilon based on change in cost between iterations. MLlib's is less computationally expensive since it computes distances for the k centroids. For our purposes, it is more instructional to examine the cost on each iteration. This way we can verify correctness of our algorithm and have more information with which to choose an optimal number of clusters k. Further work could examine the potential performance improvements with a centroid-movement based stopping criteria, similar to

MLlib.

Since MLlib does not compute the cost as a byproduct of the K-means algorithm, after training we must call

```
final_cost = model.computeCost(rdd_part_parsed)
```

MLlib's cost is not averaged over the total number of data points. To compare against our own cost we must divide by total points count_points

```
avg_cost = final_cost / count_points
```

Verification against our algorithm.

Initial runs of MLlib and our own code yield very similar results. As we would expect since we are starting from the same initial centroids. For example, first two features for $k = 3$ final clusters are

```
k1mllib = [-0.43486074, -0.2635395 , ...
k2mllib = [ 0.70858851,  0.51822871, ...
k3mllib = [ 2.80652374,  1.24607485, ...

k1ours = [-0.43455965389347995, -0.2637272176510595, ...
k2ours = [0.6853274013811156, 0.5060869236648843, ...
k3ours = [2.7765524103583608, 1.2408634997943437, ...
```

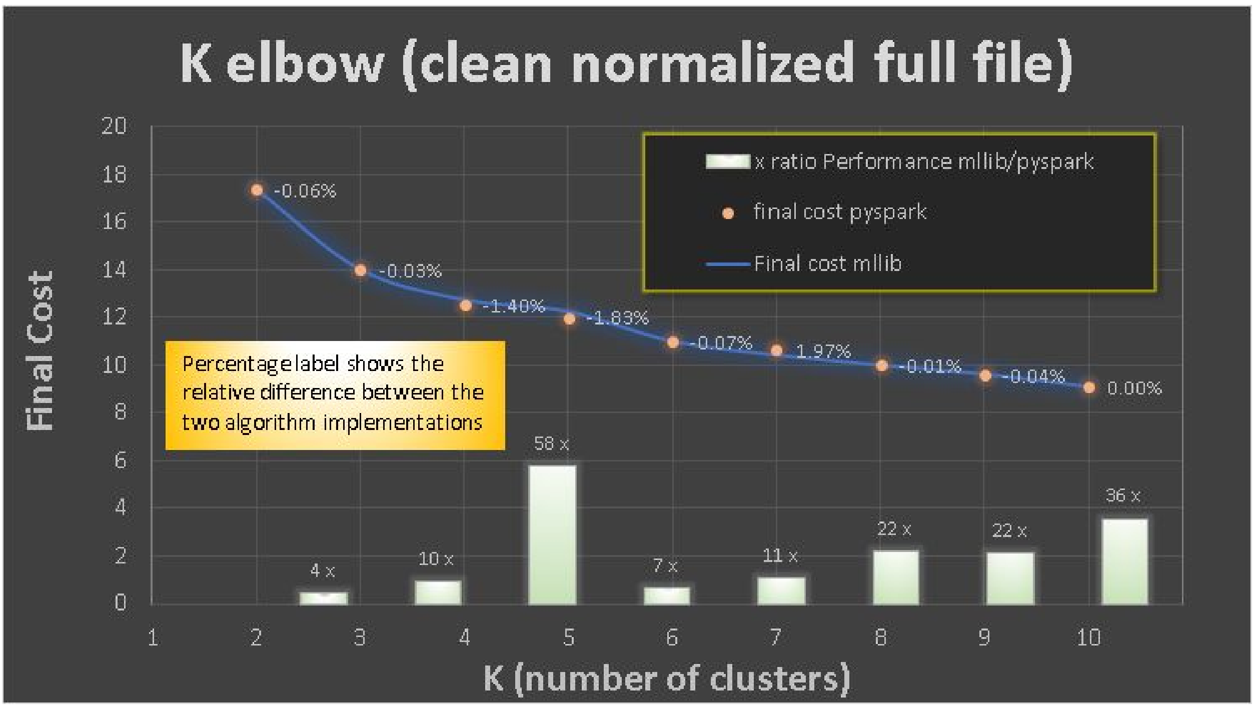
Computing the distances between the final clusters using norm() we find that we are very close:

```
# distance between our final clusters and MLlib's
k1dist =  0.012444
k2dist =  0.077655
k3dist =  0.21954
```

Since we are using a different stopping criteria (cost-delta based rather than centroid-movement based) we can expect slight differences. This is most evident for the third centroid, where the distance between implementations is 0.2. If we implement a centroid-movement based stopping criteria in further work, we should expect to get exactly the same final centroids. However, given that we are working with a 26 feature vector the distance values are very close, and we are satisfied to proceed with our algorithm.

Benchmarking MLlib against our K-means.

The figure below shows the comparison of MLlib K-means against our own algorithm for $k = 1, 2, \ldots, 10$.



Comparing MLlib K-means to our own algorithm for various numbers of clusters k

By plotting final cost against k we can use the elbow technique to choose an optimal value of . k At $k = 4$ there is a pronounced bend or "elbow". For larger k the reductions in cost are no longer as large as the jump in cost from $k = 3$ to , $k = 4$ suggesting that there are diminishing returns in choosing a value $k > 4$. (The choice of k by elbow is discussed in more detail in the previous section.)

By plotting both our own final cost (final cost pyspark) and MLlib's (final cost mllib), we have a single metric that shows we are coming up with extremely similar centroids. On average, the difference in costs between implementations is less than 1.7%.

We are also interested in computational performance. This is a little bit harder to quantify since real world timings on the cluster depend heavily on many factors (e.g. number of users & jobs running on the cluster at the time, etc.) With this caveat in mind, we can draw some basic conclusions. The white bars show the "x ratio performance" meaning for example for $k = 2$ our implementation was 10 times slower than MLlib and for $k = 3$ our implementation was 8 times slower than MLlib.

For our chosen $k = 4$, being 10 times slower than MLlib is good enough for our purposes since the overall job time is only 15 minutes approximately. Recall that we are also calculating the cost per iteration. This way we can examine and track the change in cost as the algorithm progresses. Since calculating cost is an expensive operation, we are satisfied to be within an order of magnitude away from MLlib in terms of computational performance.

Part E - Conclusion

We analysed a large set of data from stackoverflow. We first extracted what we believed were useful features for clustering, per user, e.g., age, votes, reputation etc. The features were extracted using pyspark code. We then analysed the feature set, and identified that some variables (most in fact) follow a power law, and therefore we applied the natural logarithm before applying a standard z-score scaling. We also removed one of each pair of obviously collinear variables that were identified using the correlation matrix.

Once we had clean data, we then ran a second set of pyspark code to apply the k-means algorithm. This was run with the entire data set (around 7.5 million users) for k=2 to k=10. We obtained a well behaved cost (per iteration) for all k's and concluded with an elbow chart which showed the final cost(k). The results seem to indicate the presence of at least one elbow (if not two). We selected k=4 as the most conclusive elbow value, i.e., we believe that users could be grouped naturally into 4 clusters.

We then analysed the performance and results of our code against the pyspark mllib library. The results of final cost(k) show a very good match, which increased confidence (at least numerically) in the implementation of our code. We also compared the performance of mllib versus our code and noted that the former was around 10x faster than our pyspark code, which is not surprising given that mllib is a highly vectorized open source library, and that we were calculating full cost per iteration.

We concluded by showing one of many possible projections of the high dimensional clustering into a two dimensional subspace. It seems to indicate, that despite the high dimensional aspect of the problem, clustering may indeed work for K=3 or K=4, with a somehow clear visual division of the clustering colours on some selected two dimensional subspaces.

References

[1] X. Meng *et al.*, “Mllib: Machine learning in apache spark,” *The Journal of Machine Learning Research*, vol. 17, no. 1, pp. 1235–1241, 2016.

[2] M. Matsumoto and T. Nishimura, “Mersenne twister: A 623-dimensionally equidistributed uniform pseudorandom number generator,” *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 8, no. 1, pp. 3–30, 1998.

[3] E. Kampf , “Best Practices Writing Production-Grade PySpark Jobs“, https://developerzen.com/best-practices-writing-production-grade-pyspark-jobs-cb688ac4d20f, 2017

[4] https://en.wikipedia.org/wiki/Standard_score

[5] Huang, Z, “Extensions to the k-Means Algorithm for Clustering Large Datasets with Categorical Values”, *Data Mining and Knowledge Discovery* vol. 2, pp. 283–304, 1998.

[6] MacQueen, J. B., "Some Methods for classification and Analysis of Multivariate Observations", Proceedings of 5th Berkeley Symposium on Mathematical Statistics and Probability. University of California Press. pp. 281–297, 1967.

[7] Lloyd, S. P., "Least square quantization in PCM". Bell Telephone Laboratories Paper, 1957.

[8] Lloyd, S. P., "Least squares quantization in PCM", IEEE Transactions on Information Theory. 28 (2): pp .129–137, 1982.

[9] Forgy, E.W., "Cluster analysis of multivariate data: efficiency versus interpretability of classifications". Biometrics. 21: 768–769, 1965.

[10] https://en.wikipedia.org/wiki/Elbow_method_(clustering)