Fun Times With Hadoop and Distributed K-Means

Big Data Final Project

Nera Nesic

1. Introduction

Hadoop is a distributed computing framework, based on the MapReduce compute model, which comes with its own implementation of a distributed filesystem (HDFS). Hadoop is becoming increasingly popular among the tech companies because it is an open-source implementation of MapReduce, which is very useful for executing many simple computations on a large dataset in parallel. This makes knowledge of Hadoop a useful skill for anyone working in the tech industry to have. Moreover,I've had an interest in parallel computation for some time, which made me decide to use this project as an opportunity to familiarize myself with the framework. I was interested in looking at it from a technical and practical side. First of all I wanted to learn how to set up a cluster - learning how to set it up is a great way to understand how it it is structured and how it uses configuration files. Next, I wanted to become comfortable operating the HDFS, and learn some basics of managing my cluster, such as: how are new nodes added to the cluster; how can storage capacity be extended; or how is the storage load balanced. From HDFS, I moved to MapReduce, and decided to follow CCP's example and explore how to write jobs in Python because coding in Python is a zen experience.

I have chosen a dump of Stack Exchange forums as the dataset to apply MapReduce processing on. It is a moderately sized dataset - about 65 GB uncompressed - which means it is non-trivially sized to process with MapReduce, while at the same time being manageable by my modest cluster. I focused the MapReduce jobs on extracting and aggregating user information from the dataset. I was interested in counting number and size of users' posts and comments, as well as their scoring, age, and reputation. I used this data to analyze some aspects and characteristics of user behavior.

Second part of my project was inspired by the various papers we have read about affronting challenges of distributed computation, and I wanted to try my hand at it by creating a distributed K-Means algorithm to cluster Stack Exchange users into some categories based on their behavior, much like CCP has done. The algorithm is implemented independently from Hadoop, since MapReduce does not do a good job with iterative algorithms. It runs on Tornado servers, and consists of master and worker nodes which communicate over HTTP via simple REST interface.

2. Data Processing With Hadoop

This section gives an overview of experience and lessons learned while working with Hadoop. It starts with a short overview of Hadoop architecture in Section 2.1, and proceeds to describe the process of setting up and maintaining the cluster in Section 2.2. Section 2.3 describes the Stack Exchange dataset and methods used to extract, aggregate and store relevant user data from it. Finally, Section 2.4 briefly talks about working with MapReduce, its streaming utility, and jobs written to process the dataset.

2.1. Hadoop Framework Overview

As mentioned above, Hadoop framework consists of two main components: HDFS and MapReduce engine. Both are implemented in Java, making Hadoop highly portable. HDFS is envisioned as a file system capable of handling large files and storing them distributedly across multiple machines. It does so by splitting the file into 64 MB blocks which are stored independently. The Name Node keeps track of which block belongs to which file and where they are stored physically, thus abstracting the distribution away from the user. The Name Node also ensures that each block is replicated, and that at least one replica goes to a different rack. As shown in Fig. 1, Name Node doesn't interact with the physical storage itself, instead using Data Nodes to perform reads and writes.

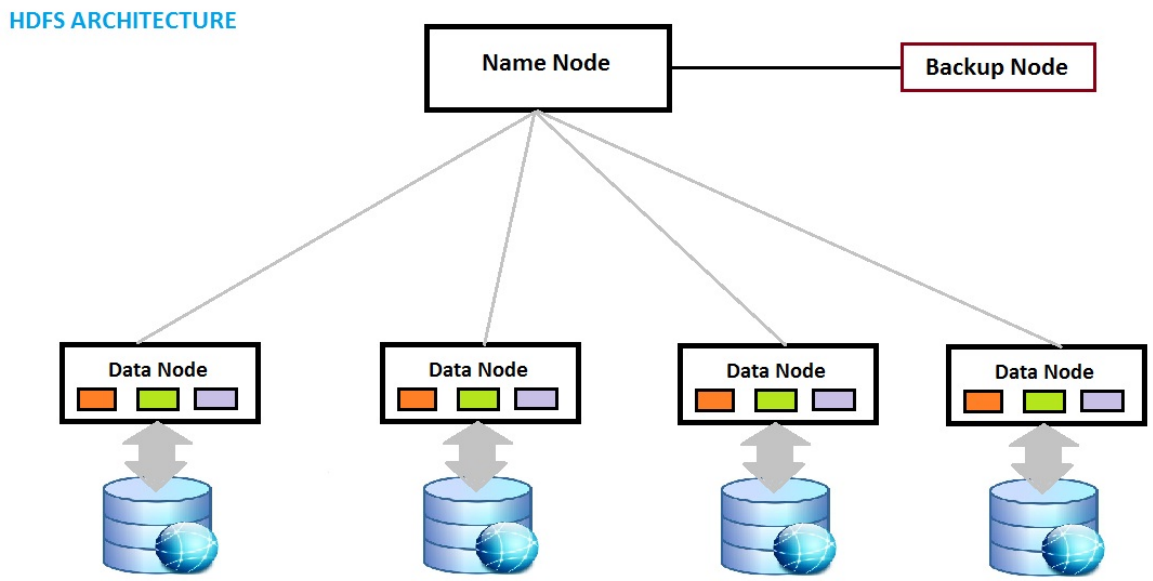

Fig. 1: HDFS architecture

The MapReduce engine (Fig. 2) similarly divides responsibilities between Master Node and Slave Nodes. All jobs are submitted to the Job Tracker on the Master Node in form of jar files, which then schedules the execution of map and reduce jobs and communicates them to Task Trackers on Slave Nodes. The Task Trackers execute the task, and communicate the outputs back to the Job Tracker.

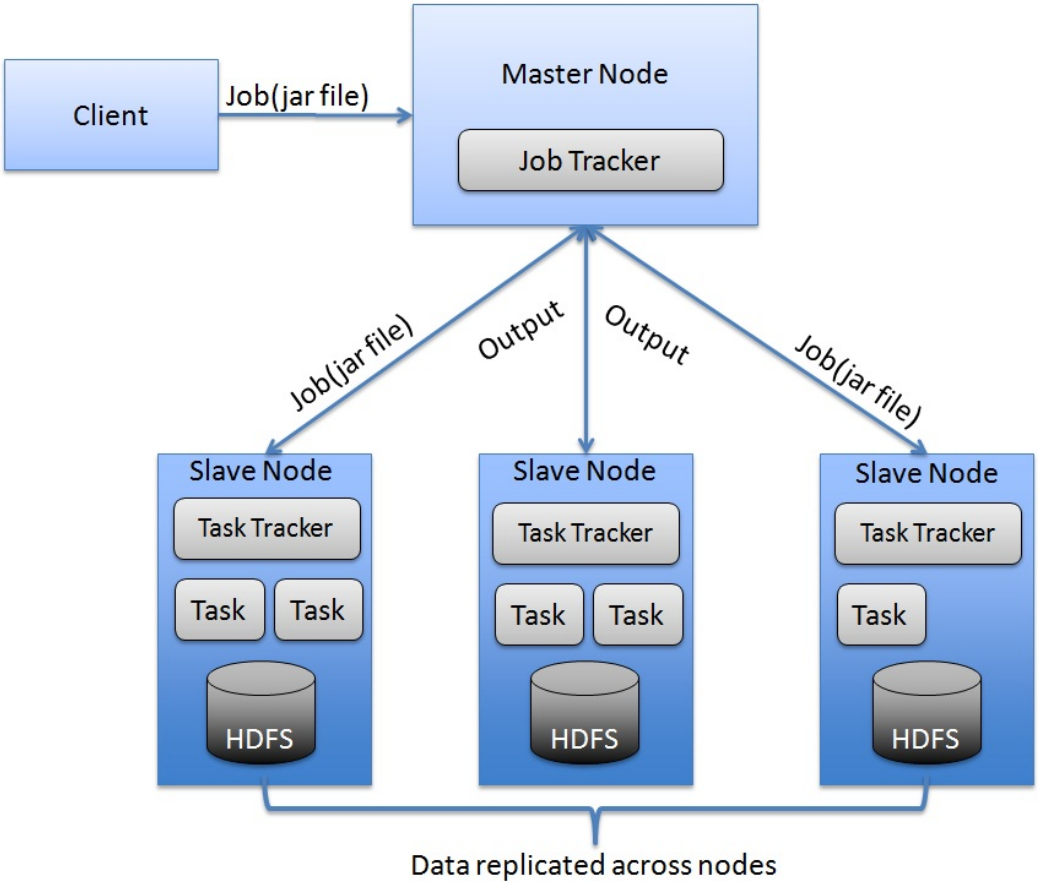

Fig. 2: MapReduce engine architecture

2.2 Handling Hadoop

My starting setup consisted of two nodes, a master and a slave, each with one CPU, 1 GB of RAM, and 40 GB of disk space. With help from an excellent blogpost[1], I have set up Hadoop 2.6.0 on both. The setup process is almost identical on both master and slave nodes, and relatively painless. It is advised to create a user with sudo privileges for hadoop on each node. Since communication between nodes goes through ssh, for each node must exchange its ssh keys with all other nodes. Beside that, it is necessary to add some configurations to let Hadoop know where its nodes are. There are five important configuration files:

- **core-site.xml:** Tells Hadoop where the Name Node is running
- **hdfs-site.xml**: Contains the configuration settings for HDFS daemons, such as replication level and directories where Data Nodes can store blocks
- **mapred-site.xml:** Specifies which MapReduce implementation should be used
- **yarn-site.xml:** Contains locations and ports of various Job Tracker services
- **slaves:** A list of addresses of all the slaves

Once Hadoop is set up, HDFS is simple to use, as it emulates the standard Unix file operation utilities. For example, we can create a new directory with:

```
hdfs dfs -mkdir /newDir
```

Next thing I wanted to find out was how to expand my cluster. To add a new node, it is sufficient to add it in the list of slaves. Similarly, to add a disk, I added an entry to the hdfs-site.xml file specifying the directory in which the disk was mounted. Both operations require restarting HDFS.

I was a bit disappointed to discover that the cluster is not automatically balanced; all the files are initially stored on the Master node. It is possible to balance the cluster by running the balancer utility - balancing is done by comparing the percentage of storage usage of the whole cluster to the percentage of usage of the individual nodes. If the usage of a node differs by more than a specified threshold from the collective usage, blocks are transferred to (or from) the node. Balancing is a costly process, though - it took nearly four hours to balance my cluster with 66 GB of data on it.

2.3 Processing the Stack Exchange Dataset

Stack Exchange is a network of question and answer forums, each one restricted to a specific field of interest. A user can post a question on the forum (referred to as post of type 1), and other users can provide answers in reply posts (type 2). Alternatively, if a user has only a short remark to make about the original question or one of the reply posts, they can write a comment to the post of interest. Users can also upvote or downvote posts and comments to express how helpful and correct they found them. Continued activity on the forums increases user's reputation, which is earned by writing comments and posts, and influenced by scores the user receives for the material they post.

The dump of Stack Exchange consists of a series of xml files, grouped by fields of interest, holding all publicly available data about posts and users. My main goal with this dataset is to learn

something about user behavior. More specifically: How does the age distribution vary in different fields? Is there a correlation between age and reputation? Can the users be clustered into separate groups, based on their usage habits?
To answer these questions, I first needed to extract and aggregate information from the xml files. Table 1 shows which attributes I have chosen to work with, and how they were extracted.

Attribute	Explaination	How it was obtained
user_id	Id of the user, as assigned by Stack Exchange	Specified in Users.xml
number_posts_type_1	Total number of original posts by user	MapReduce to count all posts made by the user in Posts.xml
number_posts_type_2	Total number of reply posts made by user	MapReduce to count all posts made by the user in Posts.xml
number_comments	Total number of comments user made	Specified in Comments.xml
avg_post_score	Average score of user's type 1 posts	MapReduce to aggregate and average scores of all posts made by the user in Posts.xml
avg_comment_score	Average score of user's comments	MapReduce to aggregate and average scores of all comments made by the user in Comments.xml
avg_post_size	Average size of user's posts	MapReduce to aggregate and average sizes of all posts made by the user in Posts.xml
avg_comment_size	Average size of user's comments	MapReduce to aggregate and average sizes of all comments made by the user in Comments.xml
tag_count	Count of post made by user tagged with a specific tag	MapReduce to aggregate and average sizes of all posts made by the user in Posts.xml
upvotes	Total number of upvotes user received	Specified in Users.xml
downvotes	Total number of upvotes user received	Specified in Users.xml
reputation	User's reputation	Specified in Users.xml
age	Age of the user	Specified in Users.xml

Table 1: User attributes extracted from the forums dump

I used mapReduce jobs to aggregate all the data on users from the xml files. Each job outputted its result into a text file, which was then picked up by the *generate_sql* python script and persisted into a local MySQL database. At this point, I had all the user data ready for analysis, which was carried out with python scripts, as will be discussed in Section 3.

2.3 Writing MapReduce Jobs

To run the MapReduce jobs, Hadoop expects a jar file containing a mapper and a reducer, each extending their base implementation java class. We can then run the job with:

hadoop jar <jarname> [jar arguments]

Fortunately, however, Hadoop comes with a utility that allows us to write MapReduce jobs in any programming language runnable on the slave nodes, which allows us to bypass writing tedious java code. The utility itself is a MapReduce jar which takes as arguments the mapper and reducer executables. It works by streaming the files assigned to mappers via stdin. Once mappers are done with processing, they output their results to stdout, which the streamer pipes to the reducers. It needs to be noted that MapReduce assumes that files can be split arbitrarily without affecting the processing, and it does so by default to distribute load as evenly as possible. Xml files, however, cannot be split arbitrarily; instead, I had to disable the file splitting in Hadoop and manually split the xml files into smaller ones that still had all their tags correctly matched.

I wrote a mapper and a reducer for Posts.xml and Comments.xml files. They are quite similar in implementation, so I will only discuss the implementation of Posts processors here. The mapper maintains a dictionary mapping user_id to User object, which holds aggregated data on number of posts of either type, their scores, sizes, and tags. The mapper then iterates through all the posts in its input, extracting the relevant information. It uses the user_id of the owner of the post to find the user in its dictionary, and updates the aggregated values of the user. Once all the posts have been processed, it prints a line of space separated values for each user in the dictionary. For example, a line output might be:

458509 1 0 2 1334 javascript 1 angularjs 1

This output from various mappers is passed to the reducer, which again maintains a mapping of user_id to User objects in which all the values belonging to the same user are aggregated. When all the processing is finished, reducer prints out the aggregated values in the same fashion as the mapper. These are converted into averages by the *generate_sql* script before storage in the database.

3. Distributed K-Means

CCP was able to get some lovely insight into what kind of players they had by running K-means algorithm on their player's ingame behavior data. I wanted to see if I could something similar with the Stack Exchange users. Moreover, K-Means is an algorithm which can benefit greatly from parallelization, so I implemented the distributed version of it. Section 3.1 explains the distributed model for the algorithm, while Section 3.2 goes into implementation specifics. Section 3.3 talks about methods chosen to analyse the quality of the clustering. Finally, section 3.4 shows the performance of the distributed algorithm compared to a single node implementation. Directions on installation and use are included in Appendix A.

3.1. Distributed K-Means Model

K-means algorithms start with a series of centroids. Every data point is assigned to the nearest centroid according to some distance metric (in my case, and generally in most cases, that is Euclidean distance). Then, each centroid is re-calculated to be the center of mass of the points assigned to it. That is, for each attribute *i*, the centroid's new value for that attribute is:

$$\frac{1}{k}\sum_{d=0}^{k} d_i$$

where k is the number of points assigned to that centroid. During the different iterations of the algorithm, there is no need to make any update on data points that would depend on anything except the centroids. Thus, the algorithm can easily be distributed on multiple nodes by sending each node a subset of data points only, and having the node return the contribution of it's points to the new values of a centroid's attributes. Say, for example, we have 2 nodes, which respectively assign *k* and *m* points to a centroid. If each node returns the sum of values of attribute *i* over all data points assigned to the centroid, as well as the number of data points assigned to that centroid at that node, we can reconstruct the new value for that attribute at the centroid as :

$$\frac{1}{k+m}\sum_{d=0}^{k} d_i + \frac{1}{k+m}\sum_{d=0}^{m} d_i$$

This allows us to construct a distributed model consisting of one master node and multiple workers, in which a portion of data points is sent to each worker node only once. At the beginning of every iteration, master sends the updated centroids to the workers, which then assign their data points to centroids, count the number of points belonging to each centroid, and calculate the sum of their points' contribution to the centroid's update. The workers send this information back to the master, which, once it has gathered the iteration responses from all the workers, calculates the total update to the centroids, and if needed, redistributes the centroids for another iteration.

3.2. Implementation

Both master and workers run on Tornado, a light web application framework implemented in Python, and communicate via a REST interface over HTML. Commands and data are encoded as JSON, and sent as POST requests. Requests from the master are of the form:

{"id": <job id>, "action": <actions>, "centroids": [centroids], “points”:[data points]}

The ID field specifies the job id that the request is being sent for. The worker maintains a mapping of job id’s to data points, allowing for multiple requests to be processed concurrently, using the job id to ensure they are done on the right set of data.

Five actions are implemented:

- points: current request includes data points assigned to this worker
- centroids: current request includes the centroids
- go: all the necessary data has been sent, and the worker is free to recalculate the centroids
- evaluate: asks the worker to calculate the distance of centroids to their assigned points, in order to perform the Davies-Bouldin index evaluation of the clustering (see Section 3.3)
- end: tells the worker that the current job is done, and it can clean all data points related to it from memory

To a “go” request, the worker will reply with :

response = {"id":<job id>, "recalculated_centroids": [{"adjustments": [adjustments per coordinate],
"point_count": <point count>}, ...]}

recalculated_centroids is a list of adjustments calculated per centroid. adjustments are represented as a list, each number suggesting the adjustment to the attribute at that index. Point_count is simply number of points associated with this centroid. The response to a “evaluate” request follows a similar format, sending aggregated distances from a centroid instead of suggested adjustments.

From the master’s perspective, the workflow is as follows:

- receive a GET request at <masterIp>:51000/master/*fieldname/num_centroids*
- retrieve all data points on users from *fieldname* from MySQL database
- normalize the data. This is done by taking the smallest and the largest value for each attribute, project them to 0 and 1 respectively, and projecting all other values onto that interval proportionally to their original value
- split the data into equally sized chunks and send it to workers
- generate *num_centroids* centroids. They are generated by getting a random value for each attribute.
- send the centroids to each worker

- on separate threads, send each worker the “go” signal
- wait for threads to join
- aggregate the adjustments to the centroids
- if the centroids have not changed significantly, send the centroids “evaluate” message
- aggregate the distances of points from centroids and evaluate the clustering

After all processing is done, the master replies to the original GET request with a colorful representation of the clustering results(more on this in Section 3.3).

3.3. Analysis of Clustering Results

One of the limitations of the K-means algorithm is that the clustering it produces may make sense from a geometrical point of view, but not tell much to the humans otherwise. Moreover, as we don’t know a priori how many clusters to expect, we need to experiment with different cluster sizes until we find a convincing one.

My original hope was that I could rely on one of clustering evaluation indices to determine which is the optimal number of centroids to use. Specifically, I had decided on Davies-Bouldin index, which favors clusters with short distance between data points and their relative centroids, and long distances between centroids. The index is calculated as:

$$DB = \frac{1}{n}\sum_{i=1}^{n}\max_{j\neq i}\left(\frac{\sigma_i + \sigma_j}{d(c_i, c_j)}\right)$$

where n is the number of clusters, c_x is the centroid of cluster x, σ_x is the average distance of all elements in cluster x to centroid c_x, and $d(c_i, c_j)$ is the distance between centroids c_i and c_j.

I discovered that the DB index of Stack Exchange users clustering is the lowest when using only two centroids - and in that case, the users are simply divided into and active and passive group. While this might be the most obvious cluster, it also does not provide me with any information I did not know already, and it is quite unsatisfying. Thus, I decided to rely mainly on how much sense the clustering makes to me, given the experience I have using the Stack Exchange forums, while still using the DB index as a sanity check for how good a clustering is.

As interpretation of clustering is mostly reliant on a human being, I needed a good way to visually represent what the clustering looks like. Once again, I drew inspiration from CCP, and represented the clusters as rainbow bubble charts (Fig. 3). Groups represent cluster centroids, and each centroid represents its attributes with bubbles whose area is proportional to the value of that attribute. As attribute values are normalized in range between 1 and 0, the bubble areas only represent the value of an attribute relative to the minimal and maximal values found for that attribute.

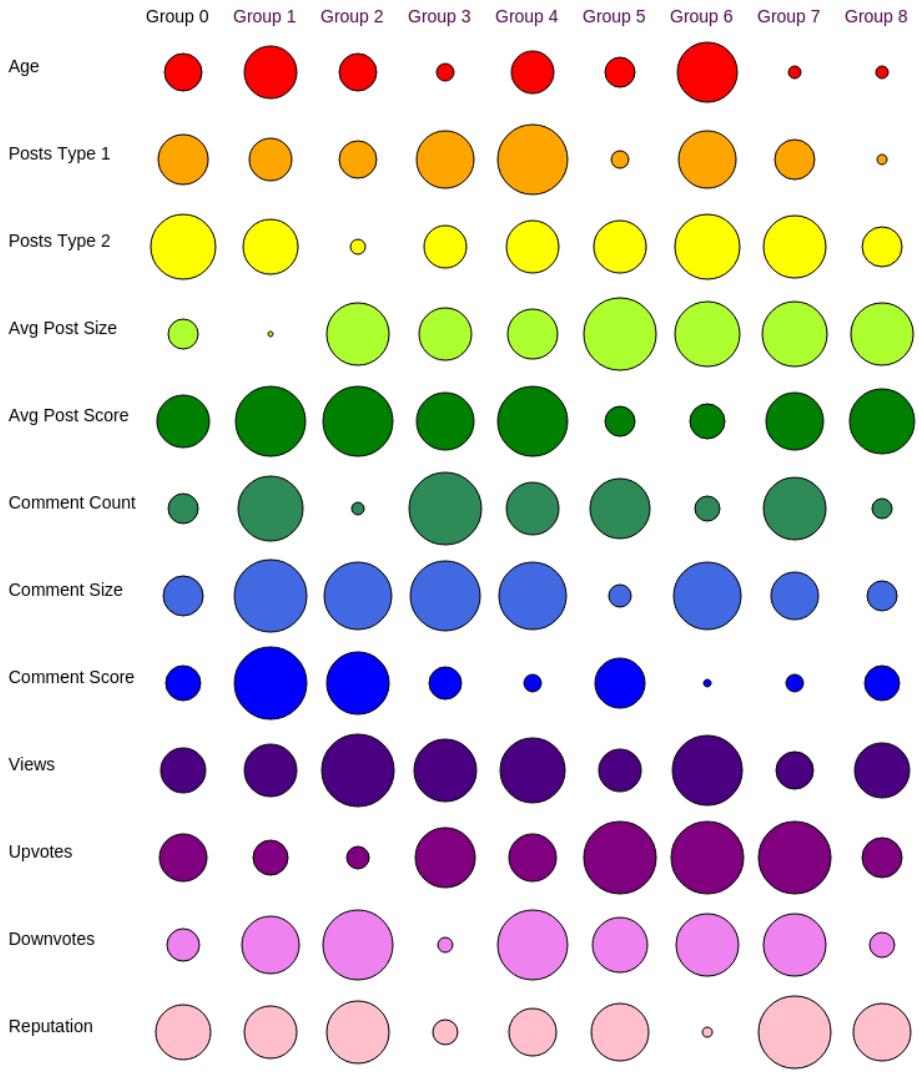

Fig. 3: Rainbow Bubble Clustering Representation

3.4 Performance of Distributed K-Means

I have measured the performance of my implementation of distributed K-Means on the three machines on my cluster. I took the 1 node setup, with master and worker running on the same machine, to be a decent approximation to the non-distributed algorithm; in fact, the only difference is in the request passing between the two components, which should be negligible, as they reside on the same machine. I performed clustering on data sets of different size, varying from 5000 to 30000 data points, and took the average time it takes to complete the processing when the computing is distributed among 1, 2, or 3 nodes. Results are shown in Fig. 4.

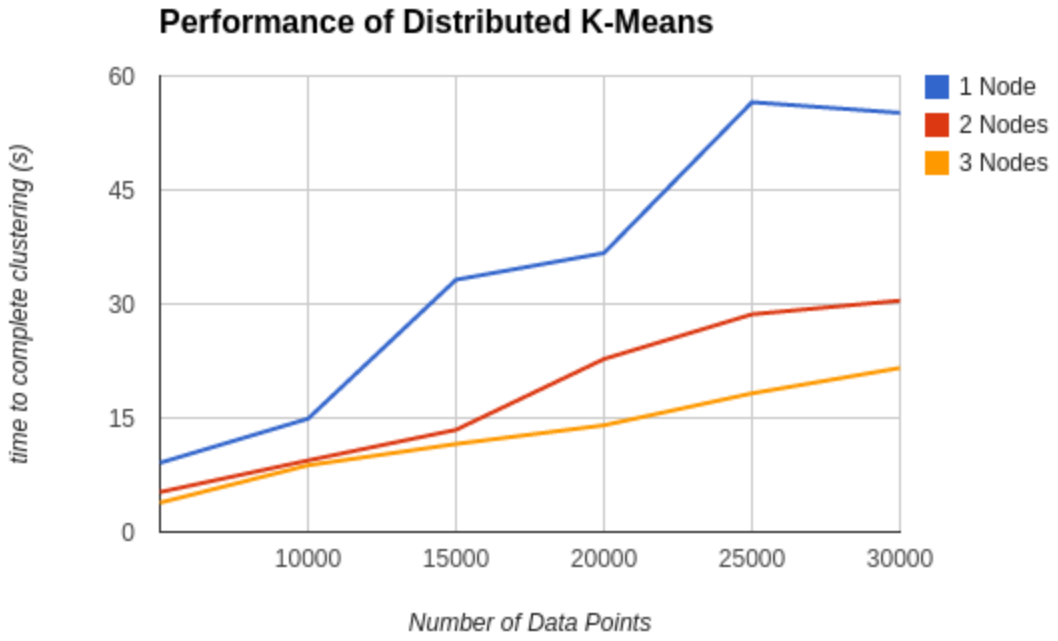

Fig. 4: Performance of Distributed K-Means

We see that adding a second worker roughly halves the clustering time. Adding the third node yields a smaller return, especially with datasets smaller than 15000 points, when the performance is very close to that of a 2 node setup. Message passing overhead might account for that. However, we see that for bigger datasets, 3 node setup improves on the 2 node setup by approximately one third.

While the performance increase is satisfying to observe, it seems that we are getting diminishing returns when adding new nodes. It would be interesting to do more experiments on larger clusters and with larger data sets to see how does the implementation scale.

4. Data Analysis

Having all data on users aggregated and summarized into a database has made it easy to analyze trends of user behavior in the different forum. Section 4.1 shows some examples of what can be learned about user age, post sizes, and downvotes. All data in this section has been analyzed using pyplot library. Section 4.2 discusses the results of K-Means clustering on the dataset.

4.1. A First Look At User Behavior

The first thing I was interested in seeing was the age distribution of users on different field forums, which is shown in Fig. 5. I have filtered out all users younger than 13, as the website does not allow users to be younger than that, so there must have been some mistake if it happens (or they had not specified their age in their profiles, in which case it is shown as 0).

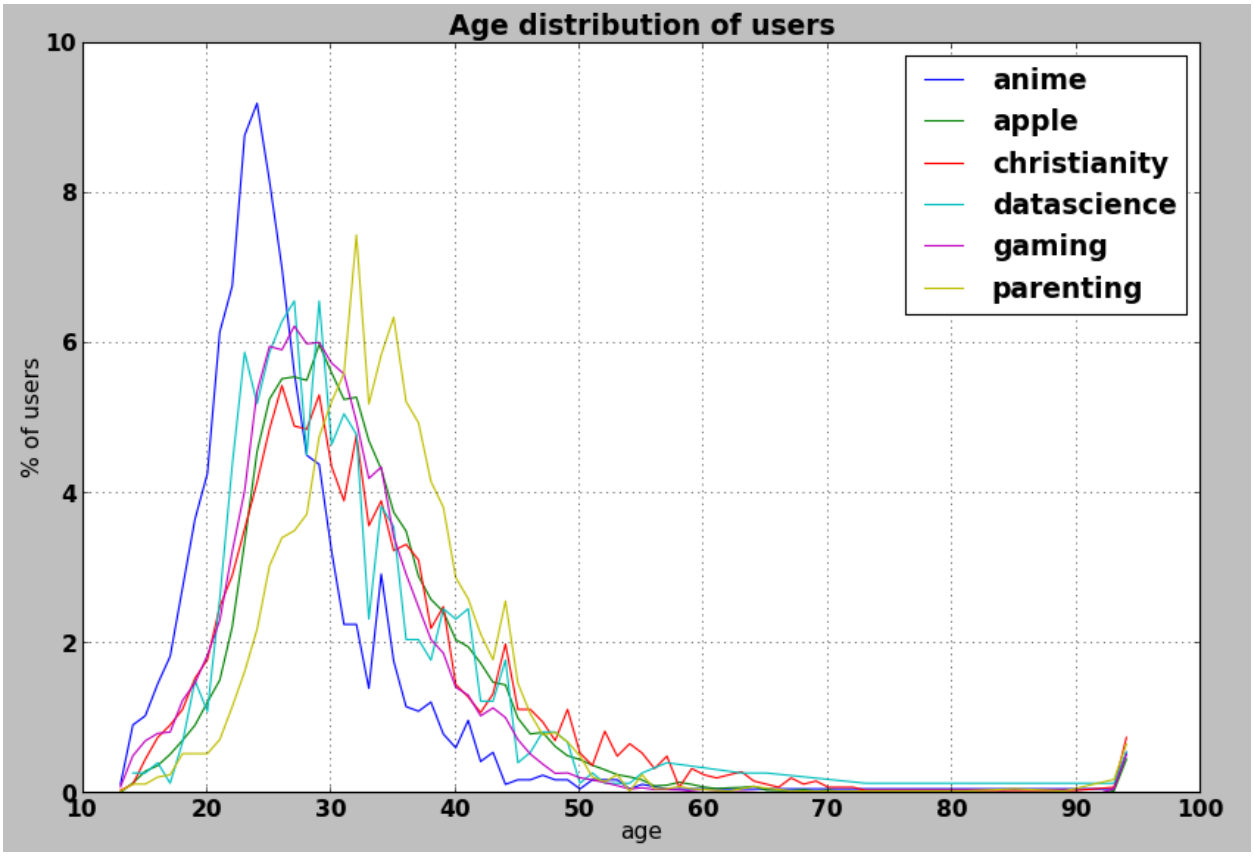

Fig. 5: Age distribution of users in different fields

There are a few interesting things we can say by looking at the chart. Anime seems to be an early twenties obsession, and people quickly lose interest in it afterwards. Data scientists, on the other hand, have a somewhat ‘blunt’ peak spanning almost a decade. People start getting very interested in parenthood in their early to mid thirties, and the topic seems to regain popularity in mid forties (which

approximately coincides with when their children are becoming teenagers). Interestingly, every field has an unexpectedly high population of ninety year olds; but this is probably due to the fact a lot of people on the internet seem to find it funny to set their age to very old.

I decided to focus on downvotes next. A user receives downvotes from other users if they find the content of a post or comment to be wrong, offensive, or otherwise problematic. Fig. 6 shows the average number of downvotes received by users of the same age.

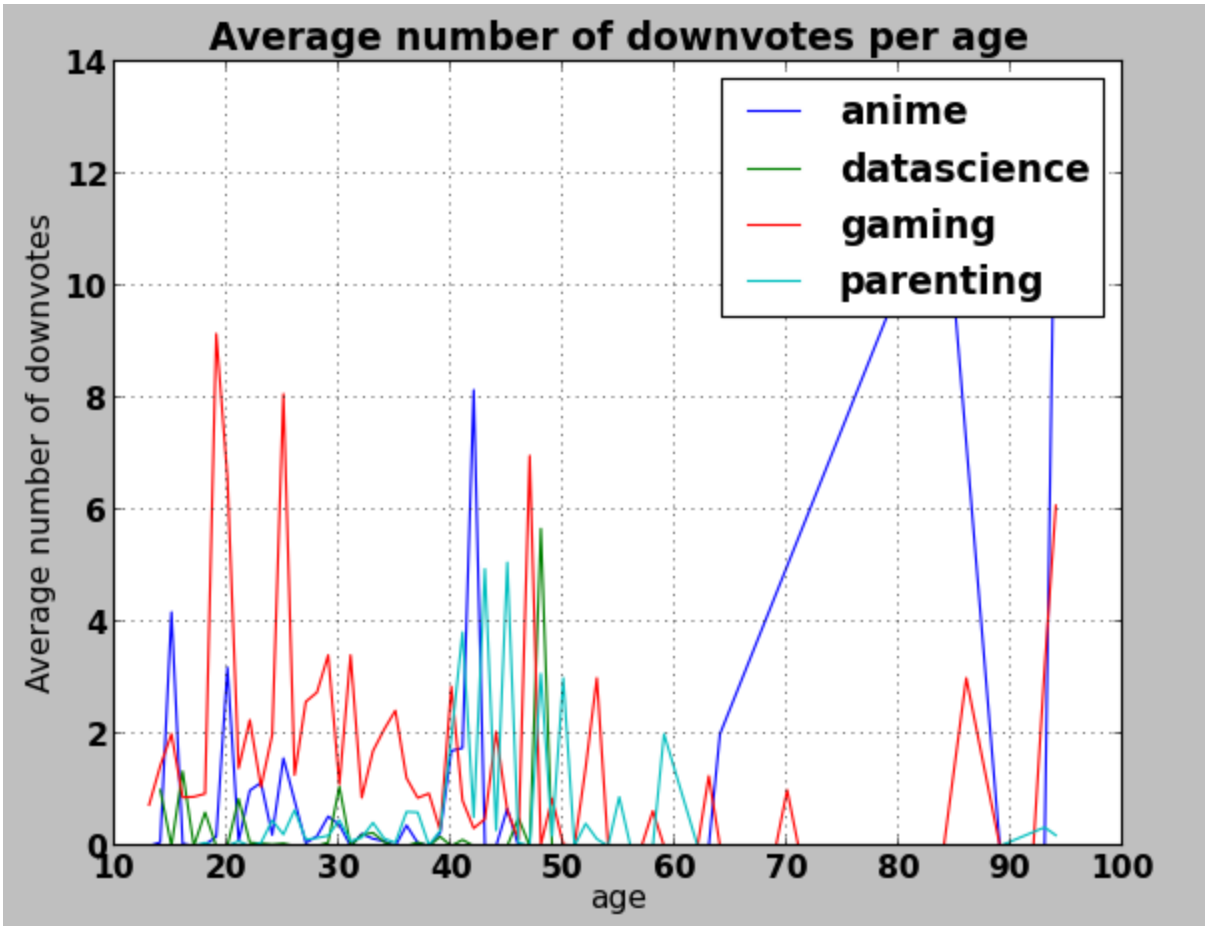

Fig. 6: Average number of downvotes by age

The graph shows the ‘obnoxious teenage gamer’ effect. It also seems that people become quite irritating when talking about parenthood in their forties. Also interesting to note are the spikes in downvotes among elderly anime and gaming community. My interpretation is that this happens due to the ‘trolls’ - people who enjoy provoking others on the Internet. It is likely that trolls make a separate account for their trolling activities - and they are probably the same people who think it is humorous to present yourself as a 90 year old. Data scientists don’t get downvoted much - partly because it is not a very trollable field - until they get to 48 years of age. It seems their whole world falls apart at that point.

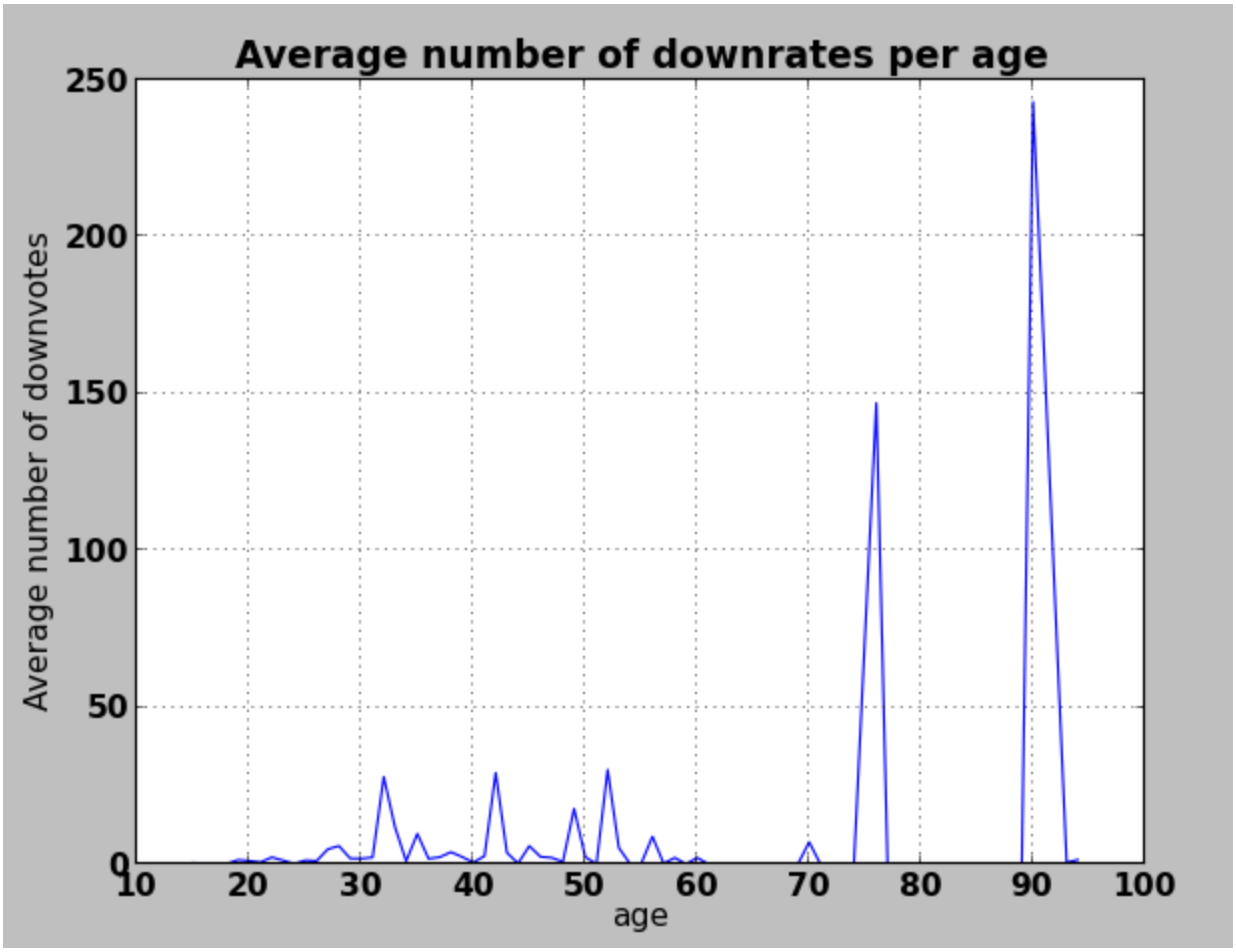

Fig. 7: Average number of downvotes by age on christianity forum

The christianity forum shows an extreme case of awful 90 year old people. Which is compliant with the troll theory, because religion is something people tend to have strong opinions about, making it that much juicier for trolls to prod them with a stick. Moreover, There are peaks in downvotes at the beginning of a new decade in life. Which I am not sure how to interpret, but it might be interesting to look more into.

Fig. 8 shows the distribution of post sizes, and effects of post size on downvotes. The graphs are constructed with aggregated data from all fields. We see that posts generally tend to be short - majority of them is around 400 characters. My hypothesis was that short posts would tend to get more downvoted, because they are less likely to contain useful information. We see, however, that the downvote spikes occur with very long posts. This might be justified by the fact that the charts show average downvotes per post of a given length. Since most posts are short, and downvoting is somewhat rare, their real volume is concealed in the graph. It is still surprising to see that very long posts tend to be so unpopular. I would have assumed that someone who writes a 6000 character essay on a topic voluntarily would know what they are doing.

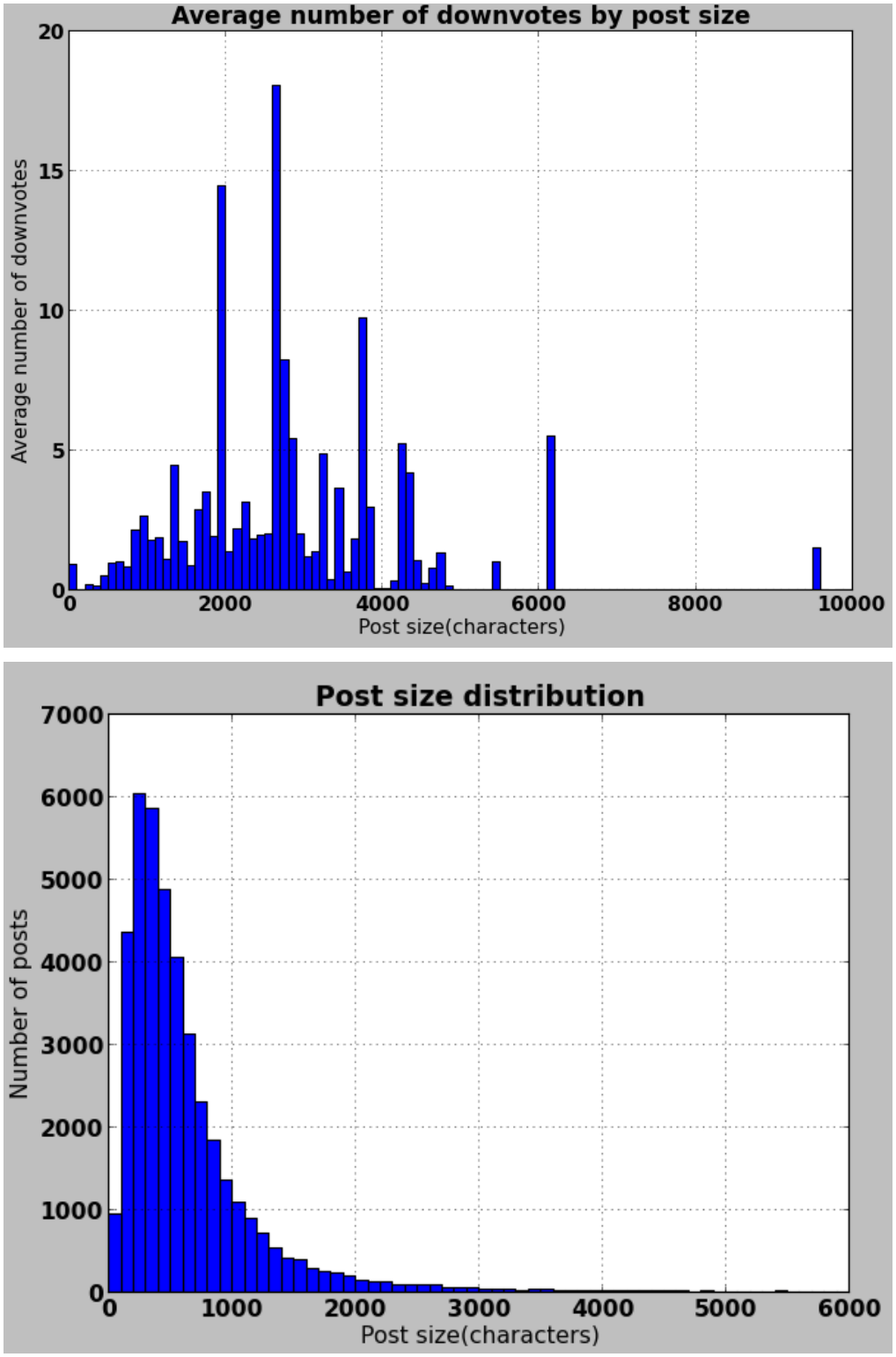

Fig. 8: Post size distribution, and effects of post size on downvotes

4.2. User Clustering

I ran the K-Means clustering on different fields. After experimenting with centroid numbers, I have decided that 8 centroids worked well, based on a compromise between DB index and understandability of results.

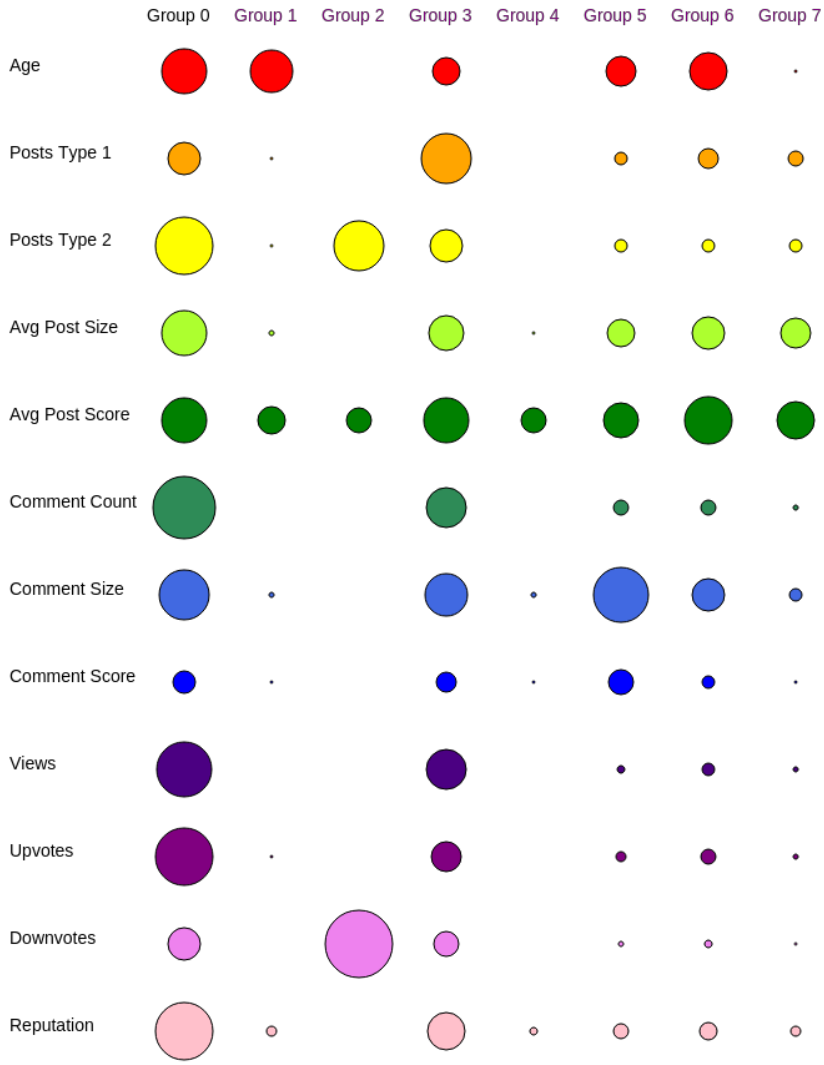

Fig. 9: Beer forum user clustering

For the users of the beer forum (Fig. 9), I have created the following user categories:

The Inactive User: Groups 1 and 4. These are users who almost never posted anything. They are separated in two groups because some of them set their age.
The God Among Men: Group 0. A user who enjoys answering questions and being helpful. This is the kind of user that has saved my life many times during my education - although on different forums.
The Newbie: Group 3. This user mostly contributes new questions.
The Casual User: Groups 6 and 7. They occasionally log in to ask a question.
The Ent: Group 5. This user doesn't participate in conversations often, but when they write a comment, it is a large comment that gets a good score.
The Troll: Group 2. They only ever reply to posts, and only ever get downvoted.

Fig. 10: Anime forum user clustering

The clustering of anime forum users shows some similar groups (the Ent in Group 7, Inactive User in Group 6 and Group 0, the Troll in Group 5). However, the difference between people who mostly post questions and those who mostly answer them is less pronounced.

5. Conclusions

The project has proven being a good starting experience for Hadoop. While I have learned the basics of setting up and managing a cluster, I cannot say I have witnessed the full potential of the framework. I was, in fact, working on a small cluster and running very small machines, which made Map Reduce jobs slow - in fact, processing files on my local machine was many times faster than running the Map Reduce jobs.

After processing the data, I was able to see some interesting trends in age, posts size, and downvotes among the users, and the dataset I extracted from the posts dump is ready to answer many more questions about user habits. In hindsight, however, it would have been good to also include the date a user joined a forum in the dataset. The age analysis can, otherwise, be problematic. The Stack Exchange forums launched in 2009, which means we can have scenario where a user joined 5 years ago, was very active for a year, and then went inactive. In the analysis I have done, however, their activity is not associated with the age at which it occurred, but instead with their current age.

The Distributed K-Means implementation is quite satisfying. The performance results presented in Section 3.4 were expected - although the gain in performance when adding an additional node is slightly lower than I had hoped for. Running the algorithm on the users of Stack Exchange has produced some intuitive categories, although the results of clustering change significantly between different runs of the algorithm. I have noticed, however, that there are a few group configurations that alternate, which probably means that if I ran the algorithm many times, I could group the results into a few categories, and then pick which category offers the best clustering. It would provide more robust results. Moreover, it would be good to see how would the clustering do if I reduced the number of attributes, to alleviate the curse of dimensionality.

6. References

[1] http://dogdogfish.com/2014/04/22/hadoop-from-spare-change/

[2] Zhao, W., Ma, H., & He, Q. (2009). Parallel k-means clustering based on mapreduce. In *Cloud Computing*(pp. 674-679). Springer Berlin Heidelberg.

[3] Maulik, U., & Bandyopadhyay, S. (2002). Performance evaluation of some clustering algorithms and validity indices. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, *24*(12), 1650-1654.

Appendix A: Using Distributed K-Means Servers

The program interfaces with MySQL database, so it will be necessary to have a server set up. I am providing a dump of the dataset I have processed and prepared for clustering.

The requirements.txt file lists python packages needed to run the program. I recommend installing them with pip with

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

The config.json file specifies the configurations the program uses. By default, master and workers will run on ports 51000 and 51001, respectively. Additional nodes can be added by simply adding their IP address to the list of workers.

The servers are started with

```
python master.py
python worker.py
```

Once the servers are started, sending a GET request to

localhost:/51000/master/fieldname/cluster_number

which will cluster the users of *fieldname* into *cluster_number* clusters.