Beer Prediction Analysis

Joel Bremner

Note* Attached in this file are two notebooks, Main, is the one I wish to be marked, Testing was simply used to run the project in entirety in order to test the times. The project may also require some pip installs, but as they are now installed, I'm unsure as to what is required. The project has been developed in entirety in vs code, and hasn't been tested using collab, so can't guarantee it will work. So to ensure it works, please mark in VS code.

Summary:

I intend to use the RateBeer dataset to research the question, can a consumers' (More specifically someone who leaves reviews on beers) preferred beers be predicted based on the reviews they've left on previous beers they have consumed. I plan to implement a utility matrix, with collaborative filtering using cosine similarity to solve this question. The utility matrix will be populated by the ratings for the beers, as well as types of beer. The final result, should hopefully be able to highly accurately predict which beer a reviewer is likely to rate high, and thus enjoy. The intended result of this project is to be able to accurately predict a user's beer preference. This would be a significant result as it would confirm there is a pattern of behavior to which a beer drinker consumes alcohol.

Background:

These datasets include reviews with multiple rated dimensions. The most comprehensive of these is the beer review dataset from Ratebeer, which include sensory aspects such as taste, look, feel, smell as well as the beer style / type. The dataset also contains fields for beerId and userID. The dataset contains over 2.8 million records.

The algorithm I plan to use implements a utility matrix; which will be populated with the rows being unique users, the columns representing unique beers and the values being users ratings of the given beers. I will perform collaborative filtering with cosine similarity on the populated utility matrix. I will take the weighted (using cosine similarity) utility matrix of the 10 closest users (users with the highest cosine similarity), and create a combined weighting rating of each beer in the utility matrix, recommending the highest rated one to the user. One of the cons of this algorithm is it needs a large dataset to start off with; however this is not an issue as there are over 2.8 million reviews in the dataset. And being that all these reviews are rating beers, there will be no sparsity issue.

Motivation:

Like most university students, I enjoy alcoholic beverages from time to time. My drink of choice often being beers. So naturally I am drawn to the idea of further diving deeper into the public's consumption habits of alcohol and more specifically beer. When taking a face value observation of my drinking habits, I cannot find any obvious pattern, when it comes to the types of beers I drink; however it will be interesting to see if there is an underlying pattern, and I hope that my proposed project implementation will be able to reveal that. If this project proves successful I'll then be able to share this with friends and family as well as myself, to find other drinks we may possibly enjoy.

There is a large variety of people who drink beer, from those with no preference who are happy to drink whatever, to connoisseurs who are extremely picky about what they are consuming. So to look into the behavior of these varied individuals also peaks my interest.

Research question:

My proposed research question is can a consumers' preferred beers be predicted based on the reviews they've left on previous beers they have consumed. These questions will help us to understand if there is a pattern in consumers behavior when it comes to consuming alcohol. Implementing collaborative filtering using cosine similarity with a utility matrix will provide a systematic approach to uncovering patterns in consumers' beer preferences based on their past reviews. By constructing a utility matrix that represents consumers' ratings of different beers, the algorithm will be able to identify similarities between consumers and beers, enabling it to make predictions about consumers' preferred beers. This predictive capability is essential for understanding whether there exists a discernible pattern in consumers' behavior when it comes to consuming alcohol. By analyzing the recommendations generated by the collaborative filtering algorithm, we can discern trends and tendencies in consumer preferences, shedding light on the underlying factors driving beer consumption choices. Ultimately, this approach offers a data-driven means to address the research question, revealing insights into the complex interplay between consumer behavior and beer preferences.

Experimental Design and Methods:

There are Multiple important parts of my projects code:

- The first part of the project is downloading and formatting the data into a useful format. I download the dataset in a gzip from the internet using the urllib.request libraries method urlretrieve. I then extract the data out of the gzip into a json file using the gzip library to open it, and "shutil" library to copy the contents of the file out to a json file. From here I read this json file, storing the contents as a list of python dictionaries. I only store the fields which are important to the algorithms, in order to reduce the data being stored and handled later on down the line. I then write the list of dictionaries out to a json lines file (.jsonl). After I've done that, the setup is done, and we are now ready to process the data.

- Next part is setting up the dask components. I start off by starting up the clusters, here is where we can decide on how many cores we want to use. Next we start up the dask dashboard to monitor the workers as they work in parallel. The link is printed out, follow this link to view. Next I load the json line file into a dask bag of python dictionaries. I am then able to take a random sample of this bag if I choose to. This is helpful for testing.

- From here the algorithm begins. We start off by filtering the data out of the bag, which isn't required and will skew the results. This also helps to reduce unnecessary data in the data set. We create a dictionary of the frequencies of which users and beers occur in the dataset. Using this python dictionary we are then able to filter the dask bag by using these dicts, by setting a threshold for the amount of beers a user has reviewed, and the amount of beers that have reviews.

- Next I create a mapping from the beerId to the name of the beer. Followed by getting a list of unique beerIds and unique profiles. These unique beers and profiles are stored in an array list. The index in which these occur in the array list from here on out will be their true indexes. We then create a utility matrix of all 0s of the size given by the size of unique beers and unique users.

- Next we create a hashmap for the "beerId" to its true index, and "profile" to its true index. This allows us to more quickly access a user and beer's value using its true index. This also saves on computing cost, as these hash maps or faster than traversing arrays.

- Next we populate the utility matrix, this can't be done in parallel, as some cores may update the same user vector (row of utility matrix) at the same time, this results in one over writing the other. In order to make this as parallel as possible, I get the values' true indexes in parallel, then loop through these values sequentially populating the utility matrix.

- Next I prompt a user to select an existing user to recommend a beer for.

- The next step is the comparison using cosine similarity. In parallel, we compare the test user with each other user vector in the utility matrix. Storing the cosine similarity for each one. From here we are able to take the top results, I'm my case I'm using 10.

- Next we loop through the utility matrix, and multiply each user vector by its corresponding cosine similarity. Giving us a weighted utility matrix. From here we can calculate a total sum of weighted ratings for each beer. Then taking the highest rated beer, the algorithm can then recommend this beer to the test user.

Libraries:

- Urllib.request: used for downloading the data set from the internet
- Gzip: use gzip.open for reading the contents of the .gz file
- Shutil: use shutil.copyfileobj for copying the contents of the gzip to a json file.
- Json: for json.loads, for converting json into python dictionaries. As well as json.dump, dumping a dictionary out to a json file.
- Dask.distributed: for creating a LocalCluster, to allow parallel processing. Client for supplying a dashboard for viewing the tasks.
- Dask: used for dask bags, for storing data to be processed in parallel
- Numpy: for np.zeros, creating lists of all 0s.
- Gc: Used for garbage collection.

Functions:

- Unzip_gzip: extract a gzip to a json file
- Json_to_jsonlines: convert a json file to a list of python dicts
- Convert_to_jsonl: convert a list of python dicts to a jsonl file
- update_sinle_record: for getting the corresponding true indexes of a beerId, profileId and rating.
- Clear_worker_data: run garbage collection.
- Cosine_similarity: calculate cosine similarity for two vectors
- Calculate_similarity: calculate cosine similarity for a utility matrix
- Get_weighted_matrix: get a matrix where each user vector is weighted by its similarity to the test user.
- Recommend_beer: Given the weighted matrix, recommend a beer to the test user.

Data And Analysis:

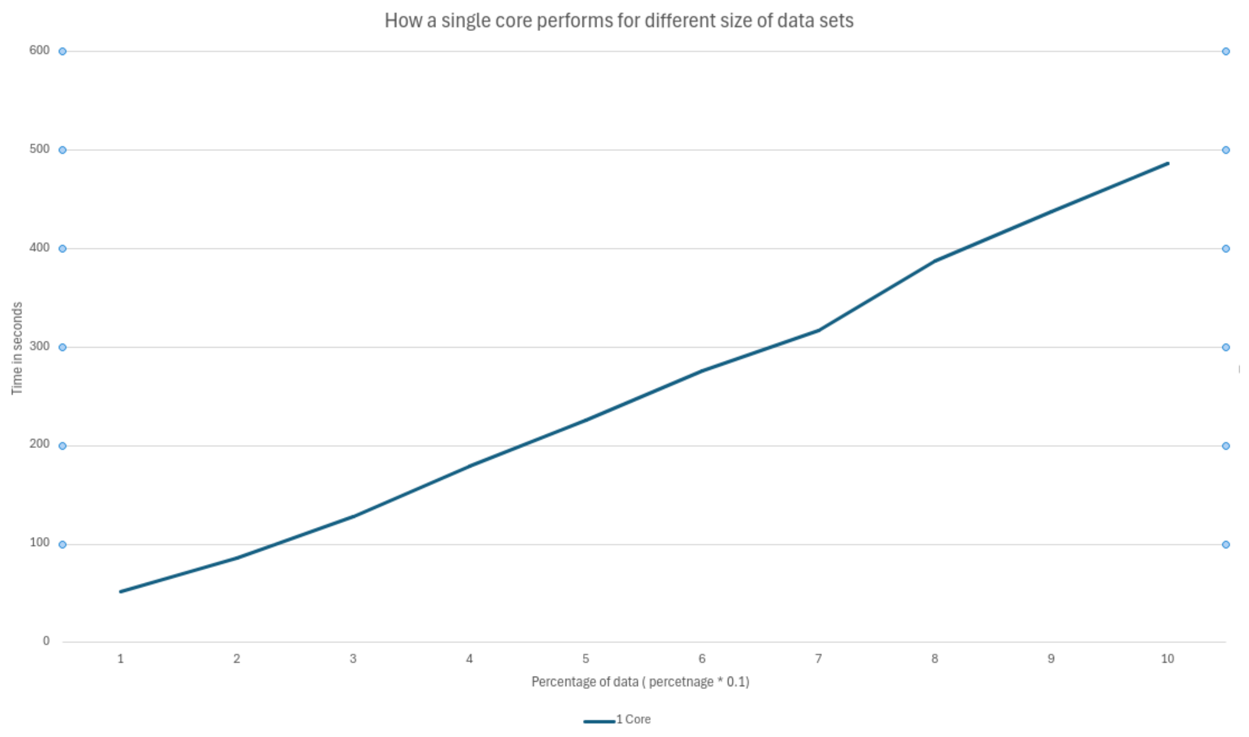

Figure 1: Outlines how 1 core performs on varying amounts of the data set (0-100%)

The graph shown in figure 1 depicts that the algorithm is linear. This would line up with what is in the code base, as there are very few instances of nested loops. And where these nested loops occur, there the data being iterated through is an extremely small constant subset (10 items) of the larger problem. Apart from these few exceptions, every operation is done in order n, O(n). We can predict the amount of time taken to perform an analysis on a given percentage of the data using the formula, time = 4.8*(percentage of data). From here we could estimate and of course only speculate that performing an analysis on 10 million items of data would take a time of approximately 1736 seconds or roughly 30 minutes.

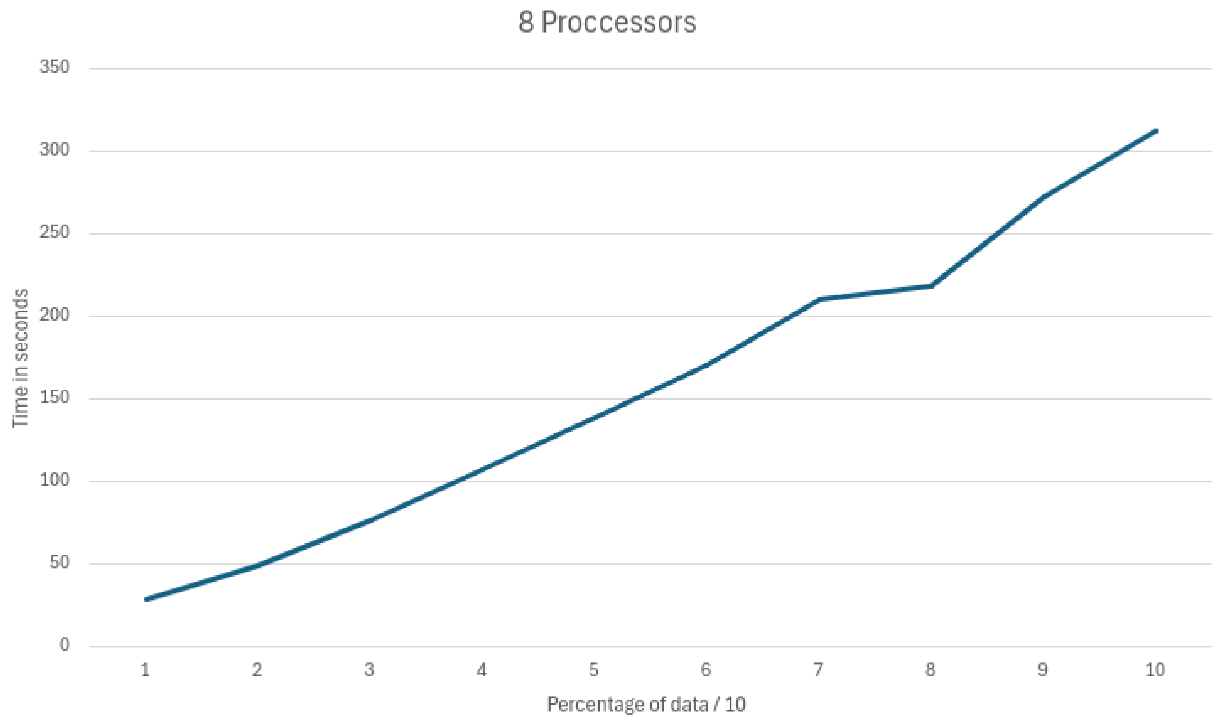

Figure 2: How 8 cores performs over varying amounts of the data set (0-100%)

This graph depicted in figure 2, outlines once again that the algorithm is linear even when running in parallel with 8 cores. Once again this makes sense due to no nested iterating on the entirety of the data set, only a small subset constant. Every other part of the algorithm operates in O(n) or less. We can predict the amount of time taken to perform an analysis on a given percentage of the data using the formula, time = 3.15*(percentage of data). From here we could estimate and of course only speculate that performing an analysis on 10 million items of data would take a time of approximately 1124 seconds or roughly 19 minutes. A decent improvement but clearly not fully parallel as that would be 8 times faster and would be able to perform an analysis on 10 million items in under 8 minutes.

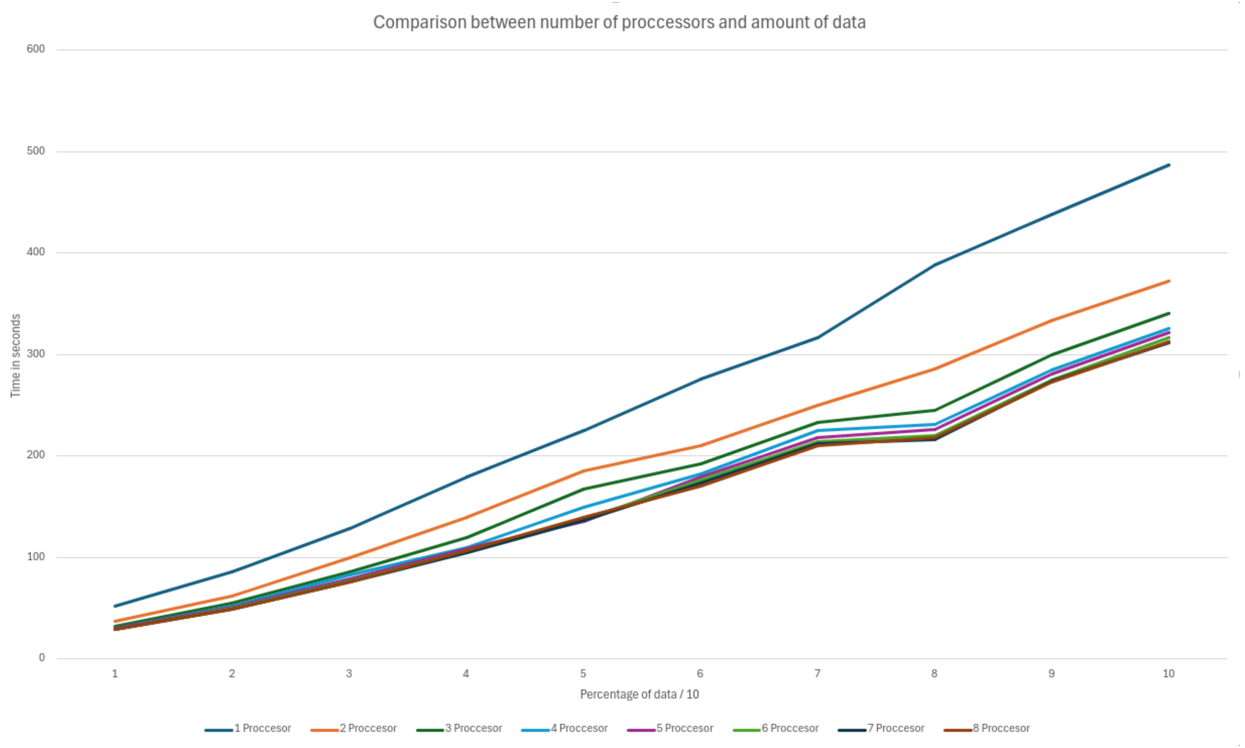

Figure 3: This graph outlines how different amount of cores (1-8) perform on varying amounts of data (0-100%)

As you can see in the above graph, there is a noticeable improvement on the performance of the algorithm in parallel, when going from one core to 2 cores. 2 cores to 3 cores. And a slight improvement going from 3 cores to 4 cores. After 4 cores the changes become quite negligible in comparison. Although still an improvement, it isn't very notable. From here we can depict that using cores up to 4 cores on the given data set size will give a noticeable benefit, but anything after that is not worth the computing resources. The reason for the decreased benefit of extra cores, is due to the fact that the time of partitioning the data out across the workers, is out weighing the benefits from which having the extra cores is giving you.

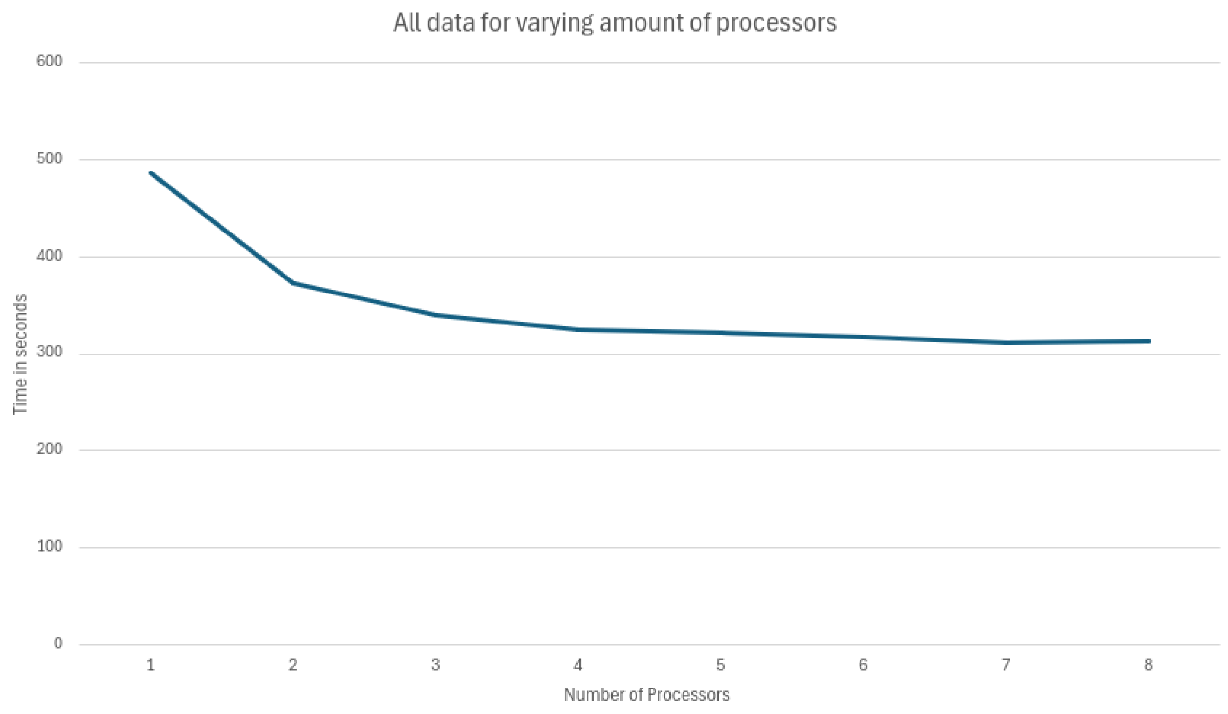

Figure 4: This figure outlines how the project performs on 100% of the data over varying amounts of cores.

The above graph shows that there is some benefit to using parallel computing with this algorithm and data setup until around 4 cores, where the line begins to flatten out. This tells us that it is partially parallel, however after 4 cores the overhead of partitioning the data is outweighing the benefit gained. Although there are still slight improvements up until 7 cores, where it fully flattens out; the improvement is negligible. This is more evidence for us to recommend the use of 4 cores, in order to not waste computing power.

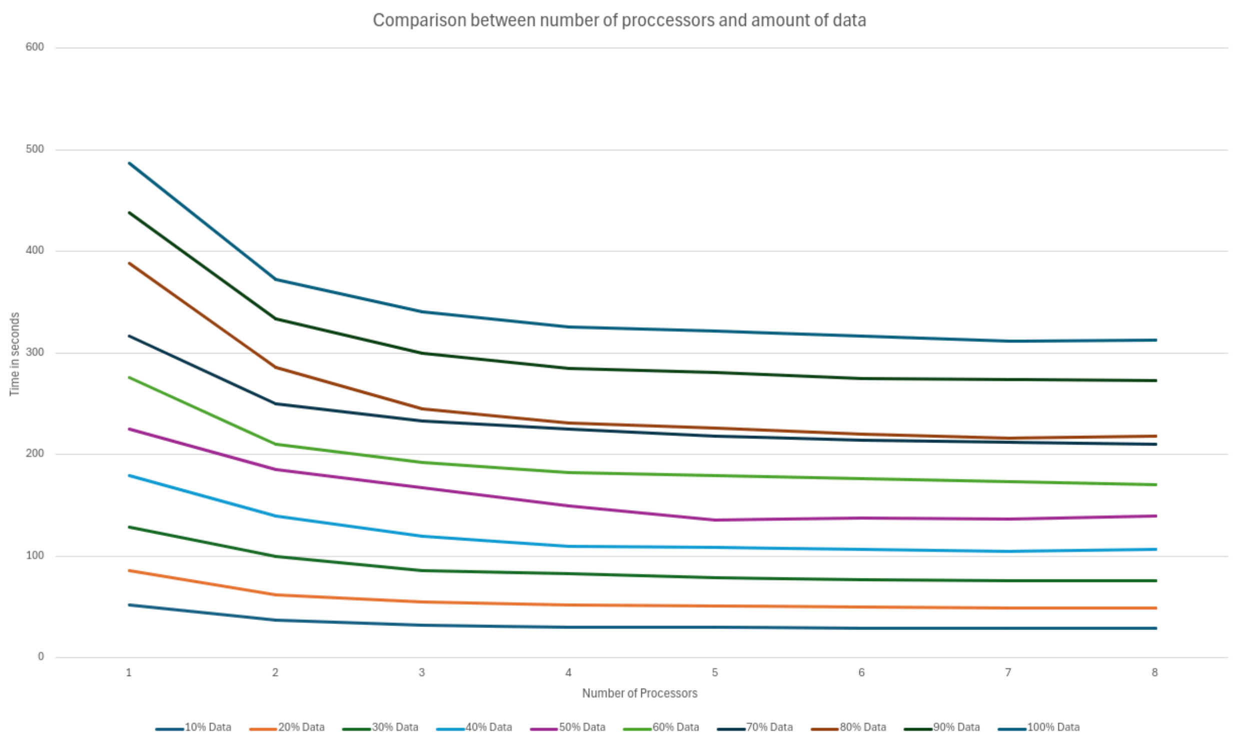

Figure 5: This figure shows the comparison of varying amounts of the data set, over the amount of cores.

The graph shown in figure 5 compares how the amount of cores is affected by the amount of data it is analyzing. For 100% of the data, as mentioned above, 4 cores is where it starts to flatten out. And that stays true for 100%-40% data. After this, 30% of the data begin to flatten out around 3 cores, 20% and below around 2 cores. After these values, the change becomes negligible and isn’t really worth the computing power.

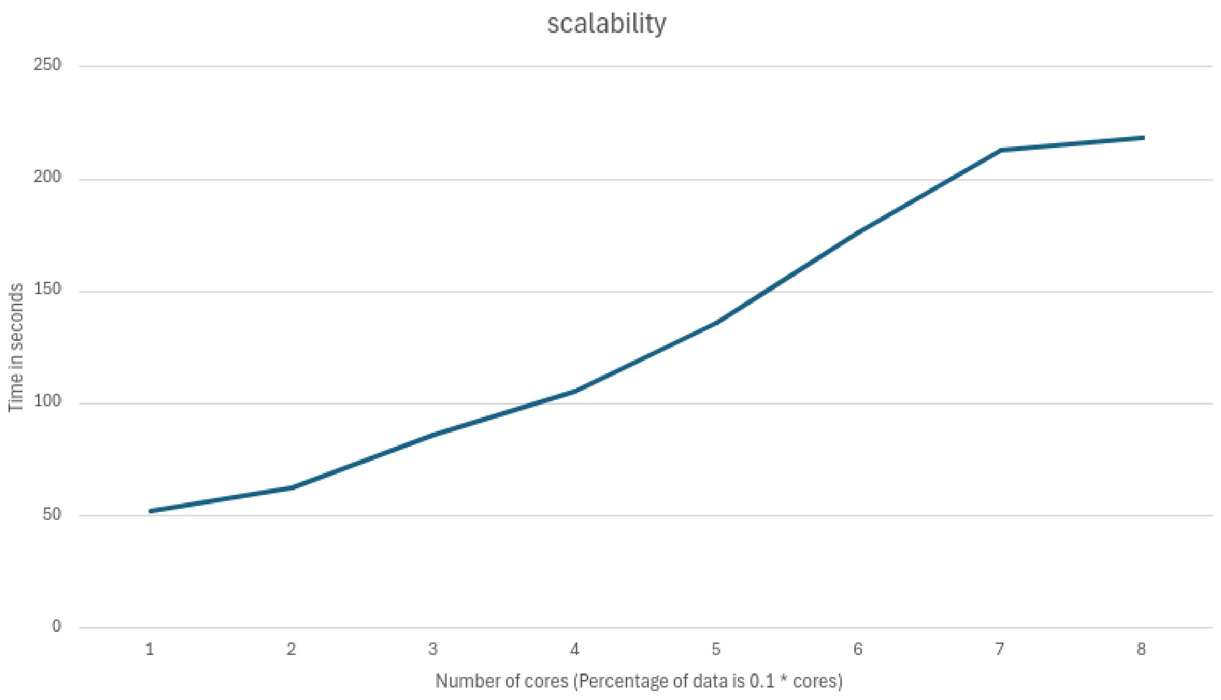

Figure 6: This figures outlines the performance of the project, as the number of cores increase at the same rate as the percentage of the data set being used.Eg. 10% = 1 core, 20% = 2 cores, … 100% = 10 cores.

The graph in figure 6 outlines that my algorithm is slightly scalable, however it is weak scalability. Ideally if it was fully parallel, we would have a horizontal line, with the change in data and cores not affecting the time taken. That is, with the amount of cores and data increasing at the same rate, the time would stay constant; however this is not the case. As my line has a gradient of around 2.4; we are aiming for it to have a gradient of 0.

Although my research question does not mention anything about the performance of the algorithms, and their parallel scalability, there are still some interesting conclusions we can draw from it. As the analysis is performed in O(n), or as far as we can see, if we wished to scale the data set up in order to hopefully increase accuracy, the time loss would be proportional. As discussed earlier, if we wanted 10 million records instead of the current 2.8 million, it would only require 18 minutes, using 8 cores. This is still a completely reasonable time. As well as this, we can also depict for data sets up to 3 million records, using anything over 4 cores is unnecessary and just a waste of computing power, as the change after 4 cores is negligible.

Conclusion:

Was I able to answer my research question? Ultimately, I had some oversight in how I was going to test, what was an accurate prediction. The research question outlines whether we are able to accurately predict a user's beer preference. There are two parts to that; can we predict a users beer preference, and can we do it accurately. I think I covered the first part. After inputting myself with my preferences as the test user, I am recommended a beer. Having not tried the beer however I cannot talk on the accuracy, but on first glance it doesn't seem out of the realm of possibility. To answer the research question itself, the answer is no. No, we can't accurately predict a user's beer preference based on their past drinking behaviors. We can predict a beer, but I'd conclude that it is not accurate. I believe there are a couple reasons for this. People who like to drink beer, like beer for what it is. Not for the specifics of the beer itself. This means, generally users will like most beers all roughly the same, this makes it hard to accurately supply one beer. I believe an easier to assess question would have been, can we accurately predict what beer a user would not like; but there is not much value in answering that question. On Top of this, it was hard for me to assess accuracy, having run the test on myself, multiple times, each time it would recommend a beer I had never heard of from another country. This makes it hard for me to assess whether this would be an accurate beer recommendation. Possibly a more local data set would have helped.

What implications do your results have? Overall, my results put forward much of an implication. My results were too inconclusive to be able to imply anything. However, based on my research and analysis I have personally concluded that beer consumers don't have a distinct pattern when it comes to their drinking behavior. But I have no concrete evidence to say that this statement is truthful.

In the future I'd like to run it on even more data, possibly including the second data set. From here I'd like to look at the performance of the algorithm in terms of time, with different amounts of cores. I'd also like to see if it was able to predict a more accurate beer preference for myself. I'd also like to incorporate a concrete way of having a confidence score, of the recommendation, as well as a measure of accuracy of how accurate the algorithm is.

Critique of Design and Project:

In practice I quickly found out that my proposed algorithm for recommending the beer itself would not work in practice, so I had to change this. In its place I now get the users cosine similarity, and multiply this by the ratings they gave for each beer. These ratings are then collated and the highest one is recommended to the user. This in practice works a lot better than my proposed method, of not doing any weighting, and just recommending a beer the query user hasn't had that every other similar user did.

The reason for the proposed idea not working. Is if you take the a user whose utility vector for example looks like the following [1, 0, 0, 1] and there is at least 1 similar user with the same binary utility vector, then there is no beer, that can be recommended cause the algorithm has to recommend a beer, all other users have had, but the query user hasn't, which is impossible in this instance, so would return null.

Reflection:

Useful course concepts and tools:

- Lecture notes - useful for further insight on how the algorithms worked
- Lectures - useful for further insight on how the algorithms worked
- Labs - useful for any questions, pairs or the internet couldn't help answer
- Dask documentation - useful for simple help with dask
- Dask Dashboard - useful for viewing what was happening in parallel

What did I learn from this project? While completing this project I fully gained an understanding and appreciation for parallel programming. The concept to me was a bit hazy until I got midway through this project, where things all started to click all together. I also learnt a lot about dask and its syntax itself, something I struggled with a lot previously. But I think the main learning points I had was the ability to adapt algorithms to meet different needs. After realizing my proposal design method didn't work the intended way, I was challenged with having to adapt it to make it work. After a lot of head scratching, I was successfully able to come up with a complete solution.

References:

- https://docs.dask.org/en/stable/graph_manipulation.html
- Lectures on learn
- Learning attitudes and attributes from multi-aspect reviews
  Julian McAuley, Jure Leskovec, Dan Jurafsky
  International Conference on Data Mining (ICDM), 2012
- From amateurs to connoisseurs: modeling the evolution of user expertise through online reviews
  Julian McAuley, Jure Leskovec
  WWW, 2013
- Dataset Overview
- Students worked with: Liam Ceelan-Thomas