Thrifter: An online marketplace with a recommender system

Ari Freyr Ásgeirsson
Ívar Oddsson
Sigursteinn Bjarni Húbertsson
Valgeir Björnsson
Ævar Ísak Ástþórsson

May 2015

B.Sc. in Computer Science

Supervisor:
Hlynur Sigurþórsson

Examiner:
Jökull Jóhannsson

Thrifter: An online marketplace with a recommender system

Ari Freyr Ásgeirsson
Ívar Oddsson
Sigursteinn Bjarni Húbertsson
Valgeir Björnsson
Ævar Ísak Ástþórsson

May 2015

Abstract

Online marketplaces in Iceland offer limited help when it comes to creating advertisements with a standardized look and feel or finding things that users are interested in. Thrifter is a marketplace system designed with the user in mind by both helping with the ad creation process and by recommending items based on recent user history. In this report we outline the design of the Thrifter platform and how the development was undertaken during the course of the final project.

Contents

List of Figures

List of Tables

1. Introduction

There is a general lack of quality consumer to consumer (C2C) marketplace systems in Iceland. People rely on forum websites to post their advertisements or even on Facebook groups with members in the tens of thousands. The problem with using such platforms is simple, they were not built for the purpose of creating ads and selling products. People post their ads on these websites where they are quickly buried in a pile of new posts, almost never to be seen again.

We saw a gap in the market and decided to develop a solution to this problem. Our main focus was to improve the current C2C marketplace systems available in Iceland. We liked the idea of software that was simple and could do one thing really well.

We introduce Thrifter, a unified marketplace where people can create quality advertisements, manage and edit them with ease while also being able to search and browse for other items to buy. Thrifter provides built in templates based on item categories which allow users to fill in only the appropriate information, resulting in quality advertisements with a standardized look and feel so that users can quickly size up the value and interest in the item. Thrifter also tracks user actions in the system. Actions like viewing an ad or buying items are logged into the user's history. We can then use this information to give each user a personalized advertisement feed with items that the user is most likely interested in.

We believe that by employing artificial intelligence, the user experience will improve greatly because the user won't have to spend as much time searching for and scrolling through items he has no interest in. Harold, the artificial intelligence (AI) in our system, is responsible for finding personalized advertisements for a given user. Harold is a recommender system, these systems have become very popular in recent years due to the growing creation of data and the need to filter this data.

One of the primary objectives of Harold is not only finding advertisements but being able to do so in a timely manner. To achieve this we researched some of the most popular clustering techniques and optimization methods for large scale systems[1]. By applying clustering and optimizing the way we find relevant users, we were able to handle searching one million recent user histories within 300 milliseconds. This research is discussed in more detail in Chapter 4 of this report. For an in depth explanation, see the research report.

To demonstrate the functionality of the system, we created a mobile app prototype. This prototype was written in the Ionic framework [2] which is designed to allow for the creation of mobile applications. This framework was chosen because we all had experience with writing web ap-

[1] Wu, X. (2007). Top 10 algorithms in data mining - University of Maryland `http://www.cs.umd.edu/~samir/498/10Algorithms-08.pdf`

[2] "Ionic: Advanced HTML5 Hybrid Mobile App Framework." 2013. 13 May. 2015 `http://ionicframework.com/`

plications and we could create a working prototype in a relatively short amount of time, while also being cross-platform. The prototype shows every current feature of Thrifter, ranging from simply registering a new user to being able to create and buy advertised items.

The work described in this report was conducted as a B.Sc final project in the School of Computer Science at Reykjavík University, spring 2015. The team consisted of five students, Ari Freyr Ásgeirsson, Ívar Oddsson, Sigursteinn Bjarni Húbertsson, Valgeir Björnsson and Ævar Ísak Ástþórsson.

The rest of this report is structured as follows. In Chapter 2 we discuss the organization aspect of the project. In Chapter 3 we take a look at the system architecture of Thrifter. In Chapter 4 we go into more depth regarding Harold and the AI research. In Chapter 5 we discuss what we learned from this process of creating a new system. In Chapter 6 we talk about the future and what we have planned for Thrifter. In chapter 7 we conclude this report.

2. Organization

When embarking on a large scale project like Thrifter, we found it important to keep a good work organization. During the work in this project the team used the Scrum methodology. There were a few factors that influenced the team's decision of using Scrum, over other development methodologies. The largest factors included:

- The team members know Scrum better than other methodologies.
- Scrum provides a visual representation of the project's progress.
- Scrum handles changes to projects well.

In the early stages of development of the project, the team proposed a few project goals for the team to work towards. These goals were twofold:

1. The team proposed finishing all **A** prioritized requirements set for this iteration of Thrifter. This iteration consisted of 69 **A** prioritized requirements and were made up of a total of 192 story points.
2. The team proposed it would turn in total work hours of no less than 1800 hours and no more than 2000.

After having finished the project, it was clear that the team had surpassed both of the goals set forth. The team ended up finishing a total of 211.5 story points and turning in a total of 2289 work hours. The reason why the team finished more story points than initially planned can be traced to a few modifications made to the project after development began. These modifications were mostly aimed towards the use of Facebook as a tool to gather information about users. When those ideas were scrapped, some **A** prioritized requirements became **B** or **C** prioritized requirements. To counter this loss of **A** prioritized requirements, the team finished a total of 15 **B** prioritized requirements instead.

After this stage of development, the team had contributed 2289 hours to the project. The reason for surpassing the total work hours proposed can largely be traced to the research done on the AI that Thrifter uses as a recommender system for its users. That particular part of Thrifter was difficult to work with given the team members lack of experience in AI research and development. It was therefore difficult to estimate precise hours for this part of the project and the team ended up estimating fewer hours on the task than needed. We also finished more hours than was proposed so we had the chance to finish a few **B** prioritized requirements and add more features to the system. For in depth explanations see the project progress report.

3. Thrifter

Thrifter is a unified online marketplace that enables users to buy and sell items. Thrifter utilizes AI algorithms to analyze recent user histories and suggest items for each user. In this section we will go over the different parts of Thrifter. First we talk about the system architecture and how all the different parts of the system communicate. In the last four sections we will give a more general description of the components that were created in the project. First the API we wrote, second the mobile prototype that was created, third the database and fourth the AI we developed for Thrifter.

3.1 System Architecture

We had certain requirements for the system architecture. We wanted the system to be flexible, scalable and have high availability. It was integral that all parts of the system would work well together and that no unnecessary steps were to be added to the architecture to decrease the chance of bottlenecks. The best way to explain the system architecture is to walk through the whole system step by step and introduce each part of it individually.

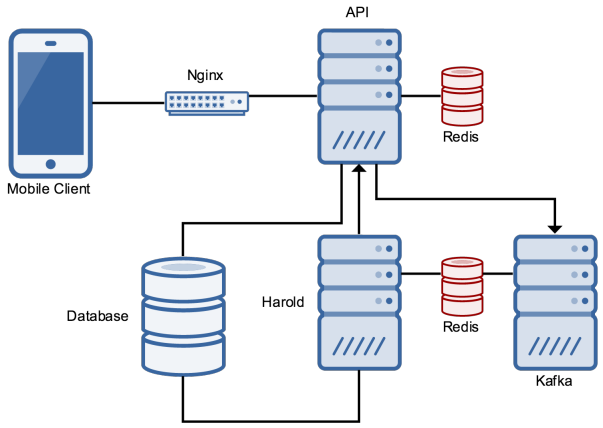


Figure 3.1: In this diagram we see a high level depiction of the whole system architecture.

3.1.1 Client to Proxy

All requests from clients will end up at the proxy and load balancer, Nginx[1] as seen in Figure 3.1. Nginx is a high performance reverse proxy server that can handle multiple concurrent HTTP and HTTPS requests sent to it. All requests that come in on the domain *api.thrifter.io* are handled by Nginx and directed to the correct API service. These services will be described next.

[1] "nginx." 2009. 12 May. 2015 `http://nginx.org/en/`

3.1.2 The Thrifter API and Micro Services

The Thrifter API was written in a microservice architecture. We split our API services into lightweight isolated modules that only perform certain tasks. All our services are Node.js applications so we used the PM2[2] process manager to run all the different service processes as seen in Figure 3.2. PM2 can run multiple instances of a Node.js application called workers. These workers are wrapped inside a cluster by PM2 and then PM2 utilizes a round robin algorithm to balance the load between these workers. One of the biggest upsides we get from using PM2 is its ability to keep processes running forever. When a service is reloaded upon update it will automatically shutdown a process and update it but other processes are kept alive until the update is finished. This gives us zero downtime on services. More information on these services can be found in Chapter 3.2

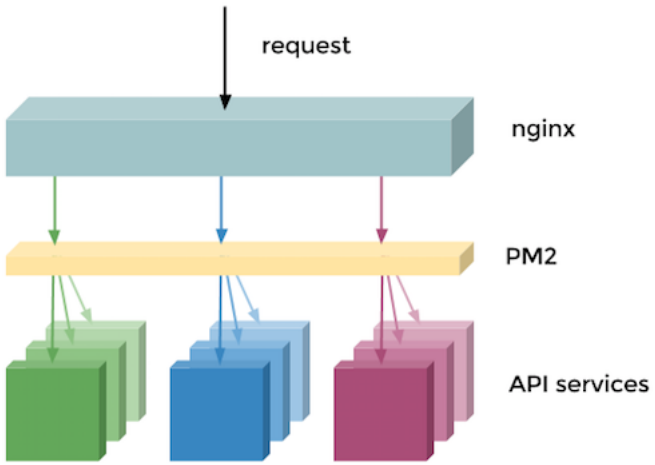


Figure 3.2: This diagram depicts how requests to different routes are delegated by Nginx to PM2 and then directed to the correct service, represented by different colors, and then load balanced by PM2, represented by different shades of the same color.

3.1.3 API and Authentication

Our API was written with RESTful[3] principles in mind so all requests to it are stateless. This works well with our microservice approach since we don't need to worry about states inside services. This, however, opens up a problem with authentication and how to make sure a request is coming from a specific user. The approach we went with was to use Redis[4] as a session store, storing cookies for a specific session. Redis is a key-value store that is capable of storing these sessions as key-value pairs and retrieve them at great speed[5]. A client can then send a cookie, created at login, with every request which can than be compared with the cookie stored in Redis

[2] "Unitech/PM2 · GitHub." 2013. 12 May. 2015 `https://github.com/Unitech/pm2`

[3] Fielding, Roy T, and Richard N Taylor. "Principled design of the modern Web architecture." ACM Transactions on Internet Technology (TOIT) 2.2 (2002): 115-150.

[4] "Redis." 2010. 12 May. 2015 `http://redis.io/`

[5] "Benchmarks - Redis." 2010. 11 May. 2015 `http://redis.io/topics/benchmarks`

to authenticate that the request originates from the correct user.

In future works we would like to update to token-based authentication. We believe that this is a better approach since it would offer many benefits to our whole system architecture. More information on future works can be found in Chapter 6.

3.1.4 Kafka and User History

One of the key features of Thrifter is being able to log and analyze user history. This is a big task and we needed a system that could endure a whole lot of data. We found a messaging system called Kafka[6] to be the most viable choice. Kafka functions in most ways like a typical messaging system but it has a unique design. At the center we have a broker. This broker keeps logs of data and stores it on disk. The data stored is kept in logs defined by topics. As of now we have two topics, *browse* and *buy*. These topics are defined by what activity we are tracking in the system. *Browse* is for every time a user looks at an ad in detail and *buy* is for when a user wants to buy an item. The data is very compact, each message is keyed with the unique identification of the user it corresponds to. The value is a small string in the form of a JSON[7] object, as seen in Figure 3.3, that can be referred to as a *user history object*.

$$\{tagid : int, weight : double, count : int\}$$

Figure 3.3: User history object

This is a depiction of a user history object. It has three fields, *tagid* is a unique id for a tag found in the systems tag tree, the tag tree and tags are used to categories items. *Weight* is the weight given to this tag according to its place in the tag tree. *Count* represents the number of times a user has viewed this tag.

3.1.5 Kafka Consumer and Producer

Kafka requires two other components to work. Firstly we have the producer. A producer sends messages to the broker. For this project we have created a simple Node module that can be imported to a service and used to send messages to a specific topic. Secondly, we have the consumer. The consumer takes the messages sent to the broker and consumes them to suit its purpose. We wrote a simple consumer that consumes new user activity, updates it and then stores it on a Redis server for later use. The consumer can be allowed to consume and update continuously, periodically or when needed. This functionality is demonstrated in Figure 3.4.

3.1.6 Redis and user history

We used Redis as a datastore for the user history. It stores the user history data as a list of user history objects keyed to the user id. Only one user history object is stored for each unique tag

[6] "Apache Kafka." 2012. 12 May. 2015 `http://kafka.apache.org/`

[7] "JSON." 2003. 13 May. 2015 `http://www.json.org/`

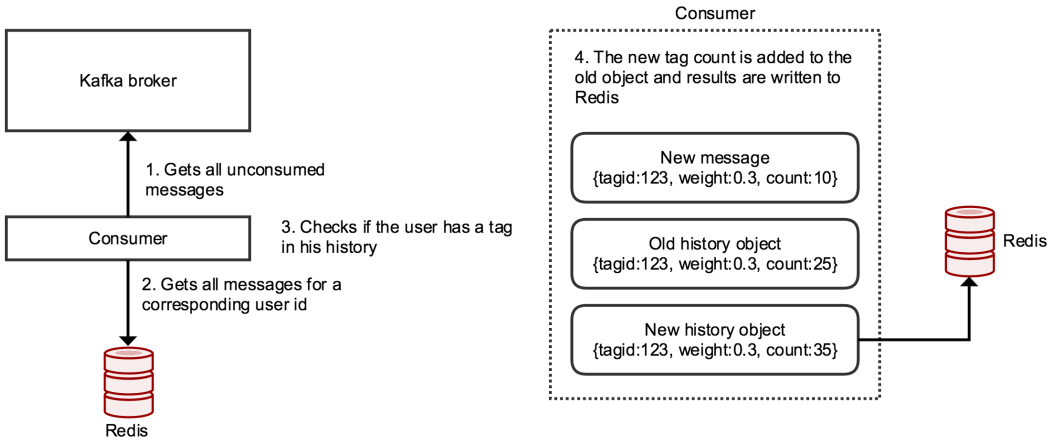


Figure 3.4: This diagram depicts how the consumer gets new data from a Kafka broker and stores it in Redis. In this example, data already exists in Redis for the tag id 123. The old object is combined with the new and stored in Redis.

id in the system. The items in the list of user history object have a *time to live* of seven days This is, however, likely to change as the system grows. If the user history grows too big the least weighted object is discarded for input of newer objects. The user history stored in Redis can then be used by Harold, our recommender system. This will be discussed in more detail in Chapter 4.

3.1.7 Harold and the API

All communications to Harold are on an API level, this serves as a good way to decouple from Harold's implementation. The API only needs to know that it can get a list of advertisements from Harold and these advertisements are recommendations for a user.

3.1.8 PostgreSQL and Data Storage

All data is shared in a single PostgreSQL database instance. Both the API and Harold can access the database. Its implementation is outlined in Chapter 3.3. This implementation was purely justified by the development time we had for this final project. This is a drawback in the sense that microservices, in their purest form, require a dedicated datastore for each and every service[8]. In Chapter 6 we outline how we would like to implement the data storage in the future.

3.1.9 Conclusion

To conclude, we can see that based on our initial requirements of flexibility, scalability and high availability we have made the correct decisions and laid a foundation for a system that can be considered flexible, scalable and available. By breaking the whole system into multiple

[8] (2015). Microservices at Netflix: Lessons for Architectural ... - Nginx. Retrieved May 14, 2015, from `http://nginx.com/blog/microservices-at-netflix-architectural-best-practices/`.

components we get a flexible system that is loosely coupled and has high cohesion. Many parts could be switched out for newer or better ones with minimal effort. The scalability comes from our decision to rely on lightweight, non-blocking systems like Nginx and Node.js to handle concurrent requests. Since these services are lightweight, we can scale our servers with minimal cost and less resources. We can contribute availability mostly to how PM2 handles reloading of services. The largest setback here is the datasource since we are still relying heavily on a single database but this will be improved in future versions.

3.2 Thrifter API

From the beginning of this project, we wanted to write the backend in a microservice architecture. Each service was intended to be responsible for one isolated task in the system. We were not able to fulfill all the requirements needed to follow the microservice architecture in this assignment. We did, however, follow the architecture in some respects and broke the system up into several services. However some services are responsible for more than one task and they all share a database. In this section, we briefly describe each service of the system.

3.2.1 User Service

Everything regarding the user is done in this service. The user service handles user actions such as registration and logging the user in, bookmark functionality, managing contact information and settings. In the future we would like to break this service up into smaller more concise pieces, with a single purpose.

3.2.2 Ad Service

This service enables the user to create ads in the system as well as returning details of specific ads. It also lists the ads for the logged in user. We would also like to break this service up in the future, like the user service. One of the things that the ad service does, that should be broken up into other services, is the search functionality. This service also returns ad-templates based on item categories, allowing the user to create high quality ads. This works by first checking if a template exists for the given category, if it does not, the parent category is checked for a template. This process is repeated until we find a template. In the worst case we end up at the top of our tree with the most basic template. This functionality could also be broken up into a specific service in the future.

3.2.3 Transaction Service

The transaction service is responsible for handling transactions between users in the system. When a buyer has accepted to start a transaction with a seller, this service is responsible for delivering information relevant to that transaction. While this service is relatively small in this first iteration of the system, we plan to expand the functionality further in the future.

3.2.4 Feed Service

This service returns a list of personalized advertisements for each user. It calls Harold, our recommender system and if the user has sufficient amount of user history, Harold will return a list of suggested ads which the feed service will then return. If there is not enough user history, the service returns the five latest ads.

3.2.5 Image Service

A microservice in its purest form. The image service is responsible for only one thing, saving images to an Amazon storage bucket[9] where they can be retrieved for later use.

3.2.6 Notification Service

The notification service is responsible for handling events in the system that affects more than one user. For example when someone bids on an item, the seller is notified via this service so he can respond to the event, which in return will send a notification to that buyer when the seller responds.

3.2.7 Review Service

The review service is responsible for rating the user. As of now, it only supports rating users with a number from one to five and saving the average. Written user-reviews will be a possible add-on in the future.

3.3 PostgreSQL

For our database choice we went with a relational SQL database, PostgreSQL. To simplify development we designed the schema to work on a single shared datasource. In Chapter 6 we talk about how we would like to change this setup to conform better with our use of micro-services. Both the API and Harold have access to the database.

3.4 Mobile client

The mobile client was written in Ionic which utilizes AngularJS, a Javascript MVW (Model View Whatever) framework. The goal was to show the functionality of the underlying system for prototyping as well as testing the usability of the system. We did so in a relatively straight-forward manner. The mobile client uses a sidebar for navigation and a search icon which is always visible. The ad-wizard allows the user to create quality ads by providing templates for items. This feature minimizes the input users have to provide. The mobile client also has a rating functionality. While being able to rate users is a neat feature on it's own, it serves as

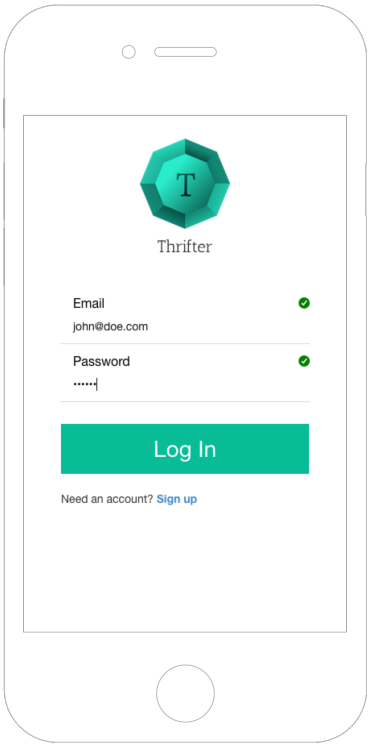
[9] "AWS | Amazon Simple Storage Service (S3) - Online Cloud ..." 2006. 13 May. 2015 `http://aws.amazon.com/s3/`

quality control as well. Users are less likely to trust other users that have received bad ratings and the users will therefore strive to keep up their ratings. The key features of the mobile client are displayed in Figure 3.5

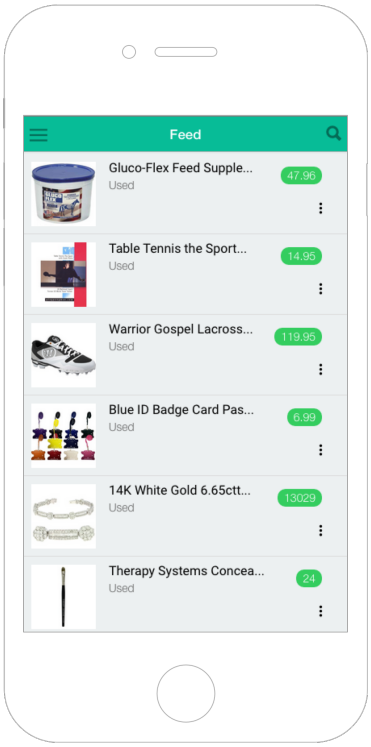
3.5 Harold

Harold was the name we gave our recommender system. Harold was implemented in .Net and C#. Harold retrieves user information from Redis. This information is then used to construct feature vectors. When a feature vector has been constructed it can be used to search clusters of information for relevant data. Tags are then extracted from this information and the database is queried for ads based on those tags. We used a machine learning framework called Accord.Net[10] to help us with the implementation.

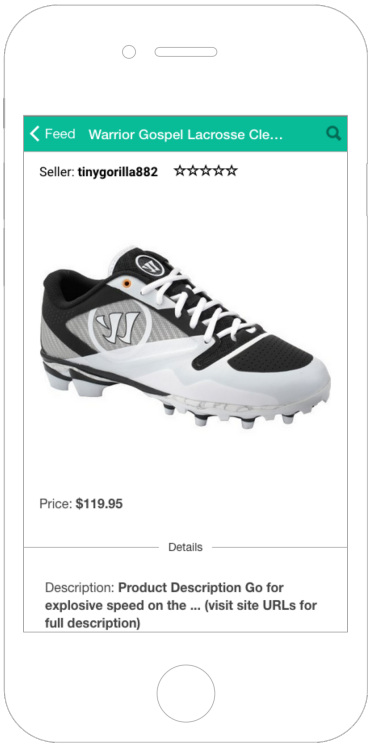
[10]"Accord.NET Machine Learning Framework." 2013. 12 May. 2015 `http://accord-framework.net/`



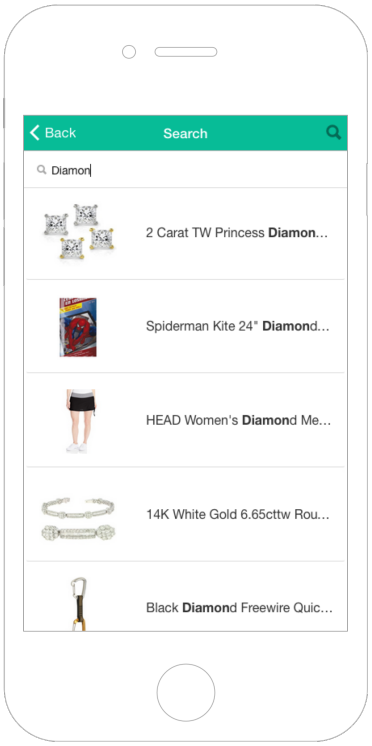
(a) The login screen



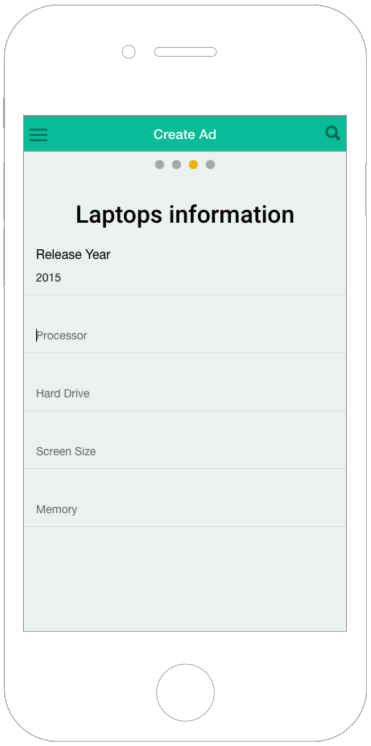
(b) The feed displays ads that are likely to be of interest for the user



(c) The detailed view of an ad shows more information



(d) The search shows a clickable dropdown list



(e) Ad creator with the "Laptops" category selected returns a suitable template

Figure 3.5: Thrifter app screenshots

4. AI Research

In this section we go over the research process that went into making the Harold AI. We knew the AI had to be fast and efficient in search for products as good user experience and satisfiability was desired. Harold uses a Collaborative Filtering method, where data from like-minded users is used to recommend products.

4.1 Data set

The data set we designed to use with our algorithms is an aggregation of the user's history and his interaction with different items. What it comes down to is basically a histogram of tag ids, each item has a number of different tags assigned to it and these tags are recorded down when a user interacts with that item e.g. if a user looks at an iPhone 6, what gets logged are the tags *Mobile Phones, Smartphones, Telephony, Communications, Electronics*. These tags are then assigned different weights based on how descriptive they are e.g. the *Smartphones* tag is very descriptive while the *Electronics tag* isn't so it has less weight. The taxonomy we used comes from Google[1].
Each entry in the data set is represented as point in n-dimensional space and the distance between these points dictate their likeness as seen in Figure 4.1.
The method we used when calculating the time of tests on the data set is called *Total Time* and is defined as follows:
$TotalTime = FindingCluster + FileReadingTime + TrainingTime + ClassificationTime$

4.2 k-NN

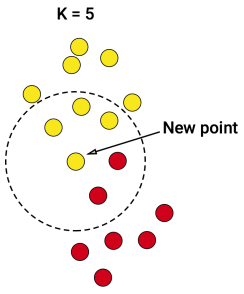


Figure 4.1: k-NN nearest neighbour search using k = 5

To search for points in a big data set we applied the k-Nearest Neighbour algorithm. k-NN searches for the k closest points in a data set as is displayed in Figure 4.1.

[1] "Categorize your products - Google Merchant Center Help." 2013. 13 May. 2015 `https://support.google.com/merchants/answer/160081?hl=en`

We began by testing the k-NN algorithm on our data set without any clustering. We soon saw that this approach was not going to work. As we were coming close to 100.000 points in our data set the total time for our solution was about to exceed the one second mark. These were not promising results since we wanted to be able to handle millions of points. Since k-NN scales linearly we saw that k-NN alone was not going to be efficient enough. The total time for k-NN on a data set of size one million was taking up to 20 seconds.
To try and reduce the total time for our k-NN search we decided to apply clustering algorithms on our data set. These algorithms split the data set into k spaces that are called clusters.

4.3 Accuracy

When measuring the functionality of AI systems you have to have some standard, to which you can measure your new results against. In this research, where distance is the metric we use to define points, the golden standard is running the whole data set against k-NN. The k-NN algorithm will always return the k-nearest neighbours given the whole data set. This accuracy will suffer when we start clustering the data set. This happens when neighbours at the edges of clusters fall into different clusters. All accuracy measurements presented in the rest of this chapter are a comparison of the points we get from running only k-NN versus running k-NN on the clustered data set.

4.4 K-Means

K-Means works by clustering our data set based on distance. A centroid is chosen at random for each of the k clusters, then the remaining points in the data set are placed around these centroids based on distance. These centroids are then recalculated for these new clusters and the points are placed around them again. These steps are repeated until the centroids remain the same between iterations.

There are number of different version of this algorithm and we looked at some of them and measured the difference between them to see which algorithm would benefit us the most.

4.4.1 K-Means++

K-Means++ is a variation of the K-Means algorithm. The main difference is how the centroids are chosen. The K-Means algorithm chooses the initial centroid randomly but K-Means++ uses a seeding method to choose the initial centroids. One centroid is chosen at random and the other centroids are then chosen using a weighted probability distribution based on the distance from the initial centroid. Using this method we can reduce the number of iterations the algorithm has to go through to achieve convergence.

4.4.2 K-Modes

The second version of the algorithm we looked at was the K-Modes method. With K-Means the centroids do not necessarily represent actual points in the data set. K-Modes uses actual

data points as centroids (medoids). This can lead to better robustness against outliers but can introduce more complexity because of the time spent on finding the actual data point rather than just finding the mean distance like in K-Means.

4.4.3 Binary Splits

The third and last algorithm we looked at was Binary Splits. The algorithm works by first splitting the data set into two clusters using K-Means, then the more sparse cluster of the two is split again with K-Means. This process is repeated until we have reached our clustering goal. This can give us good clustering times as K-Means is only run with $k = 2$.

4.5 Speed Testing

The next step in the research was to test the different versions of K-Means and see if they could lower our total time from 27 seconds to something closer to one second. We ran the algorithms on the same data set of one million points.

4.6 Clustering

We began by splitting the data set into two clusters and comparing the results to our earlier findings from the k-NN search. Immediately, we saw drastic change where total time went from 27 seconds to 11 seconds. After this we broke the data set into more and more clusters until we reached a point where the sizes of the clusters weren't the problem but the amount of clusters were starting to become the issue.

In Table 4.1 we can see how our clustering method paid off. The more we clustered the data set the better the total time. The reason for this is that k-NN has to search fewer points when the clusters become smaller.

No.Clusters	K-Means	K-Means++	K-Modes	BinarySplits
2	12039	11258	11658	10755
3	6351	6134	7752	6514
5	4086	4396	4126	4172
10	2389	2581	2748	2146
20	1147	971	1749	1008
50	704	430	819	582
100	317	358	494	351

Table 4.1: Cluster Time in milliseconds for our algorithms

4.7 Distance Calculations

To calculate the distances between points in space we decided to use Euclidean distance. We measured two variations of this method, standard Euclidean distance and Squared Euclidean distance. What the Euclidean distance allows us to do is to find the distance between the points in our dataset.

$$d(p,q) = \sqrt{\sum_{i=1}^{n}(q_i - p_i)^2} \tag{4.1}$$

In formula 4.1 we can see the Euclidean distance formula. The biggest flaw with using this equation in modern computers is the square root. Doing millions of calculations with a square root becomes very expensive.

The Squared Euclidean distance is very similar to the standard Euclidean distance except we eliminate the square root. By doing this we might not get the shortest distance between the points but the ratio between them stays the same allowing us to use it as a reliable measurement.

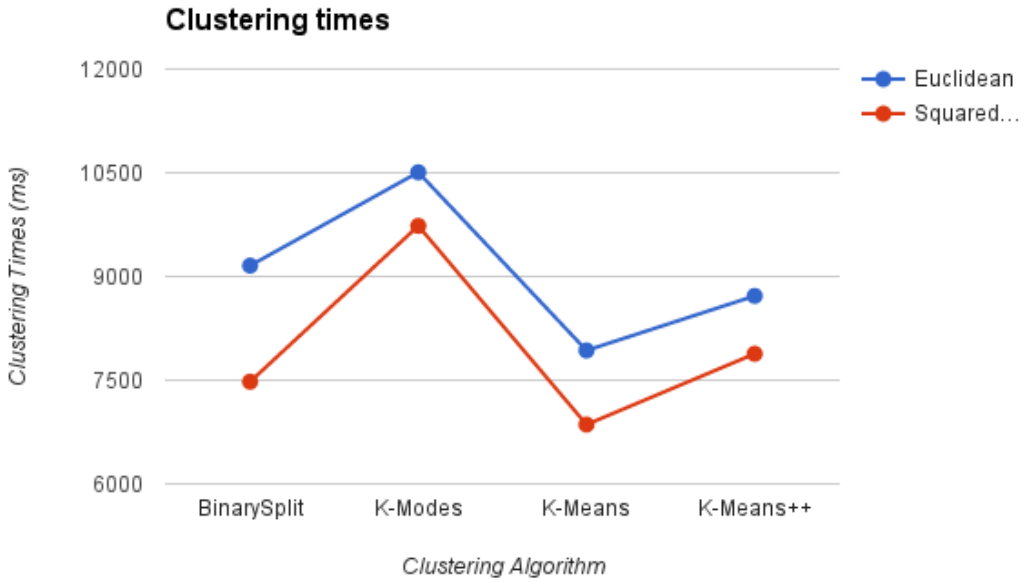


Figure 4.2: Clustering Time with and without square root

Figure 4.2 is taken from the Research Report and shows us how the two formulas affect the clustering time for our algorithms. Since the Squared Euclidean distance gives us better clustering time without any drawbacks, we decided to use it in the remainder of the research.

4.8 Accuracy Improvement

After the previous improvement we saw that the accuracy of our algorithms was dropping fast. The reason for this accuracy loss was because the k-NN algorithm was simply not getting all of the right points in the clusters we found for a specific user. When we cluster our data set into so many clusters, some of the points can end up in different clusters and k-NN will not find all of the correct points given a single cluster.

This was not ideal since we had set a standard of 90% accuracy in the beginning of the research. Since the speed of our algorithms was heading in a good direction with all versions of K-Means running under 500ms, we put all our effort into fixing the accuracy loss.

4.8.1 Margin Inclusion

To fix this accuracy loss we came up with a method we like to call Margin Inclusion. What Margin Inclusion does is simply picking more than one cluster into k-NN if our point is within some range from more than one cluster in the data set.

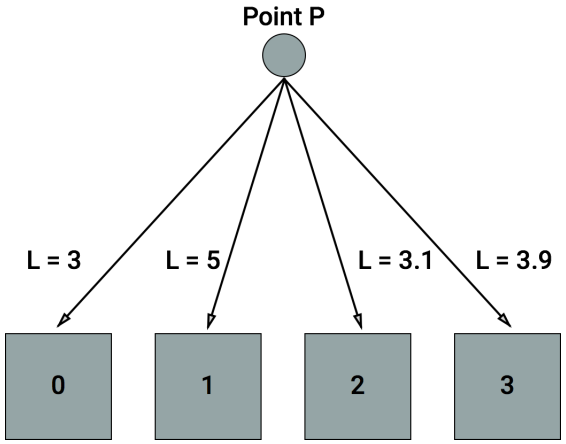


Figure 4.3: Margin Inclusion

If we take a look at Figure 4.3 we can see that we have to decide which cluster is relevant based on the distance between point P and the centroid of each of the clusters. We see that cluster 0 and cluster 2 have almost identical distance but if the algorithm has no Margin Inclusion we have to go down to cluster 0 since it has the shortest distance. But what if some of the points we were looking for are actually in cluster 2? Since there is such a small difference in distance there is a possibility that a point we are looking for ended up in cluster 2. If we apply the Margin Inclusion method to the same search with some distance fluctuation limit, for example 10%, then we can both search in cluster 0 and cluster 2 since the difference between these distances falls under the limit we set for 10%.

This improvement method came with a cost. With Margin Inclusion our accuracy was back within the accuracy range we wanted and somewhat higher than that. But when we have to

search in more than one cluster our total time rises again because we have to run the kNN algorithm on a bigger input set.

In Table 4.2 we can see an example of how Margin Inclusion affects our total time for the Binary Splits algorithm.

	With Margin Inclusion	Without Margin Inclusion
20 clusters	903 ms	730 ms
100 clusters	504 ms	351 ms

Table 4.2: Effects of Margin Inclusion on Binary Splits

We now had to figure out a way to get our total time down again. As seen in Table 4.2 we are losing roughly 150-170 milliseconds by applying the Margin Inclusion method.

4.9 Improved Speed

What we did next is called layering. What layering means is that another layer of clusters is added above our current clusters, which can also be described as clustering the clusters.

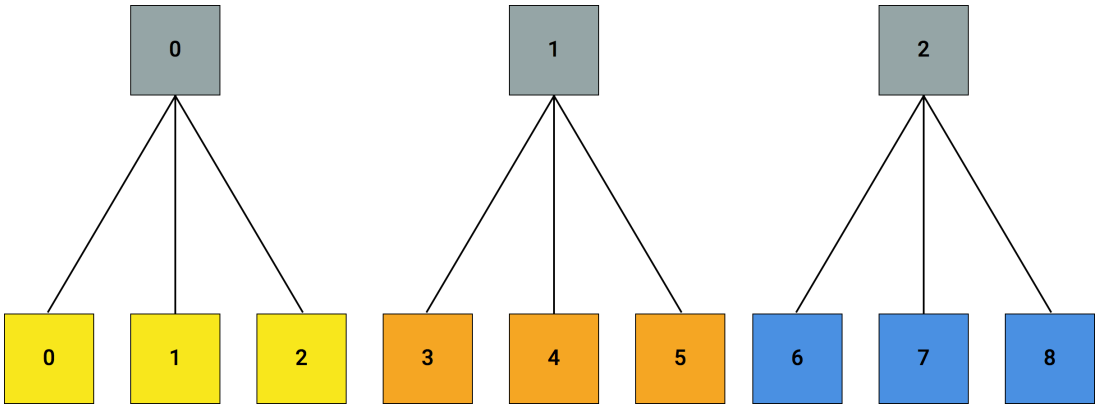


Figure 4.4: Applying layers to the clusters

Figure 4.4 describes layering in its basic form. By applying the layering method we eliminate the need to compare distances to all the subclusters. For example, instead of comparing the distance to 9 clusters we would only have to compare distance to the top 3 clusters and then only check the subclusters of the closest layer cluster. If cluster 0 has the shortest distance from the point, given that Margin Inclusion will not play a role here, we will only need to compare distances to the clusters that fall under that cluster, in this example cluster 0, 1 and 2.

By applying this method on top of Margin Inclusion we now both have better total time and a very good accuracy for our algorithms.

4.10 Results

When comparing the first and the last iterations for our algorithms we can see a drastic change as is shown in Table 4.3. The total time for all methods has dropped from averaging around 11 seconds to around 140 to 300 milliseconds depending on the chosen algorithm.

	K-Means	K-Means++	BinarySplits
First Iteration	11369	11681	11615
Last Iteration	311	164	146

Table 4.3: Total Time in milliseconds from first to last iteration

These numbers show that our research was a success and we now have a very good total time. For a data set of size 1 million we do not have to worry about which algorithm we choose since they all meet the standard that we set in the beginning.
The Binary Splits algorithm has the advantage of scaling much better than K-Means and K-Means++. We ran a clustering test on a data set of 10 million for all three algorithms.

	K-Means	K-Means++	BinarySplits
Clustering Time	19 minutes	22 hours	42 hours

Table 4.4: Clustering Time for dataset of 10 million

When looking at these clustering times in Table 4.4 it is fairly obvious that Binary Splits is much faster. Although it does not share as high of an accuracy as the other two algorithms, we can cluster our data set much more often to make up for that accuracy loss and always have the most relevant data ready.

5. Key Learnings

Although we went into this project with what we thought was a pretty good idea of how to do things, we quickly learned that was not the case. Some things worked out, other did not. The following is what we can take from the experience of doing this project.

5.1 Microservices

As previously stated in the system architecture chapter, we decided to use a microservice design for our system. In the beginning, we chose to use a NodeJS framework called Loopback for our backend (excluding Harold). After writing the bulk of our API, we felt that Loopback was really not intended to be used in a microservice architecture but rather as a single monolith application as it's trying to be an all-in-one package. This was the moment we realized that we made the right decision in the beginning when we chose to use microservices because this allowed us to simply write the remaining services in other frameworks. As an example, we went from ~63MB with the User-Service (written in Loopback) to ~7MB with the Review-Service (written in vanilla Express).

5.2 Ionic

We went into the development of the mobile client with the thought in mind that we might be able to continue the work after the project, ending with a fully fleshed out product. We quickly learned that while the idea behind Ionic is good, the application produced simply isn't up to par with apps that are written in a language native to the phone. As of this writing, we believe that frameworks such as Ionic are not suitable for writing production quality applications of this scale and we will most likely write a new client applications in a native language in the future. Ionic served us well as a prototyping tool and we have a much clearer idea of how to move forward with a native app based on what we learned by using Ionic.

5.3 Scrum

The decision to use the Scrum methodology was one of the first decisions we agreed on. Scrum seemed to be the perfect fit for the development of Thrifter, due in particular, to Scrum's adaption to changes made to a project and the visual representation of the project's progress. However, the team began having doubts about some aspects of Scrum, shortly after initial development of Thrifter began.

There were two aspects of Scrum we did not see value in for this type of project, namely the sprint retrospective meetings and the daily standup meetings. The reason for this lies in the team's infrastructure. The team consists of five people that work on the project in a small office space. The team was very verbal about what each member was working on at any given time.

Because of this, when retrospective meetings or daily standup meetings were held, the team shared a common understanding and knowledge of the status of the project, rendering them redundant for this project.

5.4 AI

We knew going into this project that the AI functionality was going to be a big unknown. We were never sure if it was going to be successful or not. After going through the process of building an AI system we have learned a few things. First, building AI systems is a lot of hard work with countless hours spent doing research and getting to know the current methods and standards. We chose .Net and C# to implement our system. Before going into it we weren't sure if this choice would come back and bite us. Fortunately the .Net framework and the machine learning framework we chose to help us proved to be very handy and easy to use compared to the complexity of the subject matter.

6. Future works

This final project was the first milestone of the Thrifter project. The team has a shared vision of the future of the project and we will continue working on it after this initial phase. In this section we cover some of the ideas the team envisions as the future of Thrifter and how the team proposes to be able to work towards these future ideas.

6.1 Harold

In Chapter 4.6 we discussed a technique we called Margin Inclusion, this method allowed us to get better accuracy from the AI system.

Margin Inclusion has a few downsides, one being the choice of margin. There are a few caveats with this number, if the number is too high for the specific data set it can increase the run times without helping the accuracy. This could be solved by some kind of halting when we reach some cluster limit.

Another problem with Margin Inclusion is that when you recluster you are not guaranteed the same clusters again so a margin that worked with one set of clusters is not guaranteed to work with the new clusters.

We would like to set up a dynamic adjustment of this margin based on some measurements, a sort of reinforcement learning based on both user evaluation and run time metrics.

One thing that we haven't discussed is that the length of the feature vector was not important to the running time measurements. While this is true, the length of the feature vector is still an important metric when it comes to the usability of the system. This problem is only something that can be evaluated by logging usage statistics of the system, things like click and buy rates of recommended ads is something we need to monitor to adjust our system for a better user experience.

6.2 Backend

We like the microservice architecture, it gives us power and the flexibility to scale our system so it can handle more stress. As it stands now, the system is not in a pure microservice form as a couple of our services are responsible for doing too many things. What we want to do in the future, is to rewrite our Loopback services in Express or other languages, as real microservices where each service is generally only responsible for one task. Each service should also have a dedicated database and use a messaging system like Kafka for inter-communication. This way, we end up with a fast, decoupled and scalable backend.

6.2.1 Token Based Authentication

When we started developing the system we went with the more traditional cookie based authentication. This worked well with our prototype app since it was built with the Ionic framework which is essentially a mobile application wrapped in a browser and browsers play well with cookies. For our future works we would like to transition to a more modern token based authentication. This would benefit us since it is more attuned with the system as it has developed over the course of this project. Cookies and CORS (Cross-origin resource sharing) don't play well together across different domains. Since we are also developing a mobile client and we are looking to deliver the Thrifter platform as a Software as a Service (SaaS), token based authentication becomes a more viable option when handling requests across multiple domains. Token based authentication is also stateless so the need for a session store becomes obsolete, cutting down on server costs and increasing scalability.

6.2.2 Gathering of User History

It quickly became clear that we would only be able to gather and process a small amount of user data. We set a scope for this project to only measure what a user browsed for or bought. In future works we would like to log more user data such as what a user searches for but also smooth out our current logging. By this we mean that e.g. when a user is browsing, we can give different weights to tags depending on the time spent looking at an ad. So we would like to gather more accurate data.

6.3 Frontend Client

In the process of developing Thrifter our main focus was set on Harold and the backend. We constructed a prototype of a smartphone app in order to demonstrate the functionality of the underlying system. That, however, was the sole purpose of the client app and therefore it was not the biggest focus point of this final project. In the next stages of developing Thrifter, the team intends to create a high quality and production ready mobile client. In this stage of development, we used the Ionic framework to construct the prototype client. Ionic was chosen for it's simplicity and rapid development time. However, it did not return the performance we were looking for in a production ready client. We would therefore like to rewrite the client entirely, preferably as a native application.

6.4 Software as a Service (SaaS)

The idea of building Thrifter in the SaaS delivery model was born in the early stages of designing Thrifter. This idea was always meant to serve as a long term goal for Thrifter and was out of scope for this final project. By building Thrifter as a service, we would enable anyone to subscribe to the Thrifter platform and use Thrifters transaction abilities in their own transaction solution. By offering this service, Thrifter would allow anyone who is interested in creating an online web store to do so without having to set up complicated payment and transaction

services. This would also serve as an important revenue stream, enabling the us to develop Thrifter into a more complete solution.

7. Conclusion

In this report we discussed the Thrifter project. After the introduction we talk about the organization of this project and how we worked over the course of this project. Next we talk about the architecture and how we managed to build a scalable system that is loosely coupled and available. We then go into detail about the research that went into Harold, our recommender system. We also discuss the key learnings we as a team take away from this project and what we can take with us into the future for Thrifter. Next we discuss what the second iteration of Thrifter has in store and where we see the system in the future.

7.1 Acknowledgements

We wish to acknowledge our instructor in this final project, Hlynur Sigurþórsson, for invaluable assistance and guidance through this first iteration of Thrifter.

Reykjavík University
Menntavegur 1
101 Reykjavík, Iceland
Tel. +354 599 6200
Fax +354 599 6201
www.ru.is