UNIVERSITY OF SOUTHERN DENMARK

SOFTWARE ENGINEERING 6. SEMESTER

Datamining and its use

Author:
Lasse Bjørn HANSEN
Simon FLENSTED

Supervisor:
Jan Corfixen Sørensen SØRENSEN

A report submitted in fulfillment of the requirements
of Software Engineering 6. semester

at

University of Southern Denmark
TEK

May 10, 2017

"Some quote"

- Gruppe 3

Contents

Chapter 1

Introduction

1.1 Motivation

The amount of data being processed around the Internet and within big systems is continuously increasing. This data should be structured and modelled in a way that makes it easily accessible and easy to work with. Handling large amounts of data the right way can prove to be very useful, not only to the company who possess the data, but also to the end users of a product. To achieve this, the art of data mining is very useful. The company Struct A/S [4] has provided us with a software engineering task of creating product recommendations where data mining will create the foundation. This report will address the use of data mining, how to develop a solution that provides the user with intelligent product recommendations, and makes it possible to maintain current and future data. [1]

1.1.1 Data mining

Data mining has become a big part of modern software engineering. Lots of companies tends to store large amount of data. If the data is analyzed properly and put into use, it can create tremendous value to the company as well as its users. In this case Struct has stored information about users visiting their websites. Previously, this data was stored in an unstructured database and not put to use. By processing the data properly, using data mining, it can be structured in a way that makes it useful to the company e.g. product recommendations.

1.1.2 Product recommendation

If an e-commerce company wants to increase its profit, product recommendation has proven to be very beneficial.[5] This is heavily used by multiple companies including the retail giant Amazon.[6] If you can predict what sorts of products your costumer may find useful, additional sales becomes more frequent. Big data sets, like the one provided by Struct A/S, can make it possible to predict customer needs, if the data is processed properly.

Chapter 2

Problem statement

2.1   Problem description

The initial problem/challenge is given to us by the company Struct A/S and is described as follows:

When launching sites, whether it being regular websites or web shops, a lot of user activity is logged. We therefore have a large amount of data associated with each of our sites but do not currently use it.
In the future we would like to be able to use logged data to generate an insight into the user activity on our site and actively use this data to create a personalized experience for the users.

This project handles the initial analysis of the data, storing it in a scalable way and utilizing the data to create features which add value to the company. The focus of the project is data storing, data mining and recommendation algorithms. These methods are used to implement a final software solution capable of storing, organizing and utilizing current as well as new data about the end users. This allows Struct to easily keep their data updated and receive tailored product recommendations for their users.

2.2   Problem statement

The data we were given is in an unstructured format and can not be put to use as it is. This leads to the following problems - structuring and utilizing the data to create a personalized experience for the users, and making the data easily maintainable.
This leads to the following research questions:

- How can you optimally organize, store and access data in a scalable way?

- How can this data be maintained and updated easily after deployment?

- How can you utilize the organized data to generate tailored product recommendations for the end user?

Chapter 3

Requirements

3.1 Requirements engineering

The requirements of the project are categorized into functional and non-functional requirements. These requirements were derived from the original case given by Struct A/S (Appendix ??), continuously planned meetings with Struct A/S and our supervisor, and as a part of the constant research done during the progress of the project.
The functionality of the final product is developed in order to fulfill the most important aspects of the case, and the requirements derived from the client meetings.

3.1.1 Functional requirements

The most important features of the system includes delivery of good quality product recommendation and handling of new data. These are very complex features and a lot of requirements must be fulfilled in, order to realize them. The most crucial functional requirements can be seen in table 3.1. These requirements have been the driving force throughout the project. For a complete list of functional requirements, see the git backlog [?].

As we can see from table 3.1, the functional requirements of the final product can be compressed into 12 requirements. This corresponds with the wish of a simple API, that provides good quality product recommendations. A lot of work has therefore been put into developing a complex and reliable algorithm that provides state of the art product recommendations. F01 was the most important requirement and has therefore acted as an ongoing task during the entire development of the product. F02-F06 was secondary requirements as they were not crucial before the recommendation algorithm was implemented. However, once the algorithm was applied to the system, requirement F02-F12 was needed in order to keep the data updated.

3.1.2 Non-functional requirements

The non-functional requirements were described at an early stage, and later clarified at the planned meetings. The functional requirements can be seen in table 3.2.

The non-functional requirements are few, but turns out to be very challenging. NF01 was a choice made based on the fact that the platform Struct is developing on is based on C#. .Net core was chosen because of its high performance and scalable systems, which was needed to realize NF02.[8] NF02 was probably the most challenging requirement to fulfill, however very important since you do not want your website to be slow. At the beginning Struct had a wish that the new database for recommendation data had to be scalable up to billions of records. Because of the denormalized data structure, it was agreed that a No-SQL database would be the right approach. This resulted in requirement NF03. In order to lower the amount of effort needed to integrate the recommendation system, the API had to easily accessible and the data output had to be in a standardized format. This resulted in requirement NF04.

F01	The webshop developer can provide tailored product recommendations to his customers. When the API is provided with information about a visitor, tailored recommendations to the customer must be returned. If the data about the visitor is insufficient to calculate enough tailored recommendations, the most popular products within the last 30 days must be used to present enough recommendations.
F02	The webshop developer can store new behavior data for a visitor in the database. When the API is provided with the required information, new behavior data must be stored in the database.
F03	The webshop developer can store new behavior data for a product in the database. When the API is provided with the required information, new behavior data must be stored in the database.
F04	The webshop developer can store new product groups in the database. When the API is provided with the required information, a new product group must be stored in the database.
F05	The webshop developer can store new visitors in the database. When the API is provided with the required information, a new visitor must be stored in the database.
F06	The webshop developer can store new products in the database. When the API is provided with the required information, a new product must be stored in the database.
F07	The webshop developer can update existing product groups in the database. When the API is provided with the required information, a product group should be updated.
F08	The webshop developer can update existing products in the database. When the API is provided with the required information, a product should be updated.
F09	The webshop developer can update a visitor in the database. When the API is provided with the required information, the visitor should be updated.
F10	The webshop developer can delete existing behavior in the database. When the API is provided with the required information, behavior data should be deleted in order to keep the data up-to-date.
F11	The webshop developer can delete an existing visitor in the database. When the API is provided with the required information, the visitor should be deleted in order to keep the data up-to-date.
F12	The webshop developer can delete an existing product group in the database. When the API is provided with the required information, the product group should be deleted in order to keep the data up-to-date.

TABLE 3.1: Functional requirements

NF01	The API should be developed in C# .NET core.
NF02	Recommendations must be delivered within 40ms.
NF03	The data used for product recommendations should be stored in a fitting scalable No-SQL database.
NF04	The API should be easy accessible through a web service.

TABLE 3.2: Non-functional requirements

Chapter 4

Design

4.1 Conceptual overview of the system

The system is developed as a part of the classic architectural pattern, Model-View-Controller (MVC).(find kilde, evt. wiki) The system itself consists of the Model and Controller part, and lets the client be responsible for the view. In figure 4.1 it is shown how the system is layered. As mentioned, the client is responsible of the view, which in this case is the webshop. Data is sent to the API (Controller) which simply communicates the data to the logic (model) of the system. The logic (model) is responsible for communication with the database, calculations regarding product recommendations, and handling of new incoming data.

4.1.1 Recommendation API (Controller)

The API is split into two Controller-classes, DataController and RecommendationController. If we take a look at figure 4.2, the two controller-classes can be seen. The RecommendationController only consists of one API-call, however this is probably the most interesting call that can be made. Calling the GetRecommendationForVisitor-method with a visitor UID and the desired number of product recommendations will return a String-array consisting of product IDs, which the client can then use to present products in a desirable way to the end-user.

The DataController is used for keeping the data in the database up-to-date. The three Put-methods allows the client to provide new visitor and behavior data to the database. The GetUpdate allow the client to initiate the build of the collaborative filter. Collaborative filtering will be discussed more thoroughly in **??**. GetUpdateVisitorTopProducts() allows the client to initiate the calculations of each visitors most viewed products. At last, CalculateTop20() lets the client initiate the calculations of the most popular products in the last 30 days. All the update-methods is special administrator methods, and should only be accessible to few access-approved staff.
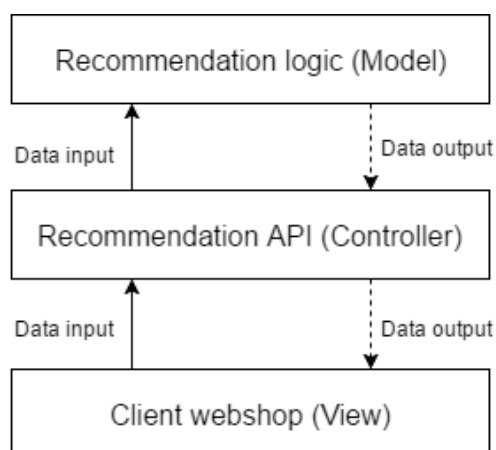

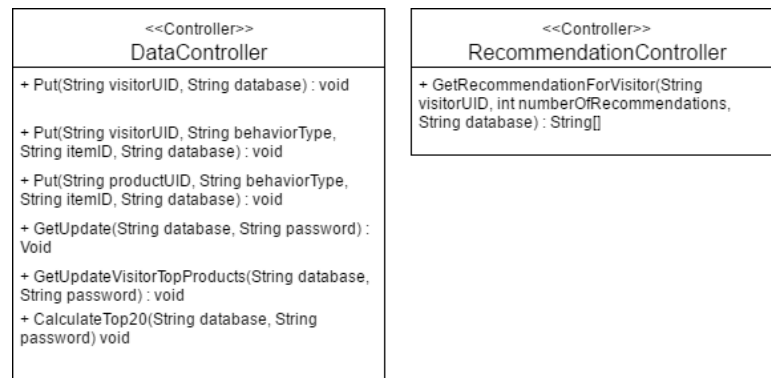FIGURE 4.1: The Model-View-Controller (MVC) pattern applied on the system

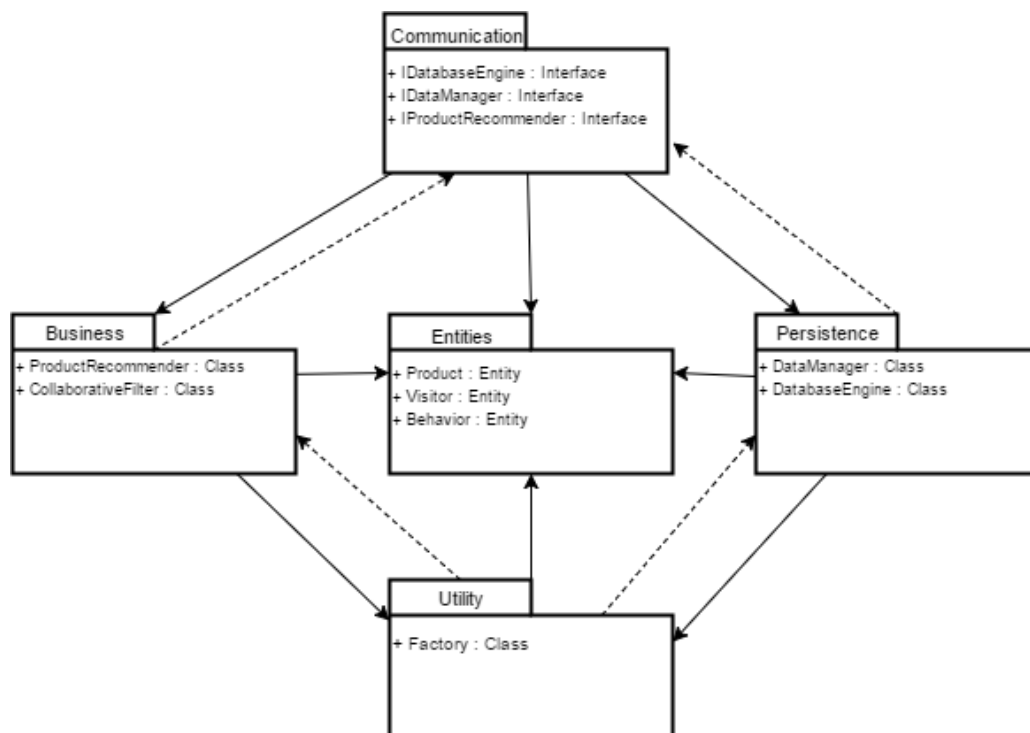FIGURE 4.2: The two controller-classes in the recommendation API.

FIGURE 4.3: Package diagram of the model layer

4.1.2 Recommendation logic (Model)

The recommendation logic is where the main operations of the system takes place. The model layer consists of five packages, and is made accessible to the controller layer through three interfaces. A package diagram of the layer can be seen in figure 4.3. All classes used for calculations are placed in the Business package. This is where the product recommendations are calculated before being sent back to the control layer. This is also the package where any offline-calculation is made before it is stored in the database. The persistence package handles all information that needs to be communicated with the database. The Entities and Utility package creates an easier and more manageable way of communicating data around within the model layer. All communication between the Controller-layer and the Model-layer is done through the interfaces seen in the Communication package. These interfaces are implemented by their corresponding classes in the Business and Persistence packages. The implementation of the Model-layer is discussed further in the **??**.
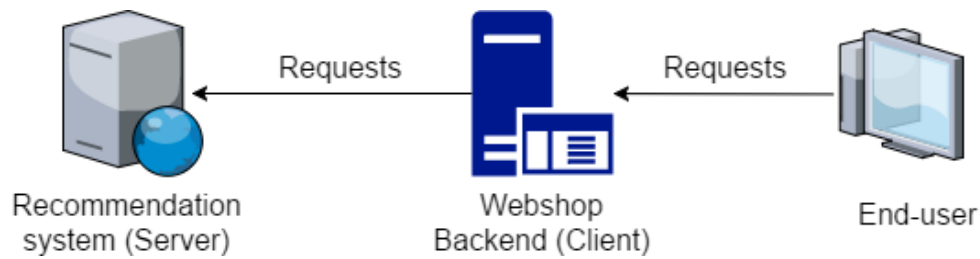
FIGURE 4.4: Package diagram of the model layer

4.2 Client-Server

When put to use, the recommendation system will be distributed and play the server role in a Client-Server model. The system should be considered an application solely for providing product recommendations. This means we have a very thin client, as its only job is to ask for recommendations by sending very little information. In this scenario, the client is the webshop that needs to provide recommendations to one of its users. The client is also able to ask the server to update its database or store new content in the database, but the concept is the same and just as simple as the request for product recommendations. The Client-Server model of the system can be seen in figure 4.4

4.3 Database design

One of the challenges of the project was to come up with a database design that would meet the requirements of the final product.

Chapter 5

Implementation

5.1 Solution overview

The final solution consists of a *RESTful* API build with ASP.NET Core and a MongoDB *No-SQL* database. Both are hosted through Amazon Webservices in an EC2 container using Docker.
The API consists of the commands seen in appendix A.

This section covers the process from the initial data delivery to the final solution in a chronological order.

5.2 Initial data dump

In the beginning of the project we were supplied with data from one of *Struct A/S'* customers. This data was in the form of many SQL tables and most of the data was not relevant for creating product recommendations. The main tables used were the following:

- Visitor: A collection of every unique visitor who had visited the customer's website, each visitor gets a unique identifier called UID. Contains 3,073,665 visitors.

- Profile: A collection of every users signed up at the website. Contains 3037 profiles.

- BehaviorData: A collection of every unique action performed by visitors on the website, for example when a visitor views a product a new row is made with the visitor's UID, the product UID and the timestamp. Contains 3,326,736 visitor actions.

- Order: Contains each profile's orders. Contains 5520 orders.

- Product: A collection of all products on the website with their unique IDs. Contains 22,445 products.

- ProductGroup: A collection of all product groups. Contains 262 product groups.

- AttributeValueRendered: A collection of product and product group descriptions in different languages. Contains 409,259 descriptions:

A snapshot of the visitor and behaviorData table can be seen in figure 5.1 and 5.2.

	Uid	UserAgent	BrowserName	BrowserVersion	IPaddress
1	6820EDD0-E6A6-4105-A078-0000127E7AE1	AdsBot-Google (+http://www.google.com/adsbot.html)	Unknown	0.0	66.249.●.1●
2	4D9FC023-9034-4B93-96D8-000013AB1940	Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.3...	Chrome	36.0	212.71.●●.●●
3	CC43266B-5E13-473E-A4F2-000019D2B065	Mozilla/5.0 (compatible; Googlebot/2.1; +http://ww...	Mozilla	0.0	66.249.●.●●

FIGURE 5.1: Visitor table from the original data

	Type	Id	UserId	Timestamp	ActivityData
1	ProductView	31070	794ebe68-49c5-451c-ba80-226cdc0508f4	2016-09-30 20:04:01.5376346 +02:00	NULL
2	ProductView	31071	055bf3eb-a556-4672-9f65-5f64932e3973	2017-01-05 12:18:05.5065939 +01:00	NULL
3	ProductView	31071	06d3f1b4-8347-499d-9edd-71c058024cc1	2017-01-27 10:42:00.8031017 +01:00	NULL

FIGURE 5.2: Behavior data table from the original data

These tables contain all the pertinent information for creating product recommendations and can be utilized after a cleaning and structuring process. This process is described in the following sections.

5.3 Data Transformation

To begin the initial data transformation a data storing technology has to be selected. The technology chosen was *No-SQL*, specifically *MongoDB* the most popular No-SQL framework [7].
No-SQL is chosen because of the good fit for this project. The data demands are not clearly specified in the beginning and with No-SQL it is easy to add or remove data or even change the data types on the fly. No-SQL's denormalized format also allows for faster retrieval of a single item without having to do joins or complex SQL queries. Finally No-SQL is easier to scale across multiple servers and many engines have built in scaling functionalities [3] which can come in handy when multiple clients begin using the service.

A brief overview of the different terminology for SQL and No-SQL is given in table 5.1.

TABLE 5.1: SQL vs No-SQL terminology

SQL	No-SQL	Comment
Table	Collection	
Row	Document	A No-SQL document can contain more complex datatype compared to a row in SQL e.g arrays or other documents

Python was used to accomplish the early migration from SQL tables to MongoDB. Several scripts were created to retrieve the data from the SQL server and transfer it to the MongoDB database in the wanted format. Pseudo code of one of these scripts can be seen in algorithm 1.

After all the scripts are finished the No-SQL database has the collections Visitor and Product. A breakdown of documents in the two collections can be seen in table 5.2

Algorithm 1 Product Script

```
 1: SQLquery = SELECT * FROM struct.Product
 2: db = MongoDB
 3: for all Rows r in SQLquery do
 4:     Product = {Id: r.id
 5:            Description: ""
 6:            Created: r.created
 7:            visitorID: []
 8:            ProductGroupId: 0 }
 9:     db.insert(Product)
10: end for
11:
12: Description = ""
13: firstId = true
14: previousId = 0
15: previousGroupId = 0
16: SQLquery = SELECT * FROM struct.product JOIN struct.attributeValueRendered ON id ORDER
    BY productId desc
17: for all Rows r in SQLquery do
18:     currentId = r.ProductId
19:     currentGroupId = r.GroupId
20:     if firstId then
21:         previousId = currentId
22:         firstId = false
23:     end if
24:     if currentId != previousId then
25:         db.update({id: previousId},
26:             $set: db.description: description
27:         db.update({id: previousId},
28:             $set: db.ProductGroupId: previousGroupId
29:         description = ""
30:     end if
31:     description += r.description
32:     previousId = currentId
33:     previousgroupId = currentGroupId
34: end for
35: VisitorIds = []
36: firstId = true
37: previousId = 0
38: SQLquery = SELECT * FROM struct.BehaviorData
39: for all Rows r in SQLquery do
40:     currentId = r.Id
41:     if firstId then
42:         previousId = currentId
43:         firstId = false
44:     end if
45:     if currentId != previousId then
46:         db.update({id: previousId},
47:             $set: db.VisitorId: visitorIds
48:         db.update({id: previousId},
49:         visitorIds= []
50:     end if
51:     visitorIds.append(r.UserId)
52:     previousId = currentId
53: end for
```

TABLE 5.2: An overview of the fields in each document in the collections

Document	Fields	Comment
Visitor	Id: string Behaviors: array ProfileUID: string CustomerUID: string	The behavior array is an array of documents with the fields Type, Id and Timestamp. This contains all the behaviors of the specific visitor
Product	Id: int ProductGroupId: int VisitorId: stringArray Description: string Created: DateTime	The visitorId array contains Ids of all visitors who have looked at this product

After the data has been cleaned and structured in No-SQL the algorithm for determining product recommendations can be made. The algorithm is described in the following section.

5.4 The product recommendation algorithm

There are multiple ways to implement af product recommendation algorithm all with their advantages and disadvantages. The method chosen for this project is called Item-to-Item collaborative filtering. Other methods and the reasoning why these weren't chosen is described in further detail in Chapter 6 discussion.

Item-to-Item collaborative filtering is a datamining tool to link items (products) with other items in terms of their similarity. This method is also the way *Amazon* handles their product recommendations [9].
The specifics of the algorithm differs from implementation to implementation. In this version each product is compared to other products based on how much they have been viewed together by customers, the likeness of their description and their product group.
The first run of the algorithm requires going through each product and the visitors of each product to see what else they have looked at. This needs a lot of resources, but once run only new behavior has to be re-calculated. A run down of the algorithm can be seen in algorithm 2.

Algorithm 2 Item-to-Item collaborative filtering algorithm

---

for all Products p **do**
    productScores = Dictionary<int, double>
    for all Visitors in p **do**
        for all Products visitorProduct in Visitor behaviors **do**
            if productScores contains visitorProduct **then**
                productScores[visitorProduct]++
            else
                productScores.Add(visitorProduct, 1)
            end if
        end for
    end for
    Sort productScores after highest value
    for all Products similarProduct in productScores **do**
        productScores[similarProduct] = **calculateSimilarityScore(p, similarProduct, productScores[similarProduct])** (see algorithm 3)
    end for
    Sort productScores after highest value
    Store top 10 productScores in database under p
end for

Algorithm 3 Similarity calculations for two products

---

calculateSimilarityScore(mainProduct p1, compareProduct p2, currentScore)
similarAttributeFactor = 0.02
productGroupFactor = 0
numOfSimAttributes = 0
if p1.productGroup equals p2.productGroup **then**
    productGroupFactor = 0.02
end if
for all words w in p1.description **do**
    if w is in p2.description **then**
        numOfSimAttributes++
    end if
end for
if numOfSimAttributes equals 0 **then**
    similarAttributeFactor = 0
end if
return $currentScore * (1 + productGroupFactor) * (1 + similarAttributeFactor^{numOfSimAttributes})$

After algorithms 2 and 3 each product in the database now has an array with the top 10 similar products based on amount of views, description and product group.

Next up is calculating the top products for each visitor, these are the products the specific visitor has viewed the most. This is accomplished by iterating through each visitor, checking their behavior and storing their top products as a field in the database. This calculation can be seen in algorithm 4. This calculation also requires a large amount of resources the first time, but very little to maintain.

Algorithm 4 Calculations of each visitors top products

---

    for all Visitors v **do**
        visitorProducts = Dictionary<string, int>
        for all Behaviors b in v **do**
            topVistorProducts[b.Id]++
        end for
        Sort visitorProducts after highest value
        Store top 5 visitorProducts in database under v
    end for

Since all these calculations are made before the actual product recommendations are requested, the process of recommending products is quite fast. The recommendation process starts with retrieving the requested visitor's top products from the database, retrieving these products top similar products, sorting them based on their score and finally returning the amount asked for. The code for the recommendation part can be seen in algorithm 5

Algorithm 5 Get product recommendations

---

    visitorTopProducts = db.GetTopProducts(visitorUID)
    productRecommendations = Dictionary<int, double>
    for all Products p in visitorTopProducts **do**
        SimilarProducts = db.GetTopProductRecommendation(product)
        for all products simProduct in similarProducts **do**
            if productRecommendations contains simProduct **then**
                productRecommendations[simProduct] += similarProducts[simProduct]
            else
                productRecommendations.add(simProduct, similarProducts[simProduct]
            end if
        end for
    end for
    Sort productRecommendations after highest value
    return amount of productRecommendations requested

The entire process of requesting product recommendation, running algorithm 5 and returning them takes less than 40ms which is one of the non-functional requirements.

Some other paths are required in certain situations such as when the visitor does not have any behavior or not enough behavior to satisfy the amount of recommendations requested. In these cases the remaining recommendations are filled from the top 20 most popular products in the last 30 days. The top 20 products are calculated by checking the timestamp and finding those in the last 30 days and then counting how many times each product was viewed. The top 20 products are stored in the database and can be calculated through an API call.

5.5 Hosting the API

An API can be hosted in several different ways, through many providers. This product recommendation API is hosted through Amazon WebServices in an *EC2 Instance* [10]. To accomplish this the ASP.NET core project is built in a Docker container, the container is pushed to the Docker Hub and then pulled and run in the *EC2 Instance*. The database is similarly packed in a docker container and

run in the *EC2 Instance*. The Docker containers have exposed ports to the rest of the internet and can be accessed via *EC2 Instance* public DNS or IP.

Chapter 6

Experimental Validation

6.1 Validation of recommendations

This section takes two approaches to validating the implemented recommendation engine:

- A statistical approach using recall

- Concrete examples

6.1.1 Statistical approach

Validation of product recommendation engines is focused around two approaches [11]:

- Offline validation

- Online validation

Offline validation

For offline evaluation measures such as Root-Mean-Square-Error (RMSE) and Mean-Absolute-Error (MAE) [12] are often used. These measures requires user ratings on products which is not present in the data for this project. RMSE and MAE can therefore not be used to evaluate these product recommendation.

Another offline method is called Recall. This method functions by using a percentage of the data available as regular input data and another percentage as test data [11]. The Recall evaluation run in this project used 80 percent of the data as input and tested on the remaining 20 percent. More specifically the remaining 20 percent was used in the following way:

- Take each visitor with more than two behaviors

- Input half of the visitor's behaviors via the API

- Generate 5 recommendations for the specific visitor

- See if the remaining half of his behaviors are in the recommended 5 items.

This resulted in a total of 1,934 visitors tested. These visitors have 11,328 behaviors where half was used as input and half was used as control. This means the recommendation engine had to predict half of 11,328 (5,664) behaviors. The algorithm succeeded in correctly predicting 2974 behaviors.

The percentage of how many times the recommendation engine would have predicted a visitors actual behavior is therefore $2974/5664 * 100 = 52.5\%$ the result of this evaluation as well as the scripts used can be found attached with the source code.

The engine only had half of each visitors behavior to go on which could be only 1 behavior and still managed to guess correctly more than half of the time. The 47.5% of the time the algorithm did not correctly guess the behavior does not mean that these were bad recommendations - the visitors

might not have discovered these items themselves but as they were recommended they might have examined these as well. The percentage of correct behaviors guessed would be even higher if more product recommendations where requested in the test.

As the algorithm acquires more data on all visitors as well as the visitor requesting recommendations the predictions will become even more precise.

Online validation

When the recommendation algorithm is put into production several new and better ways of evaluating the system becomes available. As the visitors get recommendations their behavior is logged and it is therefore easy to see how many of the recommendations are actually used and adjust the algorithm thereafter. This adjustment can potentially be automatized by using machine learning.

6.1.2 Concrete examples

To give a better understanding of the recommendations given by the algorithm a few specific examples are given below.

Visitor AAF995AE-1DD0-41C6-898B-9CBEE884E553 has looked at the following products:

- **36991:** Playset Brandmand Sam fyrtårn med figur

- **37691:** Playset Brandmand Sam Havnestation

- **37799:** Firman Sam Ocean Rescue

- **38950:** Sejt Brandmand Sam udstyrssæt

- **40786:** Brandmand Sam helikopter med lys og lyd

- **42373:** Biler Brandmand Sam og brandbil

- **52818:** Udklædning tilbehør sej Brandmand Sam megafon

- **52919:** Playset Fireman Sam

and is recommended the following products:

- **43215:** Biler Brandmand Sam bil

- **36991:** Payset Brandmand Sam fyrtårn med figur

- **37799:** Firman Sam Ocean Rescue

- **38950:** Sejt Brandmand Sam udstyrssæt med bælte

- **34392:** Brandmand Sam 104 cm Fastelavnstøj

The visitor has already looked at some of the items recommended, but others he has not. As the recommendations size increases more new products to the visitor will appear. The recommendations make sense since they are all "Brandmand Sam" stuff which is all he has looked at in the past. In the future the recommendation engine will be able to sort out his orders, but this has yet to be implemented.

Another **visitor 0036ECB4-F5AB-4B7C-824C-8D1C832CB65A** has looked at the following products:

- **43106:** Biler Scalextric C3528 BMW MINI Cooper S

- **49777:** Scalextric Racerbane C1368 Bilbaner Le Mans Prototypes Sports Cars

- **33136:** Chevrolet Camaro GT-R Biler Scalextric C3383

- **43104:** Scalextric C3524 VW Polo WRC Biler

and is recommended the following 5 products:

- **43106:** Biler Scalextric C3528 BMW MINI Cooper S

- **43104:** Scalextric C3524 VW Polo WRC Biler

- **49777:** Scalextric Racerbane C1368 Bilbaner Le Mans Prototypes Sports Cars

- **33136:** Chevrolet Camaro GT-R Biler Scalextric C3383

- **42841:** Maserati Trofeo Biler Scalextric C3388

These recommendations also makes sense theme-wise as they relate closely to the products the visitor has viewed.

A final example **visitor 22C9CF0F-8B96-4764-A2D1-6194992CEDC2** has looked at these products:

- **42809:** Bosch arbejdsbord Bosch Værktøj og Værktøjsbænke

- **42106:** Elsker du også bare paw patrol

and is recommended these products:

- **42106:** Elsker du også bare paw patrol

- **42809:** Bosch arbejdsbord Bosch Værktøj og Værktøjsbænke

- **40542:** LEGO Legends Of Chima Flyv op gennem skyerne

- **31548:** LEGO Legends Of Chima Snurrende slyngplanter

- **43713:** Fastelavnstøj Tid til at ringe efter politiet og Paw Patrols hund nummer 1

The two LEGO recommendations in this example might not seem thematically accurate, however since they have been recommended they must have a high similarity score to one or both of the products the visitor has looked at. The high similarity score is probably due to many visitors having viewed the LEGO products together with the other products in question.

6.2   Conclusion

The system has been validated statistically with the Recall method where the algorithm correctly predicted 52.5% of the visitors' behavior which would be even higher if more recommendations were requested in the test.. Concrete examples have also been examined and the recommendations makes sense when looked at. A few other measures such as RMSE and MAE could not be used due to the lack of product rating from the visitors.

Appendix A

API commands

Function	URI	Example	Description
GET	recommendation/visitorUID /numberOfRecommendatio ns/database	recommendation/AAF995A E-1DD0-41C6-898B-9CBEE88 4E553/5/Pandashop	Returns a JSON array of size numberOfRecommenda- tions containing productU- IDs which are the prod- uct recommendations for the specific visitor
PUT	visitor/visitorUID/database	visitor/AAF995AE-1DD0-41 C6-898B-9CBEE884E553/Pan dashop	Registers a new visitor with the database
PUT	product/productUID/descri ption/productGroup/databa se	product/5352/Agreatproduc t/5/Pandashop	Registers a new product along with its description and product group with the database
PUT	behavior/visitorUID/behavi orType/ItemID/database	behavior/AAF995AE-1DD0- 41C6-898B-9CBEE884E553/P roductView/5352/Pandasho p	Registers a new behavior for the specific visitor with the database
GET	Update/database/password	Update/pandashop/superse cretpassword	Builds the collaborative filter for the database
GET	Updatevisitortopproducts/d atabase/password	Updatevisitortopproducts/p andashop/supersecretpassw ord	Updates the top products for all visitors
GET	calculateTop20/database/pa ssword	calculateTop20/pandashop/ supersecretpassword	calculates the top 20 products in the last 30 days

Bibliography

[1] A. Trivedi. (Feb. 2014). Mapping relational databases and sql to mongodb, [Online]. Available: `https://code.tutsplus.com/articles/mapping-relational-databases-and-sql-to-mongodb--net-35650` (visited on 02/23/2017).

[2] M. D. C. Bowen. (Jan. 2017). What is normalized vs. denormalized data?, [Online]. Available: `https://www.quora.com/What-is-normalized-vs-denormalized-data` (visited on 02/23/2017).

[3] C. Buckler. (Sep. 2015). Sql vs nosql: The differences, [Online]. Available: `https://www.sitepoint.com/sql-vs-nosql-differences/` (visited on 02/23/2017).

[4] S. A/S. (May 2017). Struct a/s, [Online]. Available: `http://struct.dk//` (visited on 05/03/2017).

[5] S. Arora. (May 2017). How to use personalized product recommendations to increase average order value, [Online]. Available: `https://www.bigcommerce.com/blog/personalized-product-recommendations/` (visited on 05/03/2017).

[6] J. Mangalindan. (Jul. 2012). Amazon's recommendation secret, [Online]. Available: `http://fortune.com/2012/07/30/amazons-recommendation-secret/` (visited on 05/03/2017).

[7] S. IT. (May 2017). Db-engines ranking, [Online]. Available: `https://db-engines.com/en/ranking` (visited on 05/07/2017).

[8] P. C.V.V.A.S.A. C. Rai and M. Wenzel. (Nov. 2016). Choosing between .net core and .net framework for server apps, [Online]. Available: `https://docs.microsoft.com/en-us/dotnet/articles/standard/choosing-core-framework-server` (visited on 05/10/2017).

[9] G. L. B. Smith and J. York. (Feb. 2003). Amazon.com recommendations: Item-to-item collaborative filtering, [Online]. Available: `https://www.cs.umd.edu/~samir/498/Amazon-Recommendations.pdf` (visited on 05/08/2017).

[10] Amazon. (May 2017). Getting started with amazon ec2 linux instances, [Online]. Available: `http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/EC2_GetStarted.html` (visited on 05/08/2017).

[11] T. Řehořek. (Dec. 2016). Evaluating recommender systems: Choosing the best one for your business, [Online]. Available: `https://medium.com/recombee-blog/evaluating-recommender-systems-choosing-the-best-one-for-your-business-c688ab781a35l` (visited on 05/09/2017).

[12] J J. (Mar. 2016). Mae and rmse which metric is better?, [Online]. Available: `https://medium.com/human-in-a-machine-world/mae-and-rmse-which-metric-is-better-e60ac3bde13d` (visited on 05/09/2017).