UNIVERSITY OF SOUTHERN DENMARK

SOFTWARE ENGINEERING 6. SEMESTER

Datamining and its use

Author:
Lasse Bjørn HANSEN
Simon FLENSTED

Supervisor:
Jan Corfixen Sørensen SØRENSEN

UNIVERSITY OF SOUTHERN DENMARK

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

at

University of Southern Denmark
TEK

May 18, 2017

"Some quote"

- Gruppe 3

Contents

1 Introduction **1**
1.1 Motivation 1

2 Problem **2**
2.1 Problem description 2
2.2 Problem statement 2

3 Related work **3**
3.1 State of the art 3
3.2 Amazon 3

4 Requirements **4**
4.1 Requirements engineering 4

5 Design **7**
5.1 Conceptual overview of the system 7
5.2 Client-Server 8
5.3 Database design 9

6 Implementation **11**
6.1 Solution overview 11
6.2 Initial data dump 11
6.3 The product recommendation algorithm 12
6.4 Handling new data 14
6.5 Hosting the API 15

7 Experimental Validation **16**
7.1 Validation of the code 16
7.2 Validation of recommendations 17

8 Discussion **21**
8.1 Data storage 21
8.2 Maintainability 21
8.3 Product recommendations 21
8.4 Evaluating the algorithm 22

9 Conclusion **23**
9.1 Problem definition and research questions 23
9.2 Requirements fulfillment 24
9.3 recommendation accuracy 24

10 Future Work **25**
10.1 Future features . . . 25
10.2 Feedback based improvement . . . 25

A API commands **26**

B Python script for transfering products **28**

C No-SQL Product Document **30**

D Process **31**
D.1 Introduction . . . 31
D.2 Scrum . . . 31
D.3 GitHub . . . 32

E Case **33**

Bibliography **34**

1. Introduction

1.1 Motivation

The amount of data being processed on the server side and within large systems is continuously increasing. This data should be structured and modeled 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, data mining is very useful. The company Struct A/S [4] has provided 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 Data mining

Data mining is an analysis technique trying to discover useful information and relationships in large amounts of existing data [17].
Data mining has become an important 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 a database not optimized for product recommendations 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. A data set like the one provided by Struct A/S, can make it possible to predict customer needs.

2. Problem

2.1 Problem description

The initial problem/challenge is given to us by 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 a format not optimized for product recommendations 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 large amounts of 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?

3. Related work

3.1 State of the art

Datamining, webshop development and product recommendation algorithms is all something that has been around for quite some time now. This have resulted in great inspiration sources. In order to achieve the best possible result, some of the most successful developers of recommendation algorithms were investigated. The video streaming service, Netflix, has invested a lot of resources in coming up with the best possible recommendation algorithm [13]. This includes a worldwide competition for $1 million, called the Netflix Prize [14]. Netflix can definitively be considered state of the art in the video streaming field. The retail giant, Amazon, is another company having great success with its product recommendation system. Amazon's product recommendation system plays a big part in increasing their sales [15]. What these two giants have in common, is that they are both using a collaborative recommendation algorithm. This is also the reason why the recommendation algorithm of this project is developed with the same technique, and will be elaborated further in chapter 6. Other techniques include content-based filtering, knowledge-based recommendations and hybrid recommendations. These techniques could be applied to the product, however, Collaborative filtering serves best for the purpose of product recommendations. Content-based filtering is a good technique when recommending websites or articles. Knowledge-based recommendations is best put to use when high-involvement items are involved, such as cars, apartments, and financial services. Hybrid recommendations is a mix of all three methods [20].

3.2 Amazon

Amazon has achieved great success with their recommendation system. There are many different techniques to develop a good product recommendation algorithm, but to develop one that is both smart, efficient and increases sale can be a difficult task. Some of the most common methods are user-to-user collaborative filtering, clustering and item-to-item collaborative filtering. Amazon's recommendation system is based on the latter, due to its fast pace response and precise recommendations. When developing a user-to-user collaborative filtering algorithm you tend to end up with a precise, but slow recommendation system. By developing a clustering system, the response time can be very fast, but the quality of the recommendation will not be good [9]. Other recommendation systems have been developed, but Amazon comes out as one of the greatest successors in the business and their recommendation system is one of their strong assets [21].

4. Requirements

4.1 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 E), 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 final product's functionality fulfills the most important aspects of the case, and the requirements derived from the client meetings.

4.1.1 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 4.1. These requirements have been the driving force throughout the project. For a complete list of functional requirements, see the git backlog [22].

As we can see from table 4.1, the functional requirements of the final product can be compressed into 15 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 - *provide recommendations* was the most important requirement and has therefore acted as an ongoing task during the entire development of the product. F02-F15 were secondary requirements as they were not crucial before the recommendation algorithm was implemented. However, once the algorithm was applied to the system, these requirements were needed in order to keep the data updated.

4.1.2 Non-functional requirements

The non-functional requirements were described at an early stage, and later clarified at the planned meetings. The non-functional requirements can be seen in table 4.2.

The non-functional requirements are few, but turns out to be very challenging. The platform Struct use as the main tool for developing is based on the programming language C#. As a result of this, NF01 was derived. Net core was chosen because of its high performance and scalable systems, which was needed to realize NF02 - *Recommendations must be delivered within 40ms*.[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 [23]. This resulted in requirement NF03 - *Data should be stored in a scalable No-SQL database*. 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 - *The API should be easy accessible through a web service*.

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.
F13	The webshop developer can delete an existing product in the database. When the API is provided with the required information, the product should be deleted in order to keep the data up-to-date.
F14	The The webshop developer can store new order data for a order in the database. When the API is provided with the required information, new order data must be stored in the database.
F15	The webshop developer can delete an existing order in the database. When the API is provided with the required information, the order should be deleted in order to keep the data up-to-date.

TABLE 4.1: 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 4.2: Non-functional requirements

5. Design

5.1 Conceptual overview of the system

The system is developed as a part of the classic architectural pattern, Model-View-Controller (MVC) **??**. The system itself consists of the Model and Controller part, and lets the client be responsible for the view, which in this case is the web shop. Figure 5.1 shows how the system is layered. Data is sent to the controller which simply communicates the data between the logic (model) of the system and the view. The logic (model) is responsible for communication with the database, calculations regarding product recommendations, and handling new incoming data. The persistance layer is part of the model as described in [24] *"When you're working with a model you are thinking about business policies, perhaps database interactions."*.

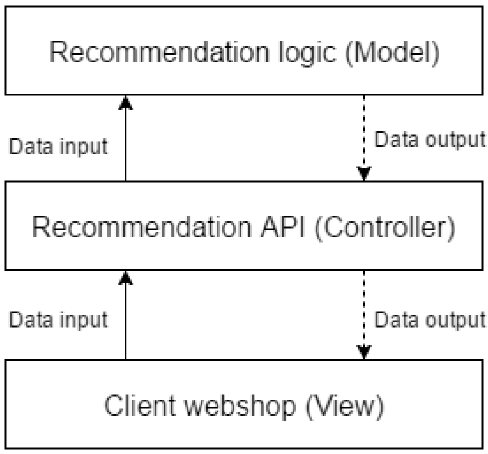


FIGURE 5.1: The Model-View-Controller (MVC) pattern applied on the system

5.1.1 Recommendation API (Controller)

The recommendation system is designed according to the MVC pattern described above. The controller layer takes input from the user (the web shop) and passes the information to the model layer. The model layer consists of the objects seen in the domain model in Chapter **??** as well as classes for handling the business logic and persistence. The different layers communicate through interfaces in order to be able to substitute implementations in the future.
The controller layer is split into two classes, one for handling when the user requests recommendations which relates to functional requirement NF01 and another for handling the data coming from the web shops relating to functional requirements F02-F15.

5.1.2 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. 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 Chapter 6.
A package diagram of the Controller and Model layer can be seen in figure 5.2

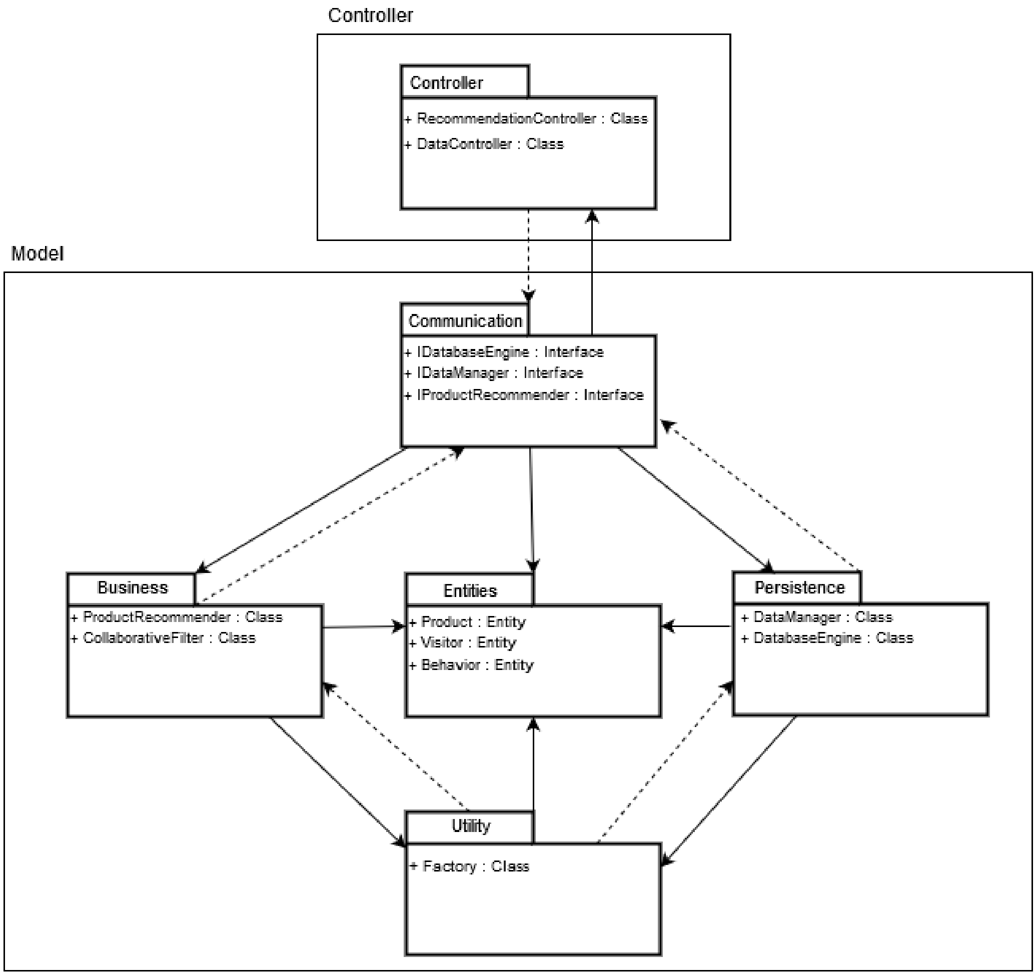


FIGURE 5.2: Package diagram of the model layer

5.2 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. In this scenario, the client is the web shop 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 5.3

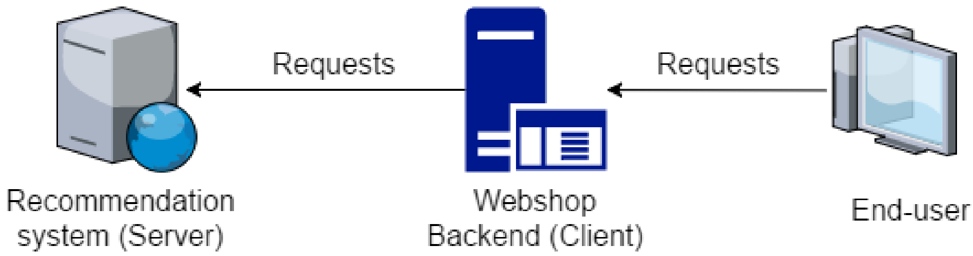


FIGURE 5.3: Package diagram of the model layer

5.3 Database design

Specific requirements for the storage of data was set by Struct, as they wanted a largely scalable structure of the data. The technology chosen was *No-SQL*. 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 whereas traditional relational databases have very strict data requirements, this also means that all data restrictions have to be handled in the code. 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 downside of No-SQL compared to relational databases is the fact it does not focus much on Online Transaction Processing (OLTP) which means there is no guarantee that the data is always stored completely. Overall No-SQL is a good fit for this project since scalability and speed is imperative.

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 datatypes compared to a row in SQL e.g arrays or other documents

The database design mimics the domain model by representing real world concepts such as a Visitors and their Behavior and Products. The No-SQL design can be seen in figure 5.2.

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 Behavior documents which 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
Behavior	Type: string Id: int Timestamp: DateTime	A behavior document holds information about a particular behavior

An example of a Visitort document can be seen in figure 5.4 and an example of a Product document can be seen in appendix C.

```
{
    "_id" : "C6CAFD83-0154-4D5E-8B59-0001B7530151",
    "ProfileUID" : "",
    "CustomerUID" : "",
    "TopProducts" : [
        NumberInt("53406")
    ],
    "Behaviors" : [
        {
            "Type" : "PRODUCTVIEW",
            "Id" : "53406",
            "Timestamp" : ISODate("2017-01-26T19:12:09.426+01:00")
        }
    ]
},
```

FIGURE 5.4: A Visitor document example

The topProducts array seen in Appendix C was not part of the original data transformation, but rather a part of the recommendation algorithm explained in Chapter 6.

6. Implementation

6.1 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 realization of the design and requirements.

6.2 Initial data dump

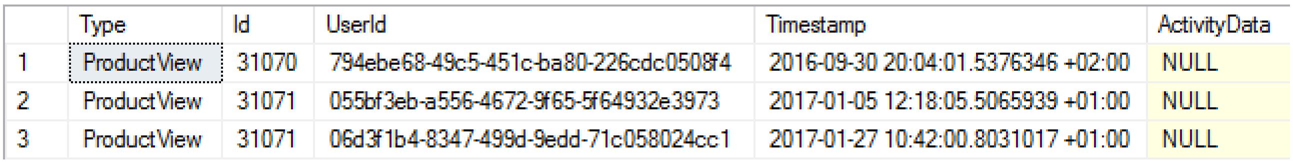
In the beginning of the project data from one of *Struct A/S'* customers was supplied. This data was in the form of many SQL tables and most of the data was not relevant for creating product recommendations. The main tables used were the following:

- 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 6.1 and 6.2.

Uid	UserAgent	BrowserName	BrowserVersion	IPaddress
77F224CD-A3D5-47A9-A2C8-000035EEB7DF	Mozilla/5.0 (iPhone; CPU iPhone OS 6_0 like Mac O...	Safari	6.0	66.249.[redacted]
FB24D20B-D34C-41CD-8F52-000043C12718	Googlebot-Image/1.0	Unknown	0.0	104.196.[redacted]
1449EB2C-ADD8-4358-AC6F-000045241C3F	Googlebot-Image/1.0	Unknown	0.0	66.249.[redacted]

FIGURE 6.1: 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 6.2: 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.

6.3 The product recommendation algorithm

There are multiple ways to implement a 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 were not chosen is described in further detail in Chapter 6 discussion.

Item-to-Item collaborative filtering is a data mining method 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. Pseudo code of the algorithm can be seen in algorithm 1.

Algorithm 1 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 2)
    end for
    Sort productScores after highest value
    Store top 10 productScores in database under p
end for
```

Algorithm 2 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 1 and 2 each product in the database now has an array with the top 10 similar products based on amount of views, description and product group.
The next step 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 3. This calculation also requires a large amount of resources the first time, but very little to maintain.

Algorithm 3 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 4

Algorithm 4 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 recommendations, running algorithm 4 and returning them takes less than 40ms which satisfies the non-functional requirement NF02.

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.

6.4 Handling new data

When new visitors are created or new behavior is discovered the client has to call the corresponding API functions in order to store this data alongside the other. When new behavior data is registered the program re-calculates the visitor's top product as well as the product's similar products. This way the algorithm is always up to date and the calculations can happen asynchronously and thereby have no affect the load times for the end user.

6.5 Hosting the API

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.

7. Experimental Validation

7.1 Validation of the code

The code is validated through a series of top-down, black-box integration tests. These tests test the methods of the two controllers including edge cases. An overview of the test cases can be seen in table 7.1.

TABLE 7.1: Integration test cases

Method	Test Case	Expected Result
GetRecommendationForVisitor	Valid arguments	String array of size 5
GetRecommendationForVisitor	Non existing visitor	String array of size 5
GetRecommendationForVisitor	Uppercase/lowercase visitorUID	String array of size 5
GetRecommendationForVisitor	Uppercase/lowercase database	String array of size 5
GetRecommendationForVisitor	Number of recommendations being larger than available products	String array of all valid products
GetRecommendationForVisitor	Visitor with no behavior	String array of size 5
GetRecommendationForVisitor	Non existing database	Empty string array
PutVisitor	New visitor	Visitor exists in database
PutVisitor	Existing visitor	Runs without exceptions
PutVisitor	Non existing database	Runs without exceptions
PutProduct	New product	Product exists in database
PutProduct	Existing product	Runs without exceptions
PutProduct	Non existing database	Runs without exceptions
PutBehavior	New behavior	behavior exists on visitor
PutBehavior	Non existing database	Runs without exceptions

Most of the tests of the recommendation part is tested with number of recommendations being 5, which is why they must return a string array of 5.
At the time of writing all tests runs successfully and a result overview can be seen in figure 7.1

FIGURE 7.1: Result of all integration tests

It is of course impossible to completely validate a systems correctness, but these go a long way in assuring that the system functions without errors. Since the system is highly data dependent use cases have been left out as this would require simulating a lot database methods.

7.2 Validation of recommendations

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

- A statistical approach using recall and precision
- Concrete examples

7.2.1 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.
Two other measures are called Recall and Precision. These measure are found using a method where a percentage of the data available is used as regular input data and another percentage as test data [11]. The 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.

The selected 20% visitors are chosen somewhat randomly as they are sorted by Id which is a randomized string assigned to each visitor. Another way of doing this is with cross validation where you divide the data into 10 folds and and use 9 as input and 1 for testing and then average the result over 10 times [11].

This resulted in a total of 1,934 visitors tested and a total of 9670 recommendations. 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. This success rate is visualized in figure 7.2

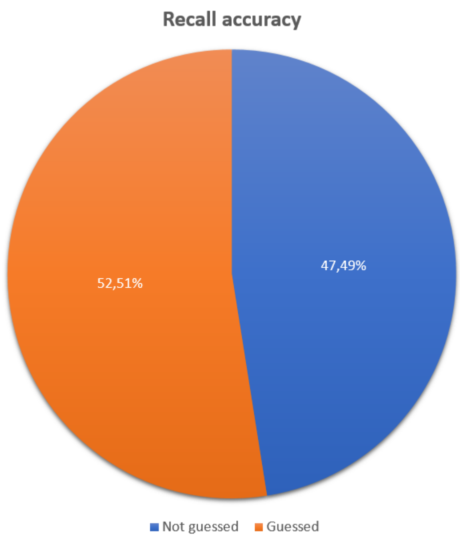


FIGURE 7.2: Recall accuracy

Recall is defined at the number of relevant items (successful guesses) retrieved divided by the total number of relevant items (all behavior). Precision is defined as the number of relevant items retrieved divided by the total number of documents retrieved (all recommendations).
In recommendation systems the two measures are also described as follows:

Recall *a perfect recall score of 1.0 means that all good recommended items were suggested in the list (although says nothing how many bad recommendations were also in the list)* [25]

Precision *a perfect precision score of 1.0 means that every item recommended in the list was good (although says nothing about if all good recommendations were suggested)* [25]

The recall percentages is calculated as $2974/5664 * 100 = 52.5\%$.
The precision percentage calculated as $2974/9670 * 100 = 30.8\%$.
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 as input which could be as low as 1 behavior and still managed to guess correctly more than half of the time. The precision score is only 30.8% in this test which implies that the algorithm also gives a lot of poor recommendations. One of the major drawbacks of these statistics is the fact that it assumes that every recommended item not in the visitors real behavior is a bad recommendation. This is not always the case as a visitor might not have looked at the product because he/she was unaware of its existence but might have decided to look at it if it was recommended [26]. Another drawback is that the numbers says nothing about catalog

coverage which is how much of the product catalog the recommendation system recommends. It is therefore possible to have a good recall rate but not be suggesting anything new to the visitors **??**. As the algorithm acquires more data on all visitors as well as the visitor requesting recommendations the predictions should 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 automated by using machine learning - this is covered in more detail in chapter 10.

7.2.2 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 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 are all "Brandmand Sam" stuff which is all he has looked at in the past. Whether it is good or bad depends on the business and how the visitors will respond. Struct A/S has said that this is fine, however only when the system is put into production can this be determined. 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 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. A closer look at the data shows the two LEGO products to have similarity scores of 24 and 15 respectively to product 42106. Since these products are not in the same product group and have zero matching descriptions the high similarity score is the amount of times they have been looked at together with this product. The main product, product 42106, have been looked at by 9 other visitors and these 9 visitors have then looked at the first LEGO product 24 times and the second LEGO product 15 times.

8. Discussion

8.1 Data storage

The first focus of the project was to organize a lot of data. This was done with the No-SQL framework, MongoDB. No-SQL was chosen because of its great flexibility and its high performance even when the amount of data accumulates. With the amount of data used for datamining, the project might have been a success if a traditional SQL database had been used. However, Struct A/S asked for a scalable way of handling the data was it to exceed billions of records. In order to accommodate this requirement, No-SQL was the better choice.

8.2 Maintainability

Another important issue was to create a system that allowed for easy maintenance. Amazon web-services served as the tool for deploying the system, and Docker created an environment that will make later updates easy to apply. The system itself has been developed in a way that allows each individual client (unique webshop) to feed it with new data. New calculations will automatically be done when new data is stored. This ensures that the recommendation algorithm is always creating recommendations based on the newest information. Other web-services could have been used, but as a low-budget project Amazon offered the best tools for free. Instead of using docker the application itself could have been deployed on a windows server, and the system would be just as easy accessible. This would have made the deployment part of the project a lot easier, since we would not have to learn a new technology. Using the Docker container allows for cross-platform deployment, and will be easier to deploy elsewhere in the future if needed.

8.3 Product recommendations

The main problem was to develop a good quality product recommendation algorithm. The algorithm of the project did undergo a lot of changes during the process. A user-to-user collaborative filtering algorithm was first implemented, but resulted in slow response times. The clustering method was declined before implementation, because research indicated that this would result in poor recommendations. [9] In the end item-to-item collaborative filtering was applied, which meant more datamining and more offline calculations. If the initial research about recommendation technologies had been more thorough, valuable time would not have been wasted on implementing the wrong algorithm. However, the mistakes gave great educational value and the final algorithm was implemented in a short amount of time. Which was due to the understanding gained while developing the initial algorithm.
The algorithm is meant to tailor the recommendations to each individual visitor. This is partly succeeded, but the item-to-item collaborative filter is based on the average customer behavior. This could result in poor recommendations for customers with more unique shopping habbits. Furthermore, if a new product is added, it would have to be visited by many different visitors before it would be recommended.

8.4 Evaluating the algorithm

One of the big challenges after implementing the recommendation algorithm was the evaluation. A comparison with Amazon's algorithm would be optimal, but Amazon does not share their algorithm with the public. Even better would be an online evaluation, as mentioned in chapter 7. A comparison to other open-source algorithms could be done, but the amount of resources required to setup such an experiment would drift the focus of the project in a wrong direction. The recall method appeared as the best offline option, however did not give the best objective evaluation of the final system. Combining the concrete examples with the recall evaluation did, however, provide a more nuanced justification of the final product.

9. Conclusion

9.1 Problem definition and research questions

The original problem definition was supplied by Struct A/S and involves them using their logged user activity data to create a personalized experience for the users. The answer to this is an API giving tailored product recommendations to the end user based on his/her behavior on the website. The API also allows for an easy way to add and maintain data about each visitor to keep recommendations up to date and relevant.
The problem definition derived the following three research questions "How can you optimally store and access data in a scalable way?", "How can this data be maintained and updated easily after deployment?" and "How can you utilize the organized data to generate tailored product recommendations for the end user?". These questions are answered below.

9.1.1 How can you optimally store and access data in a scalable way?

MongoDB, a No-SQL database, was chosen in order to accomplish scalable accessing and storing of the data provided by Struct A/S. MongoDB provides scaling functionalities due to its denormalized data which can more easily be spread across multiple servers **??**. No-SQL service platforms such as Azure or Amazon WebServices has built in scaling functions [18] which gives the distinct advantage that even if the data grows exponentially the hardware can keep up. Accessing all information about a certain user or product is also fast due to the denormalized structure without performing multiple joins or complex quires as would be the case with standard SQL databases. Overall MongoDB is a good fit for this type of project with growing data needs.

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

Maintaining and updating the data is as simple as calling the correct API functions as new data is produced. When a new visitor visits the site one API function will store the visitor in the database, similarly new behavior data is stored by calling another API function. These API functions can be called asynchronously meaning no extra load times for the end user, which is important in the online world.
Using Docker allows for easy updates and maintenance of the API as pushing and pulling new docker images from the Docker Hub is simple.

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

The data can be utilized through datamining. The datamining technique used to generate product recommendations in the project is Item-to-Item collaborative filtering. This process creates links between products based on their similarity and end users can thereby receive product recommendations based on the products they have already looked at. The recommendations are based on the entire user base and assumes users have similar tastes and purchasing patterns.

9.2 Requirements fulfillment

The requirements engineering process created **15** functional and 4 non-functional requirements. Of these requirements the most important one is F01 which is successfully met. functional requirements F02, F03, F05 and F06 have also been met. The remaining functional requirements are not met but are not imperative for generating product recommendations. The API is developed in ASP.NET core, the data is stored in a No-SQL database, the API is accessible through Amazon WebServices and product recommendations are generated in less than 40ms which means all non-functional requirements have been met.

9.3 recommendation accuracy

Recommendations generated by the algorithm are tested with the Recall method which resulted in correctly guessing user behavior 52.5% of the time. This percentage gives a good indication that the product recommendations are accurate and relevant to the end users. A few examples have also been examined and these also speak to the accuracy of the algorithm.

10. Future Work

10.1 Future features

This section focuses on the work which was not a priority in the versions of the API. Some of these features have not been implemented due to the needs of the current customer, but could prove useful in the future. Other features have not been possible due to lacking data, such as feedback from actual users.

10.1.1 Remaining requirements

In the future the remaining requirements should be fulfilled, this will allow the companies using the API to update and delete their data if, for example, a product is no longer in their catalog. Deleting older behavior will keep the recommendations more up to date if the consumer patterns shift.

10.1.2 Product ratings

The data structure could be changed to accommodate web shops with the possibility of users rating the products. This will also allow the algorithm to take the ratings into account to produce even more accurate product recommendations. Ratings on products also allows more test measures to be calculated, see chapter 7.

10.1.3 Order data

In the future it would make sense for the algorithm to take already ordered items into account. With this data the algorithm can filter out products which the visitor has already purchased. The collaborative filtering could also take orders into account. Adding orders to the similarity function would give a higher similarity score to products which have been ordered together rather than just looked at together. Oder data can be combined with the profile data already attached to the visitors.

10.2 Feedback based improvement

When the API gets put into use and feedback starts to be generated from the users the algorithm can use this and the accuracy of the recommendations will improve. For example if a user does not click on any of the recommendations the similarity function can be altered to try and remedy this. This can be further automated with the use of machine learning. A good machine learning implementation can make the algorithm learn from the generated feedback and automatically adjust some of the parameters to improve its own success rate *"One progressive step in Recommender Systems (RS) history is the adoption of machine learning (ML) algorithms, which allow computers to learn based on user information and to personalize recommendations further."* [16].

A. API commands

Function	URI	Example	Description
GET	recommendation/visitorUID/numberOfRecommendations/database	recommendation/AAF995AE-1DD0-41C6-898B-9CBEE884E553/5/Pandashop	Returns a JSON array of size numberOfRecommendations containing productUIDs which are the product recommendations for the specific visitor
PUT	visitor/visitorUID/database	visitor/AAF995AE-1DD0-41C6-898B-9CBEE884E553/Pandashop	Registers a new visitor with the database
PUT	product/productUID/description/productGroup/database	product/5352/Agreatproduct/5/Pandashop	Registers a new product along with its description and product group with the database
PUT	behavior/visitorUID/behaviorType/ItemID/database	behavior/AAF995AE-1DD0-41C6-898B-9CBEE884E553/ProductView/5352/Pandashop	Registers a new behavior for the specific visitor with the database
GET	Update/database/password	Update/pandashop/supersecretpassword	Builds the collaborative filter for the database
GET	Updatevisitortopproducts/database/password	Updatevisitortopproducts/pandashop/supersecretpassword	Updates the top products for all visitors
GET	calculateTop20/database/password	calculateTop20/pandashop/supersecretpassword	calculates the top 20 products in the last 30 days

B. Python script for transfering products

Algorithm 6 Product Script

```
SQLquery = SELECT * FROM struct.Product
db = MongoDB
for all Rows r in SQLquery do
    Product = {Id: r.id
            Description: ""
            Created: r.created
            visitorID: []
            ProductGroupId: 0 }
    db.insert(Product)
end for

Description = ""
firstId = true
previousId = 0
previousGroupId = 0
SQLquery = SELECT * FROM struct.product JOIN struct.attributeValueRendered ON id ORDER
BY productId desc
for all Rows r in SQLquery do
    currentId = r.ProductId
    currentGroupId = r.GroupId
    if firstId then
        previousId = currentId
        firstId = false
    end if
    if currentId != previousId then
        db.update({id: previousId},
                $set: db.description: description
        db.update({id: previousId},
                $set: db.ProductGroupId: previousGroupId
        description = ""
    end if
    description += r.description
    previousId = currentId
    previousgroupId = currentGroupId
end for
VisitorIds = []
firstId = true
previousId = 0
SQLquery = SELECT * FROM struct.BehaviorData
for all Rows r in SQLquery do
    currentId = r.Id
    if firstId then
        previousId = currentId
        firstId = false
    end if
    if currentId != previousId then
        db.update({id: previousId},
                $set: db.VisitorId: visitorIds
        db.update({id: previousId},
        visitorIds= []
    end if
    visitorIds.append(r.UserId)
    previousId = currentId
end for
```

C. No-SQL Product Document


```
{
    "_id" : NumberInt("31077"),
    "Description" : " Barbie børne dukke med tilbehør Chelsea og venner dukke. Barbies lillesøster Chelsea og hendes beds
    "VisitorId" : [
        "0FBD5E53-7E63-4C25-A1B8-DA790D8EB0A2",
        "3C01E864-F739-4D11-8FBC-905542265FF3",
        "A50A4F3F-DBA7-4682-BC98-0C351B53AB9F",
        "B4BA6490-01A2-4410-BB2A-BEE7802C1C24",
        "E1EE8B00-87F1-4851-9AB5-97670F039EEA",
        "E7A0F4A5-F63F-490A-9AC4-0DDF8E2A761C"
    ],
    "ProductGroupId" : NumberInt("779"),
    "Created" : ISODate("2016-04-12T16:53:26.030+02:00"),
    "TopProducts" : [
        {
            "ProductUID" : NumberInt("31077"),
            "Score" : 6.12
        },
        {
            "ProductUID" : NumberInt("38053"),
            "Score" : 4.08
        },
        {
            "ProductUID" : NumberInt("40782"),
            "Score" : 3.0600000000003917
        },
        {
            "ProductUID" : NumberInt("40425"),
            "Score" : 3.06
        },
        {
            "ProductUID" : NumberInt("40424"),
            "Score" : 3.06
        },
        {
            "ProductUID" : NumberInt("35963"),
            "Score" : 2.040000006528
        },
        {
            "ProductUID" : NumberInt("35954"),
            "Score" : 2.040000006528
        },
        {
            "ProductUID" : NumberInt("36209"),
            "Score" : 2.040000006528
        },
        {
            "ProductUID" : NumberInt("40776"),
            "Score" : 2.040000006528
        },
        {
            "ProductUID" : NumberInt("40774"),
            "Score" : 2.040000006528
        }
    ]
},
```


FIGURE C.1: A Product document example

D. Process

D.1 Introduction

This appendix elaborates on the process behind the development of the final product.

The project was initiated by the case presented by Struct A/S. The core of the case was 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."
Eventually this case led to the problem statement seen in chapter **??**.

A variety of technologies and methods was used throughout the realization of the product. To ensure that the process went as smooth as possible, the agile software development framework, Scrum, was used as a framework to structure and organize the work [19].

D.2 Scrum

Scrum is the main pillar for controlling the process of the project. Since the developing team only consisted of two students, Scrum is not applied 100% to the project. This section elaborates on the usage of Scrum and how the different roles were fulfilled.

D.2.1 Product owner

The product owner is typically in charge of which tasks needs to be done in what order. He is in charge of the product backlog and ensures that the developing team keeps adding value to the final product. Since there was no product owner in this project, the role is carried out by the team itself together with the company. The project group kept track of the backlog and had regular feedback from Struct, to ensure the development was on the right track.

D.2.2 Scrum master

The scrum master has the responsibility of removing impediments to the development team, and ensures that the scrum framework is followed. No actual scrum master was elected, which means that the development team along with our supervisor took on the role.

D.2.3 Workflow

Sprints of the length two weeks were chosen, as this was a fitting amount of time to develop and gain feedback. The sprint backlog was filled with issues and prepared before every the start of each sprint. Every work day started with a scrum meeting, where todays work was discussed. At the end of each sprint a meeting was set up with the customer (Struct A/S) to present what was implemented to ensure the project was still on the right track. The sprint backlog for the next iteration was presented as well, and then adjusted according to any feedback from the customer. By following these two week sprints, the project never deviated much from the wishes of the customer.

D.3 GitHub

The implementation of the final product required the usage of different tools. Github made it possible to structure and organize the planning, implementation and deployment of the project.

Git is a popular version control system and it is often used when developing software. GitHub is a web-based Git, and served as the primary tool when planning and developing the system. All planning is documented through GitHub issues and milestones. At the initial start, all project tasks was put in the product backlog which was made of issues. The sprints was created as milestones which was filled with issues before each iteration. The implementation of the system was controlled with Git. A consistent way of using the tool ensured that the newest version of the system was always available to the other group member, and a roleback was always possible had it been necessary. The newest version of the system was always to be found on the *master* branch. When adding new code, the group members had to create a new branch from the *master* branch, to avoid conflicts later. Once the new code was added in its own branch, and tested to ensure there were no flaws, it was merged into the newest version of the *master* branch. Besides making it possible to work simultaneously on the project, GitHub also served as a backup of the entire project.

E. Case

Dataopsamling og anvendelse

I forbindelse med alle sites vi lancerer, hvadenten der er tale om almindelige websites eller webshops, logges meget af brugeres aktivitet. Vi har således en stor mængde data for alle vores sites, men pt. gør vi ikke aktivt brug af det.

Vi ønsker fremtidigt at kunne anvende logget data til at skabe større indsigt i brugeres adfærd på vores sites, samt være i stand til aktivt at anvende dette data til at kunne skabe en personaliseret oplevelse for brugere af sitet.

Data logges i dag i en MS SQL server i et denormaliseret format. Vi ser følgende interessante områder i forbindelse med ovenstående, som vi kunne tænke os at have undersøgt nærmere og eventuelt lavet proof of concept på.

- Datalagring – hvordan lagres data bedst muligt, så dataopsamling kan skalere til flere milliarder records uden det går ud over performance hverken ved skrivning af data eller anvendelse af data

- Skabe et "Insights" dashboard

    - Vise standard metrikker såsom gennemsnitlig tid forbrugt på site
    - Via datamining eller lignende teknikker vise analyser over brugeres adfærd og skabe aggregerede oplysninger, såsom sammenhænge mellem kunders interesse for produkter og artikler (fx hvis det viser sig at brugere der kigger for produkter indenfor et bestemt segment også lader til at søge meget information omkring kundeservice)

- Personaliseret oplevelse
    - Anvendelse af opsamlet data til at skabe en personaliseret oplevelse for brugere af vores websites – eksempelvis produktanbefaling, som gør det muligt at vise de produkter systemet tror en bestemt bruger er mest interesseret i på forsiden
    - Vha. aggregerede findings (se ovenfor) vise relevant ekstern information i sammenhæng med produkter

FIGURE E.1: Original case given by Struct A/S

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. David and 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. Carter, V. V. Agarwal, S. Addie, 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. Linden, 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-c688ab781a35 (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).

[13] G. Lubin. (Sep. 2016). How netflix will someday know exactly what you want to watch as soon as you turn your tv on, [Online]. Available: http://www.businessinsider.com/how-netflix-recommendations-work-2016-9?r=US&IR=T&IR=T (visited on 05/13/2017).

[14] Wikipedia. (Apr. 2017). Netflix prize, [Online]. Available: https://en.wikipedia.org/wiki/Netflix_Prize (visited on 05/13/2017).

[15] T. Krawiec. (2017). The amazon recommendations secret to selling more online, [Online]. Available: http://rejoiner.com/resources/amazon-recommendations-secret-selling-online/ (visited on 05/13/2017).

[16] I. Portugal, P. Alencar, and D. Cowan. (). The use of machine learning algorithms in recommender systems: A systematic review, [Online]. Available: https://arxiv.org/ftp/arxiv/papers/1511/1511.05263.pdf (visited on 05/14/2017).

[17] businessdictionary. (2017). Data mining, [Online]. Available: http://www.businessdictionary.com/definition/data-mining.html (visited on 05/15/2017).

[18] M. Azure. (). Azure cosmos db, [Online]. Available: https://azure.microsoft.com/da-dk/services/cosmos-db/?WT.srch=1&wt.mc_id=AID529505_SEM_8cs5Yo1z (visited on 05/15/2017).

[19] Wikipedia. (May 2017). Scrum (software development), [Online]. Available: https://en.wikipedia.org/wiki/Scrum_(software_development) (visited on 05/18/2017).

[20] A. Felfernig, M. Jeran, G. Ninaus, F. Reinfrank, S. Reiterer, and M. Stettinger. (). Basic approaches in recommendation systems, [Online]. Available: http://www.ist.tugraz.at/felfernig/images/author.pdf (visited on 05/18/2017).

[21] J. Desjardins. (Dec. 2016). The extraordinary size of amazon in one chart, [Online]. Available: http://www.visualcapitalist.com/extraordinary-size-amazon-one-chart (visited on 05/18/2017).

[22] L. Hansen and S. Flensted. (May 2017). Github product backlog, [Online]. Available: https://github.com/HHIED/Datamining/issues (visited on 05/18/2017).

[23] E. Ciurana. (). Getting started with nosql and data scalability, [Online]. Available: https://dzone.com/refcardz/getting-started-nosql-and-data (visited on 05/18/2017).

[24] M. Fowler, *Patterns of enterprise application architecture*. Addison Wesley, 2002.

[25] M. Caraciolo. (Apr. 2011). Evaluating recommender systems - explaining f-score, recall and precision using real data set from apontador, [Online]. Available: http://aimotion.blogspot.dk/2011/05/evaluating-recommender-systems.html (visited on 05/18/2017).

[26] G. Shani and A. Gunawardana. (). Evaluating recommendation systems, [Online]. Available: http://www.bgu.ac.il/~shanigu/Publications/EvaluationMetrics.17.pdf (visited on 05/18/2017).