Hogeschool van Amsterdam

Big Data Scientist and Engineer

Final Assignment 2

Sylvan Ridderinkhof – BD1 – 500766715

28/01/2020

HBO-ICT Software Engineering – Hogeschool van Amsterdam

Version: 1.1

Changelog

Date	Version	Author	Changes
25/01/2020	1.0	Sylvan Ridderinkhof	Create initial document, add methods
28/01/2020	1.1	Sylvan Ridderinkhof	Add conclusion, theory and describe results

Summary

This report contains the theory behind the different models that were used, the theory behind the visualization, with Shneiderman's mantra, and the theory behind the problems with large datasets. RAM storage problems can be solved by storing everything on the hard drive with references in the RAM. From this theory. Based on this theory, some methods are decided, like what to use for creating the different models, how to improve the speed of the application and how to make it look good and function well. These methods include using FFBase to solve any possible RAM storage problems by storing big datasets on the local storage, rather than the RAM. Using Apache Spark to simulate parallel processing with the creation of the models and evaluation of the models, live sentiment analysis. Lastly using shiny to visualize data and using a live MongoDB connection with aggregations to present the data.

The six models that were trained are Random Forest, Decision Trees, Gradient Boosted Trees, Linear Regression, Support Vector Machine and Naïve Bayes. The models that came out the best are Linear Regression and Support Vector Machine. However, there is still lots of room for improvement for each of the six models, because of the tweaking that can be done in sparklyr.

Table of Content

1. Introduction

This is the second assignment for the course Big Data Scientist and Engineer. This report contains my experiences of creating a RShiny dashboard which contains a visualization based on Shneiderman’s mantra, furthermore it contains a database connection with MongoDB, a script to simulate RAM problems. It will also contain a chapter where I look back at the results. This document was written in two iterations, the first iteration being the initial programming of the R code and setting up the document structure. In the second iteration the code is moved into a nice-looking shiny application and writing down the results.

If you encounter any terminology that you do not understand, please refer to the “Glossary” chapter.

2. Background

This chapter contains the theory behind the models which have been used in this project, the theory behind the RAM problems and the theory behind the visualisation, and choices made in the dashboard.

2.1 Models

There are several models that have been trained for this assignment, below there is a small explanation of how these models work. The information for Naïve Bayes, Support Vector Machine and Random Forest is copied from my previous assignment.

2.1.1 Naïve Bayes

“Naïve Bayes (NB) is a classifier which uses the Bayes Theorem. It assumes that the value of each element which is being compared, is completely independent of the other elements. This means that the model will classify a given string based on the probability that each of those elements in the string is positive or negative. Then, based on the number of positive and negative elements it will determine if it is positive or negative (Narayanan, 2013).” (Ridderinkhof, 2019)

2.1.2 Support Vector Machine

“A support vector machine (SVM) classifies every data point as a vector. Each vector is then placed in a hyperplane. SVM can create multiple sets of hyperplanes to better classify given vectors (developers, 2019). The developers at DeepAI explain hyperplanes in SVM the following way: “learning a hyperplane amounts to learning a linear (often after transforming the space using a nonlinear kernel to lend a linear analysis) subspace that divides the data set into two regions for binary classification.” (DeepAI, 2019). What this means is that the vector is placed onto a plane and then based on where it lies, it is either A or B, which is the classification. In the case of reviews, each word is placed on a plane, and then the positives and negatives are compared, they give a rating of either positive or negative.” (Ridderinkhof, 2019)

2.1.3 Random Forest

“Random Forest (RF) uses a lot of Decision Trees (DT), hence the name. A DT splits a classification question into all of the data. The final decision is made by asking questions until a final outcome is achieved. The outcome is always one single value (Sehra, 2019). RF has each individual DT give a classification, and then takes the most common outcome as its own outcome (Yiu, 2019). In the case of a review, every word is checked whether or not if it is mostly positive or negative, and then it determines it for the entire sentence. RF ensures the DTs are not correlated by splitting the features of the dataset into N subsets, where there are no subsets which contain every same value and where N is the number of DTs. A model trained can be improved by playing with the number of trees.” (Ridderinkhof, 2019)

2.1.4 Decision Tree

As explained above, therefore keeping it short, a Decision Tree (DT) has nodes, where each node has a yes or no answer, with which every leaf of the tree can be reached based on the questions asked to the model. (Brid, 2018)

2.1.5 Gradient Boosted Trees

A Gradient Boosted Trees model starts by calculating the average value of that which is being predicted. A residual is calculated by subtracting the predicted value of the actual value, with which a DT is made. There can be multiple residuals in one leaf, of which then the average is computed. New residuals are then calculated by subtracting the actual values from the leaves, as many times as specified by the algorithm. The final leaves are used for prediction. (Maklin, 2019)

2.1.6 Logistic Regression Classification

Logistic Regression (LR) uses a formula to calculate a value between 0 and 1. Then the creator of the algorithm specifies the cut-off value, which is then used to represent either a 1 or a 0. (Pant, 2019)

2.2 RAM problems

A data.frame in R can be quite large. Especially when there are many rows and many columns. For example, when you turn a corpus of one million rows of reviews with more than five words, into a document term matrix. This will result in a huge object. This object too big to put into RAM (active memory, of which you do not have a lot). This problem can be tackled by using a different structure. In R, there are several packages, the one that was used in this project is “ffbase”. This package allows the user to store the file temporarily on their storage (HDD or SSD). This means the storage used in ram is much smaller, because the package only needs to remember where to find which data. To show this, I’ve found some example code on RPubs. (Dzyuban, 2016) This modified to be simpler, example shows directly the impact of using the local storage and the RAM and can be found in figure 1.

```r
## Example as seen on https://rpubs.com/demydd/235941

# Load required libraries
library(ff)
library(ffbase)

# Make the initial data frame and show it's size
df_size <- 50000000
x <- data.frame(a = numeric(df_size), b = numeric(df_size), c = numeric(df_size))
print(paste(round(object.size(x)/1024/1024,2),"Mb"))
# Expected size is high

# Convert to ffdf object and show it's size
x1<-as.ffdf(x)
print(paste(round(object.size(x1)/1024/1024,2),"Mb"))
# Expected size to be smaler than that of the normal x.

# Remove the objects to save some RAM :)
rm(x)
rm(x1)
```

Figure 1: Basic FFBase example.

The output

The first print statement shows off the size of the data frame in the computer’s RAM, which is 1144.41 Mb, whereas converting this data frame to a FFDF, supplied by FFBase, the RAM that is used is only 0.01Mb. This solves the RAM problems; however, the user might eventually, on giant datasets, run into the problem that there is not enough disk space. But disk space is both more plentiful and cheaper to increase.

2.3 Data Visualisation

To visualize data properly for a user, there is an important balance. The user needs to immediately see enough to know what they are looking at, but not be overloaded by details and functionality. To solve this, Schneiderman’s mantra is used, which is: Overview first, then zoom, then filter, then details on demand. First of all, the user should get some idea of what they can do with the data. Then they can zoom in and out, to dive deeper into the information. They can then filter out information that does not add value for them, and lastly, receive more details about elements that they find interesting.

3. Methods

3.1 General Code structure

The code was edited mostly in Atom, which is a text editor, with some attached packages to allow running the code as well. During this course I've also written a report on how to use different editors to program and compile R code, which I have included in appendix 2.

Initially, to just get all of the explicit requirements to function, I wrote one large R script called initial.R, which I have included in appendix 1. This script is actually fully functional and can achieve all of the requirements, namely: it implements FFBase to read in the dataset from the csv file, it splits the dataset into positive and negative and assigns a sentiment to them. It creates a shiny application with Shneiderman's mantra in mind, which uses a live mongoDB connection to retrieve data. Lastly, it has an implementation of Apache Spark, which simulates parallel processing in R. This is used to train the aforementioned several models.

The next step was getting all of this into a nice shiny dashboard. There are three main sections: Data gathering and storing, Data visualization and lastly Data Analysis. To get all of this into a good-looking dashboard, some structure was needed. In the project for Big Data, I had already created such a structure, based on a template supplied to us by teacher Odenhoven. I took the structure that the project had, replicated it in my personal assignment. This meant I could easily add pages and new components based on my requirements. The structure is: A "common" folder, which contains general code such as utility functions and loading functions. A database folder containing the connection and all the queries. Lastly a Shiny folder, split into Server and UI, the basics of a shiny application. The UI contains that which needs to be directly loaded, and the server folder will, after all the changes, contain all the good stuff like the machine learning with Spark.

Data is stored in and retrieved from a MongoDB database, using mongolite as the package. In figure 2 the different database functions that are used throughout the project are show. There is a connection to the database on the first line, some filtering and some aggregated queries.

```
db.connect <- function() {
  mcon <<- mongo(collection = "hotel_reviews", db = "hotel_reviews", url = "mongodb://lc
}

db.disconnect <- function() {
  mcon$disconnect()
}

db.findAll <- function() {
  mcon$find(
    query = '{}'
  )
}

db.drop <- function() {
  mcon$drop()
}

db.insert <- function(df) {
  mcon$insert(df)
}

db.findReviewsWithSentiment <- function() {
  mcon$find('{}', fields = '{"_id":0, "Review":1, "Sentiment":1}')
}

db.findRandomForLatLng <- function(lat, lng) {
  res <- mcon$find(
    '{}',
    fields = paste0(
      '{"_id":0, "Lat":1, "Lng":1,',
      '"Average_Score":1, "Hotel_Name":1, "Hotel_Address":1,',
      '"Total_Number_of_Reviews":1, "Reviewer_Nationality":1,',
      '"Review_Date":1, "Review":1, "Reviewer_Score":1, "Tags":1 }'
    )
  )

  # Filtering in the query is impossible due to float inaccuracy, therefore filter after
  # Query with the values of the floats times 10 mil.
  res <- filter(
    res,
    floor((Lat*10000000 - lat*10000000)) == 0 & floor((Lng*10000000 - lng*10000000)) ==
  )

  i <- sample(nrow(res):1, 1)
  ret <- c(
    res$Hotel_Name[i], res$Average_Score[i], res$Hotel_Address[i], res$Total_Number_of_R
    res[i, "Review"], res[i, "Reviewer_Score"], res[i, "Reviewer_Nationality"], res[i, "
  )
  return(ret)
}

db.getMarkersAggregated <- function(min, max, minReviews, hotelNames) {
  df <- mcon$aggregate('[
    {
      "$sort":{ "Lat":1, "Lng":1,"Average_Score":1, "Hotel_Name":1, "Hotel_Address":1 }
    },
    {
      "$group":{
        "_id":"$Hotel_Name",
        "Lat": { "$first" : "$Lat" },
        "Lng": { "$first" : "$Lng" },
        "Average_Score": { "$first" : "$Average_Score" },
        "Hotel_Name": { "$first" : "$Hotel_Name" },
        "Hotel_Address": { "$first" : "$Hotel_Address" },
        "Total_Number_of_Reviews": { "$first" : "$Total_Number_of_Reviews" }
      }
    }
  ]') %>%
    filter(Average_Score >= min) %>%
    filter(Average_Score <= max) %>%
    filter(Total_Number_of_Reviews >= minReviews)

  df <- na.omit(df)
  return(df)
}
```

Figure 2: Database file

3.2 Data Discovery and Preparation

The first step is Data Discovery. There is a button in the dashboard that can be pressed to start the process, as seen in figure 4. This figure also shows the time which the multiple steps took. As shown in the theory chapter, gathering a large dataset can lead to memory issues. This is why I decided to retrieve the data from the csv directly into an FFDF, supplied by FFBase. This results into a lot of files being stored on the disk, as seen in figure 5. The code in figure 3 shows the way that this is implemented. First, FFBase is set up with a temporary directory, then the read.csv.ffdf function is used to retrieve the raw data and store it onto the disk, with references in R. Then the data is sampled into a subset as required by the assignment: 10.000 positive and negative reviews, with a sentiment label assigned. It is important to note that negative reviews were also filtered on actual review rating being less than 5.5, to make sure the review was truly negative. This is then inserted into the MongoDB. As seen in figure 4, it takes my laptop 15 seconds to retrieve the data, and 20 seconds to write the data to the database. Quite long, in a future iteration there could be looked at a better solution for writing data.

```
dataset.observer.updateDataset <- function() {
  # Create ffbase local folder and use it as working directory
  incProgress(1/steps, detail = "Setting up FFBase")
  system("mkdir ffdf")
  ffbaseDir <- paste0(getwd(), "/ffdf")
  options(fftempdir = ffbaseDir)

  incProgress(1/steps, detail = "Getting raw data from CSV")
  # Step 1: Get the data from the csv file into a ffdf and store the time it takes
  timeRaw <- system.time(raw <- read.csv.ffdf(x = NULL, "Data/Hotel_Reviews.csv", encoding="ASCII"))

  # Step 2: Create a balanced dataset: There should be a collection of balanced set of reviews,
  # for instance a collection consisting of 10.000 positive and 10.000 negative reviews
  incProgress(1/steps, detail = "Create a useful subset (10000/10000 split of positive and negative)")
  timeSample <- system.time(dataset <- dataset.observer.sampleRawData(raw))

  # Step 3: Overwrite the database
  incProgress(1/steps, detail = "Update the database")
  db.drop()
  timeDB <- system.time(
    db.insert(as.data.frame(dataset))
  )

  return(c(timeRaw, timeSample, timeDB))
}
```

Figure 3: Updating the dataset

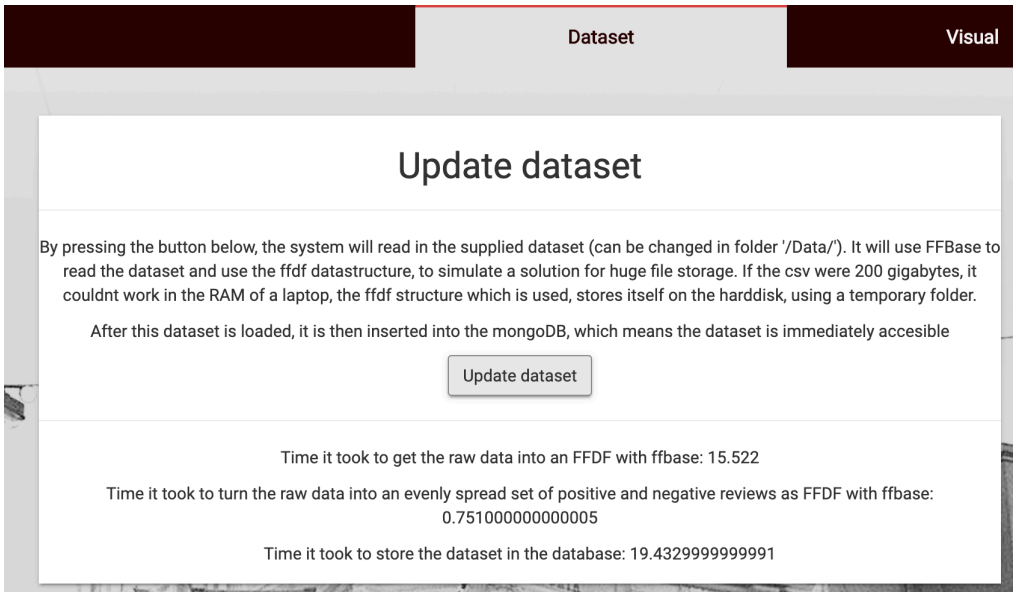

Figure 4: Result of updating the dataset

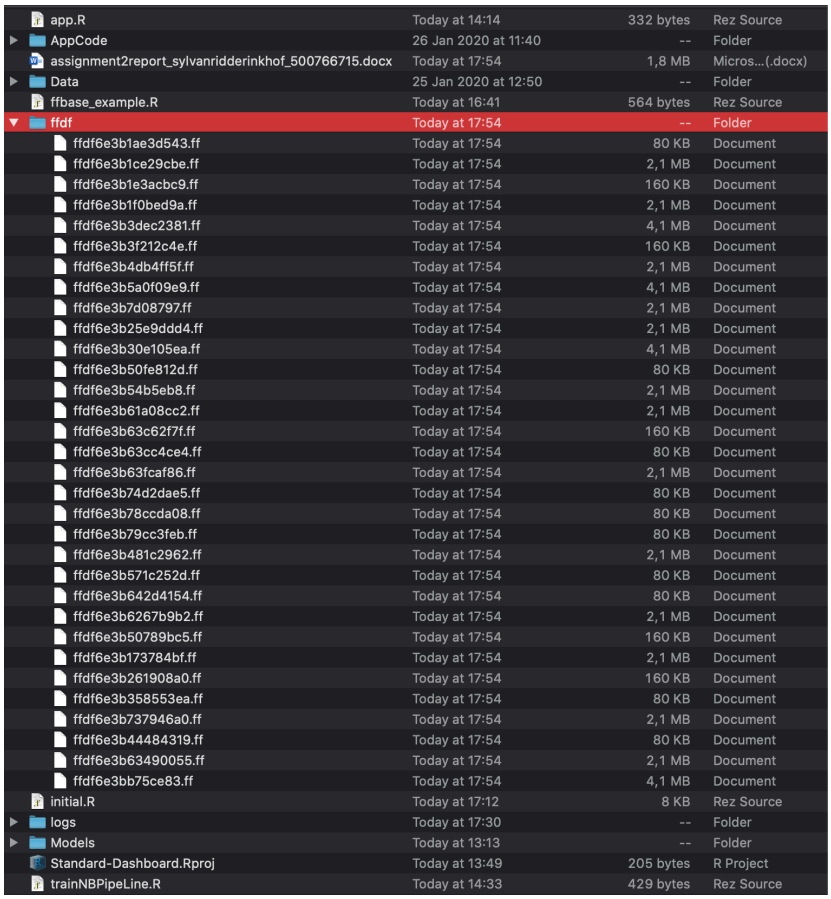

Figure 5: Temporary ffdf files

3.3 Creating the Map

To create the map and the page on which the map is found, the visualisation mantra was kept in mind. The first thing that comes up when you are looking for nice map visualisations in R, is Leaflet. Leaflet has a ton of nice features such as markers based on longitude and latitude, which is exactly that which is in our dataset. The first step was getting all of the data on the map; Overview first. The next step was zooming in, possible with leaflet by default, but also happens when you use the filtering, which is the third step. You can filter on: Minimum and maximum average review score, number of reviews and you can select specific hotels. The last thing to be added was details on demand. Leaflet, by default, has a popup that shows the user additional details. I decided to go a step further, and attached an observer to my shiny app, which listened to clicks on the map. This observer would receive the longitude and latitude of the marker, which was clicked, based on which a live query was fired to the MongoDB, to get the reviews of the corresponding hotel. I would have liked to directly filter out the correct hotel within the query, but there was a problem: Float inaccuracy. Due to float inaccuracy, the longitude and latitude received from the map did not directly match that which was stored in the database. Therefore, I gather all of the relevant data from the database and then use the dplyr package to filter out the correct longitude and latitude. I Solve the float inaccuracy there by multiplying the values with a large amount, which removes any decimals. The resulting data was then sampled (taken a random element of) and displayed in the dashboard. A random review and a lot of statistics are displayed.

Sketches for the different states can be seen in figure 6 (the scan is a bit overexposed), as we learned during Data visualisation that it is good to figure out what you want to show the user. On the top left, the overview is shown, which shows all the data. Zooming in shows the data more closely, but does not remove data, zooming back out still shows all the data that you could see before zooming in. Filtering removes data that is not encapsulated by the selected filters, zooming out does not show more data. Lastly, details on demand gives the user more details about a selected element.

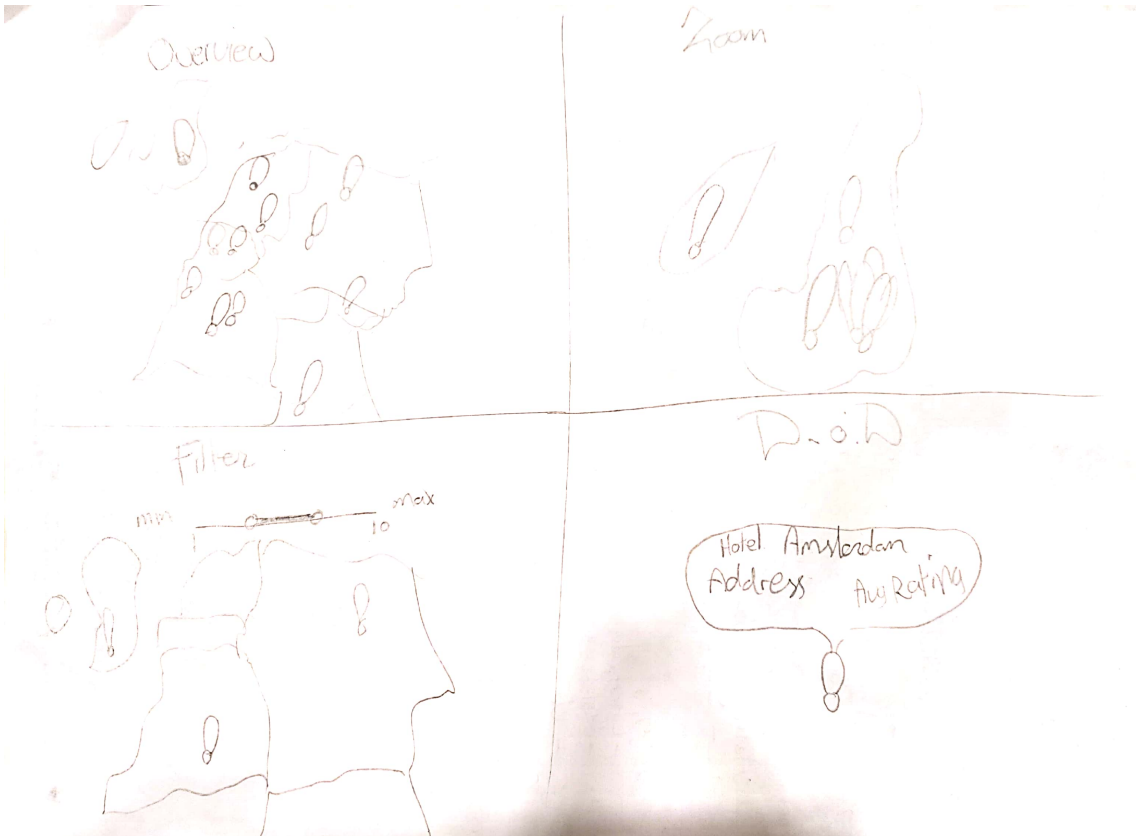

Figure 6: Sketches of Shneiderman's mantra

The aggregate query I used was to actually retrieve the data from the database that was needed to display the markers. It can be found in figure 2 from line 54. Each hotel would only need one marker, which meant grouping by hotel name. The results are also sorted and the first of all the to-be-displayed in the small label and popup are gathered. This aggregated query is fired every time a filter is changed by the user, to correctly show the markers on the map. The filters are also listening to changes in the other filters, which is nice, because then if you select a minimum rating of 9/10, you can only select specific hotels which have higher than or equal to that rating. Figure 7 shows the fact that the inputs from the filters are used to get the aggregated data, which is then passed to a map. The last part after the piping symbol only makes the map a bit prettier.

```
output$map <- renderLeaflet({
  score <- input[[id.filter.score]]
  minReviews <- input[[id.filter.minReviews]]
  hotelName <- input[[id.filter.hotelName]]

  mapPoints <<- db.getMarkersAggregated(score[1], score[2], minReviews, hotelName)
  if (!is.null(hotelName)) mapPoints <<- subset(mapPoints, Hotel_Name %in% hotelName)
  map() %>% addProviderTiles(providers$Esri.NatGeoWorldMap)
})
```

Figure 7: Rendering the map

3.4 Simulating Parallel Processing with Spark

R, by default, is mono-threaded. This means that anything that happens in R is done linearly, which does not take into account the capabilities of computers in 2020. A simple laptop already has four cores that can be used to parallelize tasks, yet R is still mono-threaded. To simulate parallel processing in R, we can use Apache Spark. There is an R package called sparklyr, that allows you to connect to a spark session, in which you can execute tasks. To use the available methods, your data needs to also be in the spark session. This is achieved by copying the data into the spark connection. Spark has a lot of built in functions, which I used to my advantage when creating models to use in my dashboard.

The models have been explained in chapter 2.1, and for each model, the sparklyr package has a dedicated function to train. To train a model, a correct dataset is required though. First of all, the reviews and their sentiments are retrieved from the database. Then a Document Term Matrix (DTM), occurrences of a term within the entire dataset, is built, and the sentiment is re-assigned. This DTM is then passed to the spark connection, as seen in figure 8. The data within the spark session is then split into an 80/20 train and test set, and then for each of the six aforementioned models, a model is built, tested and scored for the RoC and the accuracy. In figure 8 the code for Random Forest is displayed, the other models are all similar, with a nice function from sparklyr.

```
# Connect to spark
sc <- spark_connect(master = "local")
scdata <- sparklyr::copy_to(sc, dtmDF, "Hotel_Reviews")
# Create a 80/20 split
partitions <- sdf_random_split(scdata, training = 0.8, test = 0.2, seed = 1)

# Predict with Random Forest
system.time(rfModel <- ml_random_forest(partitions$training, Sentiment ~ ., type = "classification"))
rfPrediction <- ml_predict(rfModel, partitions$test)
rfResult <- ml_binary_classification_evaluator(rfPrediction)
```

Figure 8: Creating a Random Forest model with Spark

These models can be saved, to be retrieved later. There are seven models stored within the dashboard, the six you would expect, and one extra one. As I later discovered, it is not possible to simply pass a string to these models and have them churn out a prediction. To do this, a spark pipeline is required. As seen in figure 9, this pipeline does not take a DTM, but rather builds it by itself using the tokenizer and the count vectorizer. The resulting DTM is then used by the pipeline to build a model, by using the fitting function.

```
datasetDB <- mcon$find('{}', fields = '{"_id":0, "Review":1, "Sentiment":1}')
train <- copy_to(sc, datasetDB, overwrite = TRUE)

pipeline <- ml_pipeline(
  ft_tokenizer(sc, input_col = "Review", output_col = "tokens"),
  ft_count_vectorizer(sc, input_col = 'tokens', output_col = 'vocab'),
  ml_naive_bayes(sc, Sentiment ~ ., label_col = "Sentiment")
)

model <- ml_fit(pipeline, train)
ml_save(model, "Models/nbModelPipeline/")

```

Figure 9: Creating a pipeline for NB

This model is then loaded, and can be used to predict a new value, like a string that the user has given to the dashboard. This review is placed in a data frame, then passed to the spark session, and then the transform function in combination with the model gives a result, from which you can request the sentiment, as seen in figure 10.

```
sc <- spark_connect(master = "local")
# Load the model
incProgress(1/steps, detail = "Loading the model")
model <- ml_load(sc, "Models/nbModelPipeline")
# Insert the review text into spark as a dataframe.
incProgress(1/steps, detail = "Adding the review to Spark session")
reviewDF <- data_frame(Review = c(reviewInput))
sReview <- copy_to(sc, reviewDF, overwrite = TRUE)

incProgress(1/steps, detail = "Evaluating the review")
res <- ml_transform(model, sReview) %>% collect
```

Figure 10: Predicting input with the pipeline model

The ROC and Accuracy of the six models is calculated with the built-in functions from sparklyr as well, as shown in figure 11.

```
# Get RoC for all, then get accuracy for all.
rfRoC <- ml_binary_classification_evaluator(predictions[[1]])
dtRoC <- ml_binary_classification_evaluator(predictions[[2]])
gbRoC <- ml_binary_classification_evaluator(predictions[[3]])
lrRoC <- ml_binary_classification_evaluator(predictions[[4]])
svmRoC <- ml_binary_classification_evaluator(predictions[[5]])
nbRoC <- ml_binary_classification_evaluator(predictions[[6]])

rfAccuracy <- ml_multiclass_classification_evaluator(predictions[[1]], metric_name = "accuracy")
dtAccuracy <- ml_multiclass_classification_evaluator(predictions[[2]], metric_name = "accuracy")
gbAccuracy <- ml_multiclass_classification_evaluator(predictions[[3]], metric_name = "accuracy")
lrAccuracy <- ml_multiclass_classification_evaluator(predictions[[4]], metric_name = "accuracy")
svmAccuracy <- ml_multiclass_classification_evaluator(predictions[[5]], metric_name = "accuracy")
nbAccuracy <- ml_multiclass_classification_evaluator(predictions[[6]], metric_name = "accuracy")
```

Figure 11: Evaluating models

4. Results

Here, first off, the results of creating the dashboard are shown. Then, we look more deeply at the result of using the different models.

4.1 Visual results

In my opinion, the dashboard is very visually pleasing, and it is nice to have all of the functions that are in the “initial script”, available to you in a dashboard. First off, let’s start with the visualization.

4.1.1 The interactive visualisation

This is what opens up when you start the dashboard. Initially, you see everything; Overview First, as seen in figure 12.

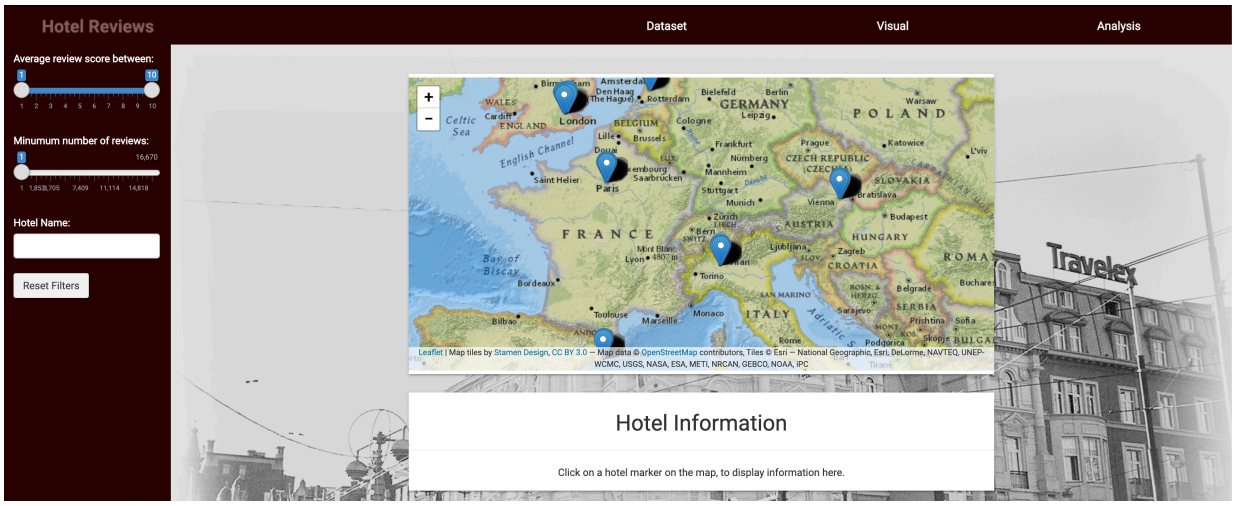

Figure 12: The map, overview

Then, the user can zoom in and out without losing data using the plus and minus signs on the top left. Filtering can be done using the filters on the left, as seen in figure 13.

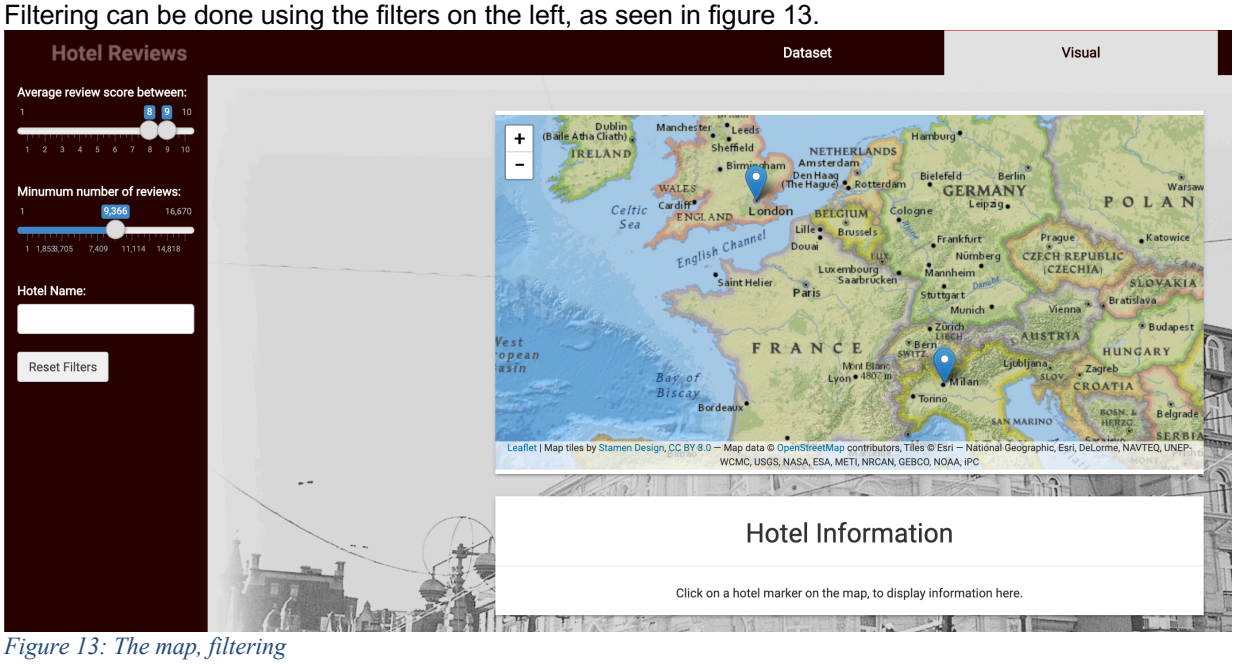

Figure 13: The map, filtering

Lastly, as indicated by the information at the bottom, the user can click on a marker to receive details on demand, as seen in figure 14.

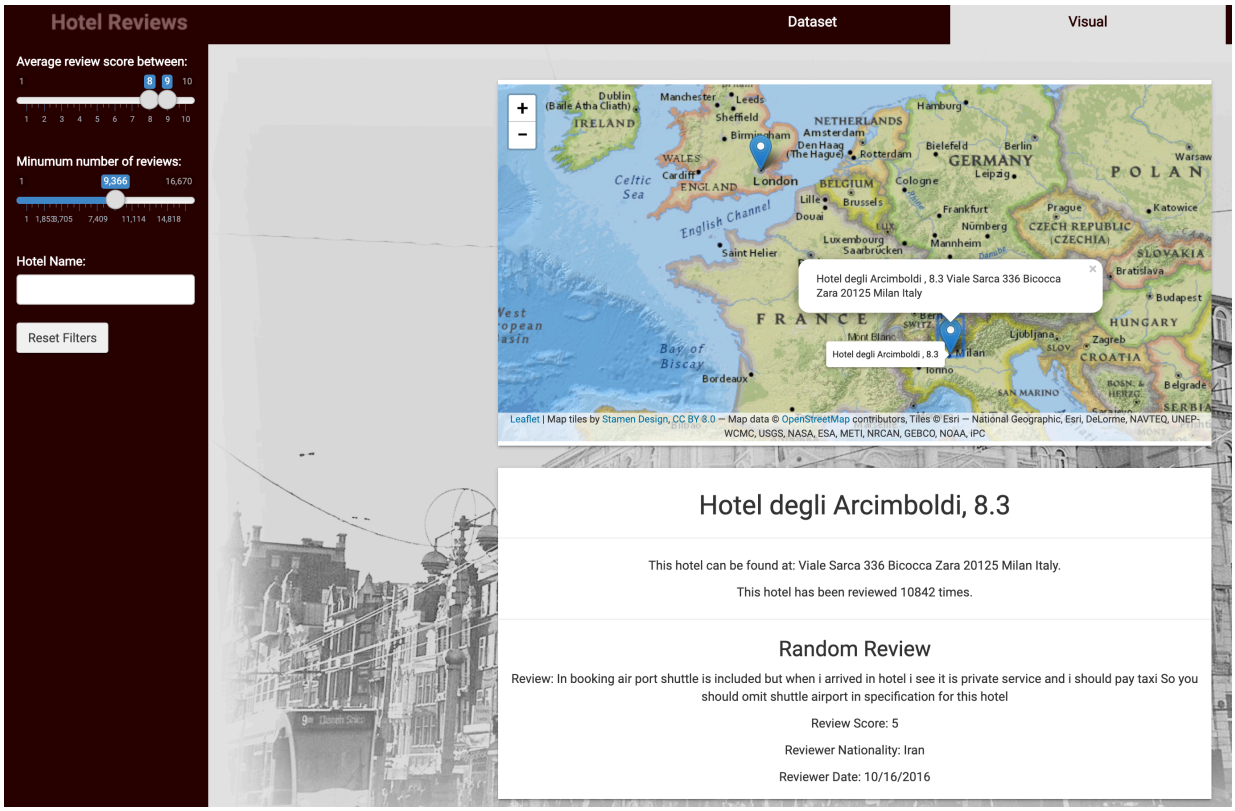

Figure 14: The map, details on demand

4.1.2 Updating the dataset

The result of updating the dataset can be seen in figure 4. As mentioned before, clicking the button updates the dataset in the MongoDB database using FFBase to reduce memory requirements. By only inserting the 20000 evenly spread positive and negative reviews, the time required to write to the database is reduced. One thing I would like to add to this page, is the option to choose the seed with which the random 20000 reviews are picked. This would allow the dataset to be changed to a pre-determined set.

4.1.3 Analysis

The analysis page consists of the three major components: Analysing reviews, showing results for stored models, and re-training the models with different splits and different seeds, which allows the user to find the best setup. These elements are visible in figure 15.

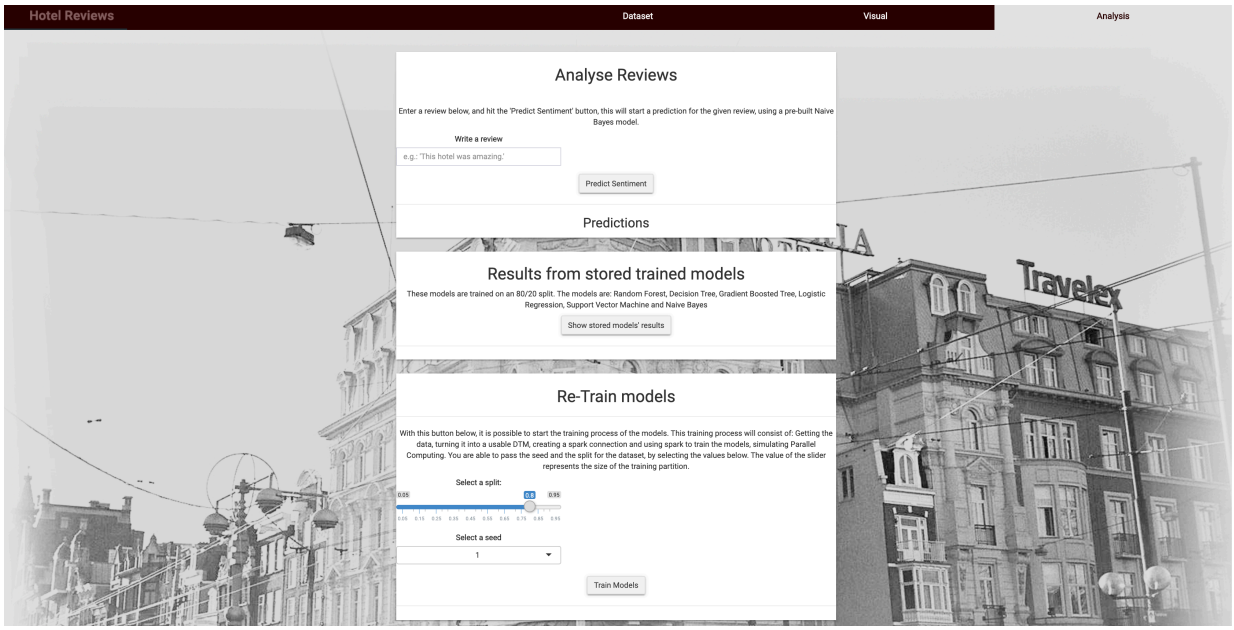

Figure 15: Analysis page

The first element, analyze reviews, uses a stored pipeline with naïve Bayes, to analyze the user's input. It will tell the user whether or not the input was positive, as seen in figure 16. One thing that can be improved in a future iteration is pre-loading the model when opening the page, because about 90% of the time waiting for the analysis is taken up by loading the model. However, that'd mean keeping the spark connection open at all times which is bad for performance.

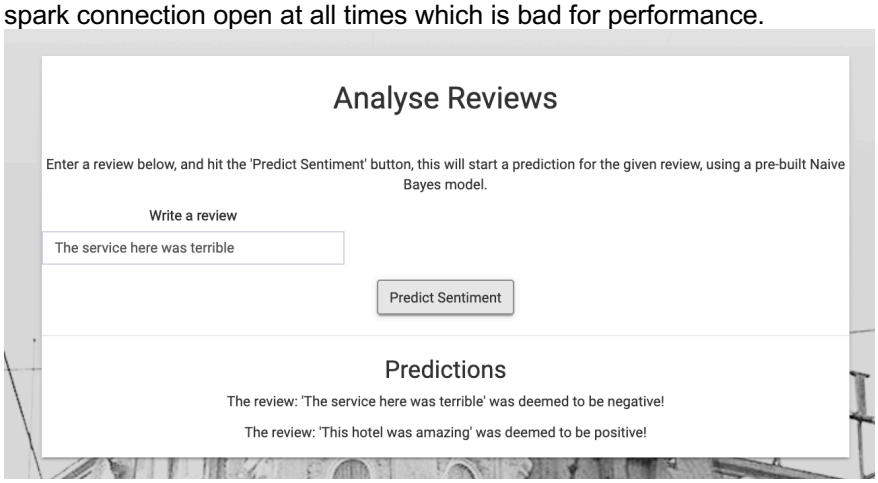

Figure 16: Analyzing reviews

The "Results from stored trained models" element retrieves the models and evaluates them on the spot. The loading takes some time, but then the evaluations are shown. It takes about a minute to load all the models, as seen in figure 17.

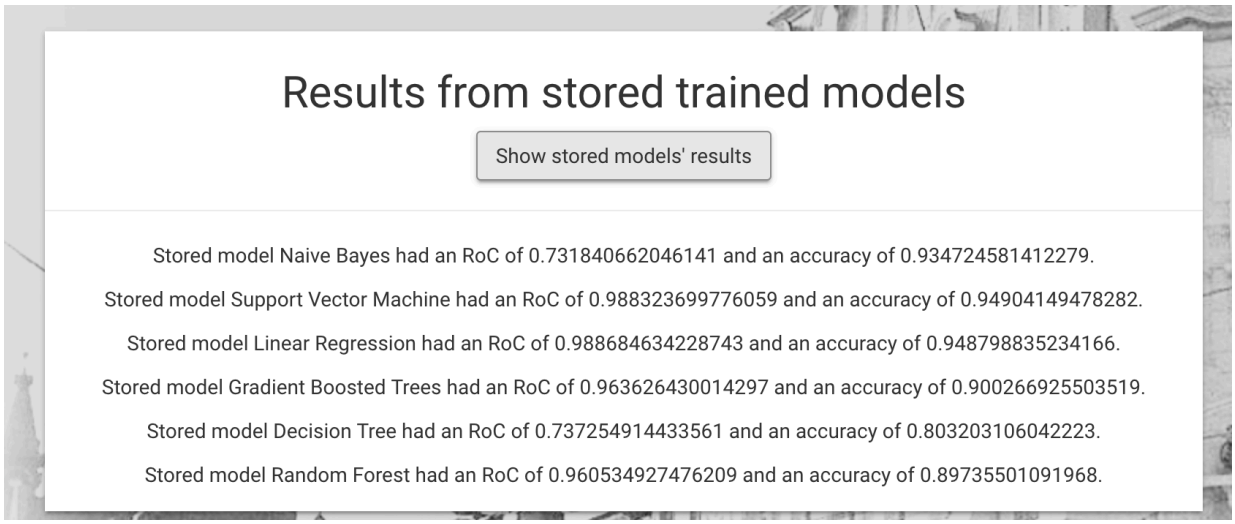

Figure 17: Result of stored models

Lastly, the Re-Train models element allows the user to re-train the models based on the seed and split for which the models are trained and tested. After pressing the button, you need to wait around 8 minutes, and then the output in figure 19 is shown.

Important elements on the screen that require the user to wait, all have loading indicators, one of which can be seen in figure 18. The user gets an idea of how far the progress is, and what is happening behind the closed doors.

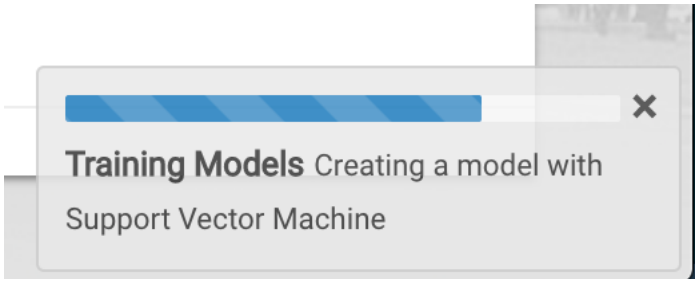

Figure 18: Loading indicator

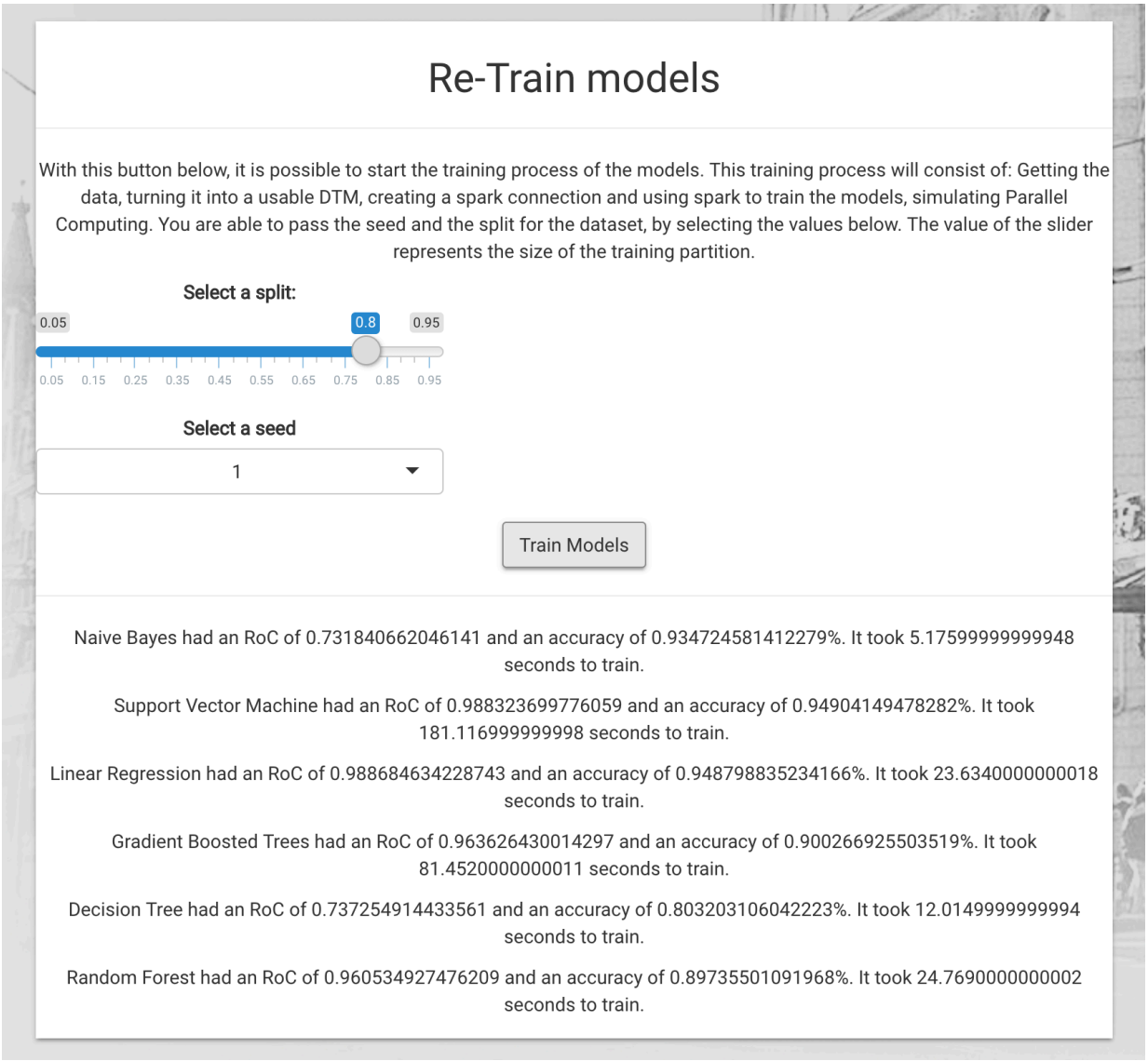

Figure 19: Re-training models with new parameters

4.2 Tweaking the models

My base point was an 80/20 split, with the seed 1. This led to the results that can be seen in the figure above, figure 19.

To check if tweaking the models by just changing the split and the seed has any major effects, I trained the models with 75/25, seed 3 and 85/15, seed 2 as well.

Random Forest
In table 1, the results are shown. In this table we see a slight increase in the ROC, the higher the split but the accuracy is the highest for the normal split. From this I draw the conclusion that the effects of changing the seed and the split is effective, gaining more than 1% accuracy over a different set. In a future iteration, also the number of trees and other parameters should be tweaked. There are so many parameters that can be altered, which could lead to a better result than changing the split and the seed.

	75/25, seed 3	**80/20, seed 1**	**85/15, seed 2**
RoC	95,52%	96,14%	96,63%
Accuracy	87,76%	89,01%	88,88%

Table 1: Results for RF

Decision Tree
In table 2, the results are shown. In this table we see a constant increase in the RoC and the accuracy. From this I draw the conclusion that the effects of changing the seed and the split are positive when the

split becomes higher. The more data that is used to train the model, the better. In future iterations, the maximum depth and other parameters should be tweaked as well.

	75/25, seed 3	80/20, seed 1	85/15, seed 2
RoC	72,02%	72,53%	73,76%
Accuracy	78,64%	79,72%	81,15%

Table 2: Results for DT

Gradient Boosted Trees

In table 3, the results are shown. In this table we see no real change in the RoC, but a steady increase in the accuracy of the model. From this I draw the conclusion that changing the seed and the split is a little inefficient for this model, although the accuracy did increase by 1,5%. In a future iteration the same parameters as in DT should be tweaked to get more improvement.

	75/25, seed 3	80/20, seed 1	85/15, seed 2
RoC	96,34%	96,80%	96,99%
Accuracy	89,64%	90,22%	91,13%

Table 3: Results for GB

Linear Regression

In table 4, the results are shown. In this table we see the best result for a more even split. From this I draw the conclusion that changing the seed and the split is not very effective. In future iterations a deep dive into the possible parameters for LR would be adequate. LR has a ton of options and can probably be even better than it already is with these default settings.

	75/25, seed 3	80/20, seed 1	85/15, seed 2
RoC	98,73%	99,20%	99,05%
Accuracy	94,38%	95,56%	94,98%

Table 4: Results for LR

Support Vector Machine

In table 5, the results are shown. In this table we see no real change in the RoC, however it is very high. From this I draw the conclusion that changing the seed and the split is not very effective for SVM. In future iterations the parameters should be tweaked to have an even higher accuracy

	75/25, seed 3	80/20, seed 1	85/15, seed 2
RoC	98,70%	99,13%	99,12%
Accuracy	94,42%	95,47%	95,41%

Table 5: Results for SVM

Naïve Bayes

In table 6, the results are shown. In this table we see, as expected, a bigger dataset means better accuracy, however the RoC drops. From this I draw the conclusion that changing the seed and the split is not necessarily the most effective for NB. This means that changing the DTM or the parameters should be able to improve the results.

	75/25, seed 3	80/20, seed 1	85/15, seed 2
RoC	73,10%	73,14%	71,72%
Accuracy	93,03%	93,25%	94,11%

Table 6: Results for NB

5. Conclusion

The dashboard looks fancy and is very usable. Shneiderman's mantra is applied in the interactive visualization. FFBase is a good tool to solve RAM problems and is used in the application to reduce memory load of the application. There is live querying to the database, including an aggregate. However, in the future, the map-reduce structure should be used. Spark was used to implement several machine learning algorithms, and with the use of the dashboard they were compared. The best accuracy was achieved by Linear Regression with 95,56%, closely followed by SVM with 95,47%, but taking into account the fact that LR was almost 8 times faster in training time than SVM, it seems that LR is the best option to continue with, at least on face value. The algorithms were tweaked, but there is room for improvement as described.

6. Glossary

Atom: A text editor.
Data.frame: A common table-like structure in R.
DTM: Document Term Matrix, which represents the number of times a term was seen in a document, in this specific case the document is the review.
Shneiderman's Mantra: Overview first, then zoom, then filter, then details on demand, a design principle.
Raw data: Purely gathered data, no cleaning or filtering has been done.
Labeled raw data: Raw data but with a positive or a negative label.
R-environment: The local session in which the R-code is run.
RVest: A web scraping package in R.
MongoDB: A no-sql database.
Accuracy: The number of correct predictions compared to the total amount of predictions.
RoC: The value that belongs to the area under the ROC curve.

7. References

Narayanan, V. A. (2013). Fast and accurate sentiment classification using an enhanced Naive Bayes model. *International Conference on Intelligent Data Engineering and Automated Learning*, 194-201.

developers, s.-l. (2019). *SVM*. Retrieved from scikit-learn: https://scikit-learn.org/stable/modules/svm.html

DeepAI. (2019). *Hyperplanes*. Retrieved from DeepAI: https://deepai.org/machine-learning-glossary-and-terms/hyperplane

Yiu, T. (2019). *Understanding Random Forest*. Retrieved from TowardsDatascience: https://towardsdatascience.com/understanding-random-forest-58381e0602d2

Sehra, C. (2019). *Decision Trees Explained Easily*. Retrieved from medium: https://medium.com/@chiragsehra42/decision-trees-explained-easily-28f23241248

Ridderinkhof, S. (2019, 10 28). BDSE Final Assignment 1.

Brid, R. (2018, 10 26). *Introduction to Decision Trees*. Retrieved 1 28, 2020, from Medium: https://medium.com/greyatom/decision-trees-a-simple-way-to-visualize-a-decision-dc506a403aeb

Maklin, C. (2019, 5 18). *Gradient Boosting Decision Tree Algorithm Explained*. Retrieved 1 28, 2020, from TowardsDataScience: https://towardsdatascience.com/machine-learning-part-18-boosting-algorithms-gradient-boosting-in-python-ef5ae6965be4

Pant, A. (2019, 1 22). *Introduction to Logistic Regression*. Retrieved 1 28, 2020, from TowardsDataScience: https://towardsdatascience.com/introduction-to-logistic-regression-66248243c148

Dzyuban, D. (2016, 12 16). *Out-of-memory packages (ff, ffbase)*. Retrieved 1 25, 2020, from rpubs: https://rpubs.com/demydd/235941

8. Appendices

8.1 Initial Codebase

```
 # install.packages("mongolite", "here")

library(here)
library(mongolite)

# setwd("here::here()")
setwd("/Users/sylvanridderinkhof/Projects/Courses/Big Data/BDSE2")

## Set up ffbase
# install.packages("ff")
#install.packages("ffbase")
#install.packages("doBy")

library(ff)
library(ffbase)
library(doBy)

# Create ffbase local
system("mkdir ffdf")
ffbaseDir <- paste0(getwd(), "/ffdf")
options(fftempdir = ffbaseDir)

# Step 1: Get the data from the csv file into a ffdf and show the time it
takes
system.time(raw <- read.csv.ffdf(x = NULL, "Data/Hotel_Reviews.csv",
encoding="ASCII"))

# Step 2: Create a balanced dataset: There should be a collection of
balanced set of reviews, for instance a collection consisting of 10.000
positive and 10.000 negative reviews having a least the following structure
set.seed(1); # using seed 1 to get the same random data set every time.

# Get all the positive reviews, assuming positive reviews have a minimum
rating of 5.5, drop others, also want at least 5 words.
pos <- ffdf("Hotel_Address" = raw$Hotel_Address,
  "Hotel_Name" = raw$Hotel_Name,
  "Lat" = raw$lat,
  "Lng" = raw$lng,
  "Average_Score" = raw$Average_Score,
  "Total_Number_of_Reviews" = raw$Total_Number_of_Reviews,
  "Additional_Number_of_Scoring" = raw$Additional_Number_of_Scoring,
  "Reviewer_Nationality" = raw$Reviewer_Nationality,
  "Review_Date" = raw$Review_Date,
  "Review" = raw$Positive_Review,
  "Review_Word_Counts" = raw$Review_Total_Positive_Word_Counts,
  "Total_Number_of_Reviews_Reviewer_Has_Given" =
raw$Total_Number_of_Reviews_Reviewer_Has_Given,
  "Reviewer_Score" = raw$Reviewer_Score,
  "Tags" = raw$Tags,
  "Sentiment"= as.ff(rep(1, times = nrow(raw)))
) %>% subset.ffdf(Reviewer_Score >= 5.5) %>% subset.ffdf(Review_Word_Counts
> 4)

# Get all the negative reviews, assuming negative reviews have a maximum
rating of 5.5, drop others, also want at least 5 words.
neg <- ffdf("Hotel_Address" = raw$Hotel_Address,
```

```
  "Hotel_Name" = raw$Hotel_Name,
  "Lat" = raw$lat,
  "Lng" = raw$lng,
  "Average_Score" = raw$Average_Score,
  "Total_Number_of_Reviews" = raw$Total_Number_of_Reviews,
  "Additional_Number_of_Scoring" = raw$Additional_Number_of_Scoring,
  "Reviewer_Nationality" = raw$Reviewer_Nationality,
  "Review_Date" = raw$Review_Date,
  "Review" = raw$Negative_Review,
  "Review_Word_Counts" = raw$Review_Total_Negative_Word_Counts,
  "Total_Number_of_Reviews_Reviewer_Has_Given" =
raw$Total_Number_of_Reviews_Reviewer_Has_Given,
  "Reviewer_Score" = raw$Reviewer_Score,
  "Tags" = raw$Tags,
  "Sentiment"= as.ff(rep(0, times = nrow(raw)))
) %>% subset.ffdf(Reviewer_Score < 5.5) %>% subset.ffdf(Review_Word_Counts >
4)

a <- pos[sample(1:nrow(pos), 10000),]
b <- neg[sample(1:nrow(neg), 10000),]
rownames(a) <- c(1:10000)
rownames(b) <- c(10001:20000)
# Select 10000 random rows from pos and neg, make them a single df.
pos <- as.ffdf(a)
neg <- as.ffdf(b)

dataset <- ffdfappend(pos, neg)

# Connect to db
mcon <- mongo(collection = "hotel_reviews", db = "hotel_reviews", url =
"mongodb://localhost")
# Insert into db
mcon$drop()
mcon$insert(as.data.frame(dataset))
# Retrieve data from db
datasetDB <- mcon$find('{}', fields = '{"_id":0, "Review":1,
"Sentiment":1}')


library(tm)
corpus <- Corpus(VectorSource(datasetDB$Review))
dtm <- DocumentTermMatrix(corpus) %>% removeSparseTerms(0.99)
dtmDF <- as.data.frame(as.matrix(dtm), stringsAsFactors = FALSE)
dtmDF <- cbind(dtmDF, Sentiment = datasetDB$Sentiment)

# Spark
# Setup spark
# install.packages("sparklyr")
library(sparklyr)
library(DBI)
# spark_install(version = "2.4.0")

# Connect to spark
sc <- spark_connect(master = "local")
scdata <- sparklyr::copy_to(sc, dtmDF, "Hotel_Reviews")
# Get data from spark
# dbGetQuery(sc, "SELECT * FROM hotel_reviews")

# Create a 80/20 split
partitions <- sdf_random_split(scdata, training = 0.8, test = 0.2, seed = 1)
```

```
# Predict with Random Forest
system.time(rfModel <- ml_random_forest(partitions$training, Sentiment ~ .,
type = "classification"))
rfPrediction <- ml_predict(rfModel, partitions$test)
rfResult <- ml_binary_classification_evaluator(rfPrediction)

# Predict with Decision Tree
dtModel <- ml_decision_tree(partitions$training, Sentiment ~ ., type =
"classification")
dtPrediction <- ml_predict(dtModel, partitions$test)
dtResult <- ml_binary_classification_evaluator(dtPrediction)

# Predict with Gradient Bossted Tree
gbModel <- ml_gradient_boosted_trees(partitions$training, Sentiment ~ .,
type = "classification")
gbPrediction <- ml_predict(gbModel, partitions$test)
gbResult <- ml_binary_classification_evaluator(gbPrediction)

# Predict with Logisic Regression
lrModel <- ml_logistic_regression(partitions$training, Sentiment ~ .)
lrPrediction <- ml_predict(lrModel, partitions$test)
lrResult <- ml_binary_classification_evaluator(lrPrediction)

# Predict with Multilayer Perceptron (neural network)
#mpModel <- ml_multilayer_perceptron_classifier(partitions$training,
Sentiment ~ ., layers = c(11,15,2))
#mpPrediction <- ml_predict(mpModel, partitions$test)
#mpResult <- ml_binary_classification_evaluator(mpPrediction)

# Predict with Support Vector Machine
svmModel <- ml_linear_svc(partitions$training, Sentiment ~ .)
svmPrediction <- ml_predict(svmModel, partitions$test)
svmResult <- ml_binary_classification_evaluator(svmPrediction)

# Predict with Naine Bayes
nbModel <- ml_naive_bayes(partitions$training, Sentiment ~ .)
nbPrediction <- ml_predict(nbModel, partitions$test)
nbResult <- ml_binary_classification_evaluator(nbPrediction)

ml_save(rfModel, paste0(getwd(), "/Models"), overwrite = TRUE)
ml_save(dtModel, paste0(getwd(), "/Models"), overwrite = TRUE)
ml_save(gbModel, paste0(getwd(), "/Models"), overwrite = TRUE)
ml_save(lrModel, paste0(getwd(), "/Models"), overwrite = TRUE)
ml_save(svmModel, paste0(getwd(), "/Models"), overwrite = TRUE)
ml_save(nbModel, paste0(getwd(), "/Models"), overwrite = TRUE)

results <- as.data.frame(cbind(c("Random Forest", "Decision Tree", "Gradient
Boosted", "Logistic Regression", "Support Vector Machine", "Naive Bayes"),
c(rfResult, dtResult, gbResult, lrResult, svmResult, nbResult)))
colnames(results) <- c("Type", "Accuracy")
results <- results[order(results$Accuracy, decreasing = TRUE),]

#### SHINY APP
# install.packages("leaflet")
library(shiny)
library(leaflet)

data <- mcon$find('{}', fields = '{"_id":0, "Lat":1,
"Lng":1,"Average_Score":1, "Hotel_Name":1, "Hotel_Address":1}')

ui <- fluidPage(
```

```
  leafletOutput("mymap"),
  p(),
  sliderInput("sliderScore", "Average Review Score Between:", 1, 10,
c(1,10)), # Slider with a min and a max.
  #selectInput("hotelInput", data$)
)

server <- function(input, output) {
  data <- mcon$find('{}', fields = '{"_id":0, "Lat":1,
"Lng":1,"Average_Score":1, "Hotel_Name":1, "Hotel_Address":1}')
  points <- na.omit(distinct(data))

  print(head(points))

  map <- function(points) {
    # create map
    renderLeaflet({
      leaflet() %>%
        addProviderTiles(providers$Stamen.TonerLite,
                         options = providerTileOptions(noWrap = TRUE)
        ) %>%
        addMarkers(
          lat = points$Lat,
          lng = points$Lng,
          label = paste(points$Hotel_Name, ", ", points$Average_Score),
          popup = paste(points$Hotel_Name, ", ", points$Average_Score, "\n",
points$Hotel_Address)
        )
    })
  }

  # Listen to the slider changing and filter the dataset
  observeEvent(input$sliderScore, {
    filteredPoints <- filter(points, Average_Score > input$sliderScore[1])
    filteredPoints <- filter(filteredPoints, Average_Score <
input$sliderScore[2])
    print(head(filteredPoints))
    output$mymap <- map(filteredPoints)
  })

  output$mymap <- map(points)
}

shinyApp(ui, server)
```

8.2 Link to “Editing code outside of RStudio” report.

https://drive.google.com/open?id=1exTyLXuePdQT6pFY3kxl98i8OPlDfkUf
or a shorthand link: https://bit.ly/2TYSTvs