BDSE Part 2

Quirijn Langedijk
500782499
quirijn.langedijk@hva.nl
email@quirijnlangedijk.nl

Inhoud

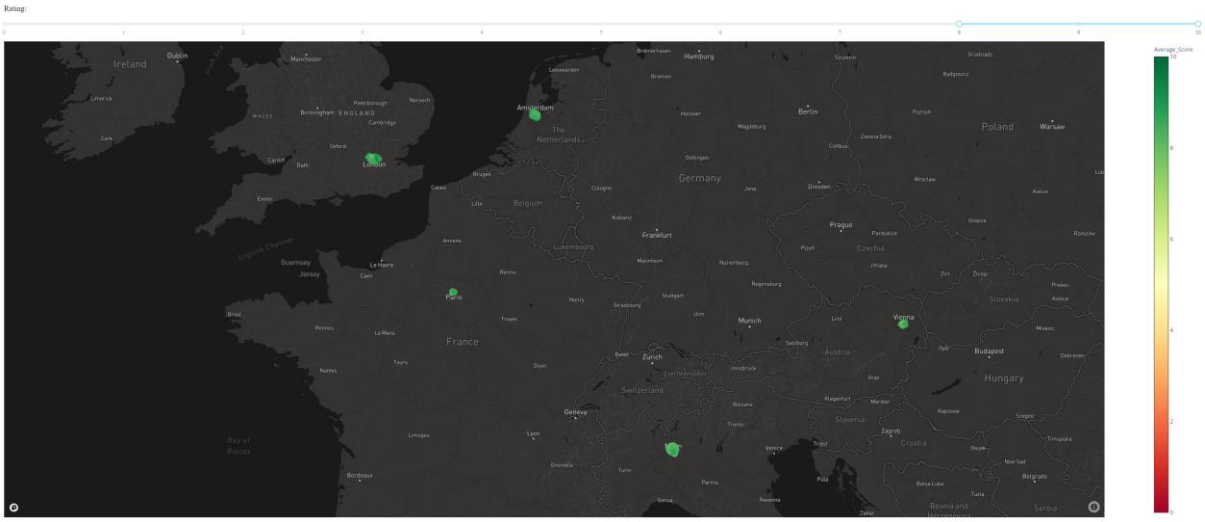
Summary

I used Spark to create two different models, Logistic Regression and Support Vector Machine. To create the best possible model, I used cross validation with both models. The 'best' models I created had an accuracy of 94.08% (LR) and 93.91% (SVM) and ROC-AUC of 98.07% (LR) and 97.86% (SVM)

I also checked if the accuracy went up if I increased the size of the dataset. In this case, it improved with both models. In the case of Logistic Regression the accuracy went up with 6.18% (1000 vs 100000 rows), the ROC-AUC went up with 4.62, while the time to train went up with 57 seconds. With Support Vector Machine, the differences between the biggest and smallest dataset were +6.81% accuracy, +4.04% ROC-AUC and an increase of 11m54s. Because of the shorter training time, and higher accuracy, I think Logistic Regression is the better model in this case.

I also did some small tests to compare the memory usage of Dask vs Pandas. In my case, Dask could handle way more data than Pandas, although loading data took a bit longer versus Pandas.

I also visualized the dataset that I used to train both models. The greener the dot, the higher the rating, the bigger the dot, the more ratings there are. It's also possible to filter the data on the average rating.



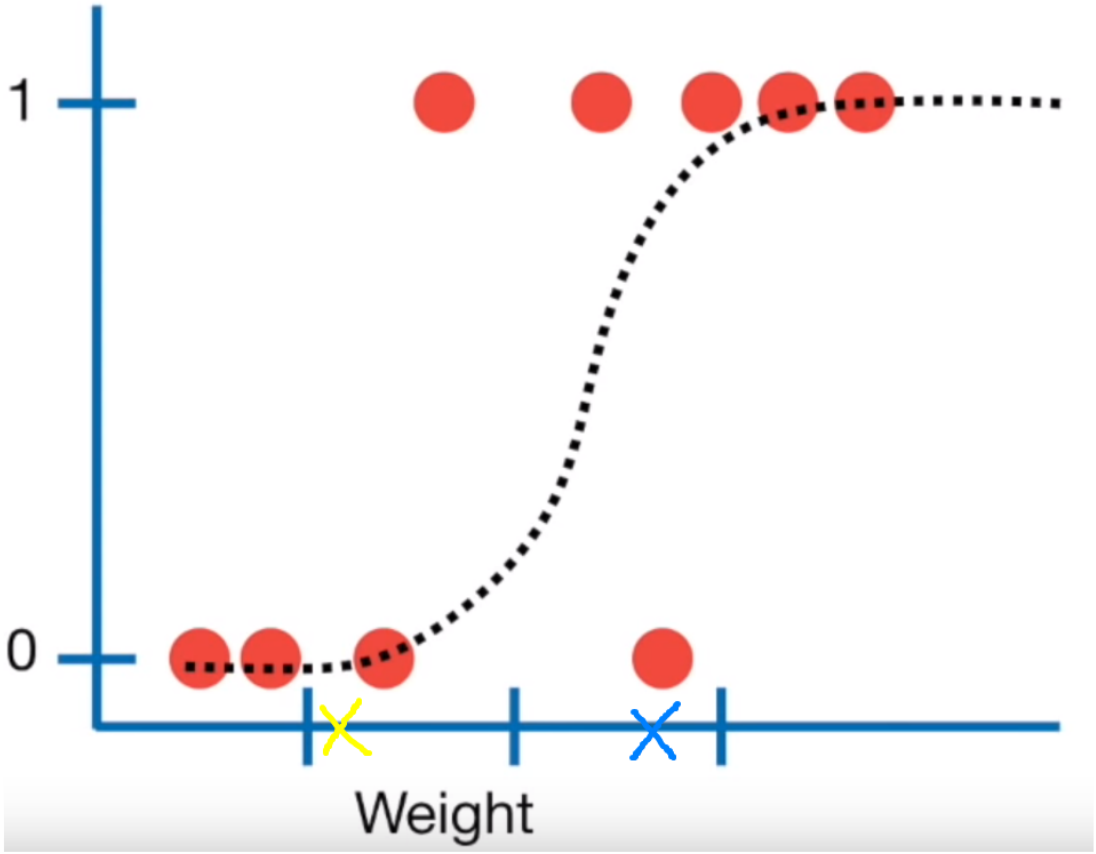
Introduction

With this assignment, it was our task to do a couple of things; Create a visualization of a dataset containing around 600.000 hotel reviews, use Spark to create a machine learning algorithm to classify the reviews as positive or negative, and create a simulation on how to solve storage problems.

Background

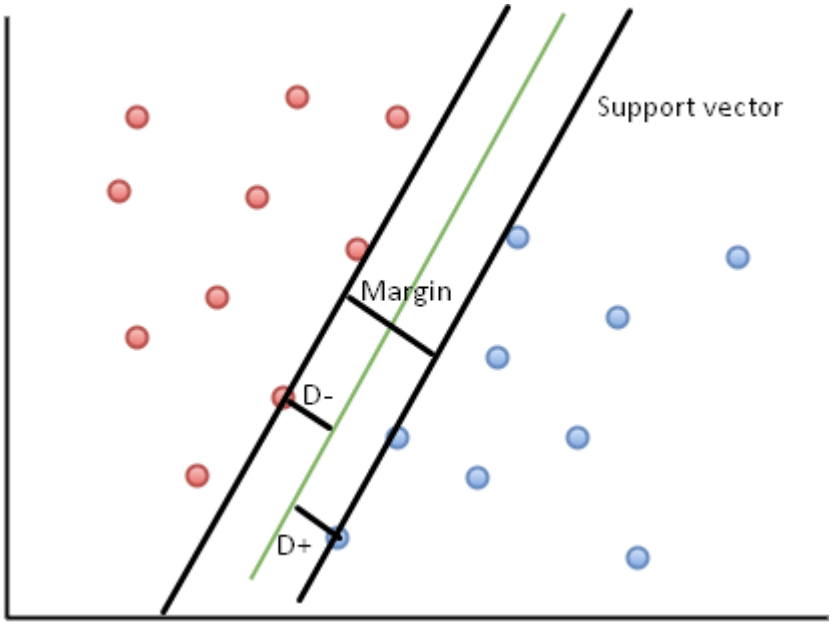
For this assignment, I used Spark to implement a Logistic Regression model. Where Linear Regression is able to predict a value based on an input value (for example size based on weight, infinite Y values), Logistic Regression can only predict if a value is True or False (Y between 0 and 1).

Instead of having a straight line, Logistic Regression uses a Sigmoid function to calculate probability, which is displayed as the following:

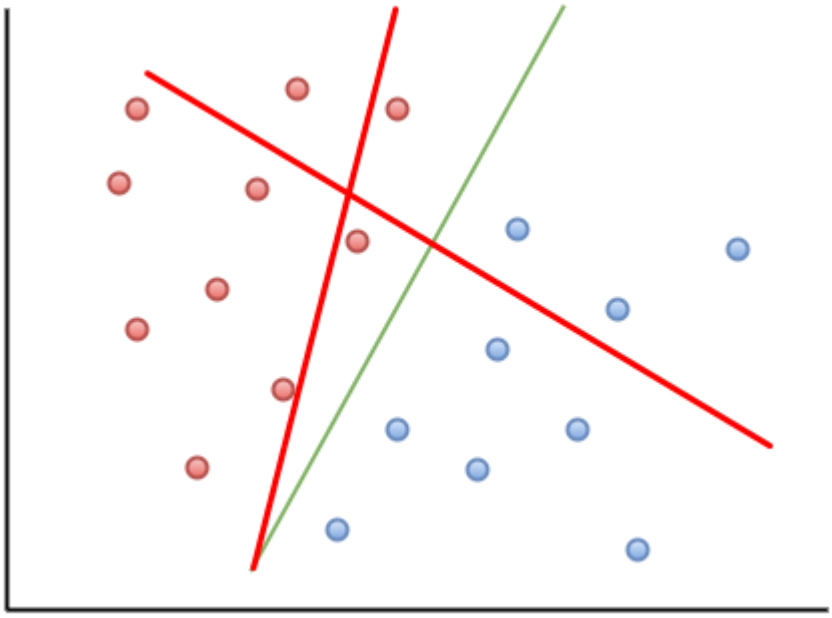


With this graph, we can predict the probability someone is obese by their weight. In the example above, if 0 is classified as not obese, and 1 as obese, there's a big chance the blue cross is classified as obese, while the chance is lower with the yellow cross. (Starmer, 2018)

For my other model I used a Support Vector Machine, which works by plotting the points created by the TF. The algorithm will then find the line that's the furthest away from both the positive (D+), as the negative (D-) points.



In the example below, the green line is the best hyperplane (Totta datalab, 2018). The red lines are not the ideal lines, since the D- and D+ distances aren't the highest they could be. If a wrong hyperplane is chosen, lots of misclassifications will happen, since the model uses the position of a point relative to the line to classify an input.



Methods

Spark Logistic Regression

To classify the reviews as positive or negative, I used a Spark Pipeline with Logistic Regression. We first create a SparkSession to retrieve data from the MongoDB:

```
def create_spark():
    return SparkSession.builder.appName("myApp") \
            .config("spark.mongodb.input.uri", "mongodb://127.0.0.1/PO2.balanced_data3") \
            .config("spark.mongodb.output.uri", "mongodb://127.0.0.1/PO2.balanced_data3") \
            .config('spark.jars.packages', 'org.mongodb.spark:mongo-spark-connector_2.11:2.3.2') \
            .getOrCreate()
```

Then load the data, drop the _id column (since it's useless for classification), and divide the SparkDF into a train and test set:

```
if spark:
    df = spark.read.format("com.mongodb.spark.sql.DefaultSource").load()
    df = df.drop('_id')
    df.printSchema()
    (train_set, test_set) = df.randomSplit([0.85, 0.15], seed=1234)
```

Were then ready to build the Pipeline, since were processing text, we will have to create values for each word before we can use them in Logistic Regression. This is done by tokenizing and hashing the words.

```
tokenizer = Tokenizer(inputCol="text", outputCol="words")
hashingTF = HashingTF(inputCol=tokenizer.getOutputCol(), outputCol="features")
lr = LogisticRegression(maxIter=10, regParam=0.001)
pipeline = Pipeline(stages=[tokenizer, hashingTF, lr])
```

To optimize our model, we use cross validation, this means Spark will try out different combinations of the given parameters, and picks out the ones that fit the data the best:

```
param_grid = ParamGridBuilder() \
    .addGrid(hashingTF.numFeatures, [10, 100, 1000, 10000]) \
    .addGrid(lr.regParam, [1, 0.1, 0.01, 0.001]) \
    .build()

cross_validation = CrossValidator(estimator=pipeline,
                                  estimatorParamMaps=param_grid,
                                  evaluator=BinaryClassificationEvaluator(),
                                  numFolds=2)
```

Were then ready to fit the data, and test the model afterwards:

```
model_cv = cross_validation.fit(train_set)
evaluator = BinaryClassificationEvaluator(rawPredictionCol="rawPrediction")

predictions = model_cv.transform(test_set)
accuracy = predictions.filter(predictions.label == predictions.prediction).count() / float(test_set.count())
roc_auc = evaluator.evaluate(predictions)

print("Accuracy Score: {0:.4f}".format(accuracy))
print("ROC-AUC: {0:.4f}".format(roc_auc))
```

In the case of these parameters the Accuracy was 92.4%, ROC-AUC 96.7%, and train time 39 seconds.

Spark Support Vector Machine

The way I trained by SVM was similar to the way I trained Logistic Regression, except for the line where I create a LogisticRegression object, I created a SupportVectorMachine object:

```
svm = LinearSVC(maxIter=10, regParam=0.1)
pipeline = Pipeline(stages=[tokenizer, hashing_tf, svm])
```

Dask vs Pandas

To test the RAM usage of Pandas and Dask, I first took the time and current memory usage:

```
memory_before = get_process_memory()
start_time = time.time()
```

```
def get_process_memory():
    process = psutil.Process(os.getpid())
    return process.memory_info().rss
```

Then I read the dataframe and Dask dataframe, and concat them 5/10 times to increase the size of the (Dask) dataframe:

```
df1 = pd.read_csv('../../../DataSet/Hotel_Reviews.csv')
pd_list = [df1, df1, df1, df1, df1]  # Concat 5 dfs
df = pd.concat(pd_list)
show_results(start_time, memory_before, 'Pandas Load')
```

```
ddf1 = dd.read_csv('../../../DataSet/Hotel_Reviews.csv')
ddf_list = [ddf1, ddf1, ddf1, ddf1, ddf1, ddf1, ddf1, ddf1, ddf1, ddf1, ddf1, ddf1, ddf1, ddf1]  # Concat 10 ddfs
ddf = dd.concat(ddf_list, axis=0, interleave_partitions=True)
show_results(start_time, memory_before, 'Dask Load')
```

After that, I measured the time/RAM usage to filter the dataframes:

```
def filter_dfs(df, ddf):
    memory_before = get_process_memory()
    start_time = time.time()
    df_filter = df[df['Total_Number_of_Reviews'] > 2000]
    show_results(start_time, memory_before, 'Pandas Filter')

    memory_after_pandas = get_process_memory()
    start_time = time.time()
    ddf_filter = ddf[ddf['Total_Number_of_Reviews'] > 2000]
    show_results(start_time, memory_after_pandas, 'Dask Filter')

    # garbage collection
    del df_filter
    del ddf_filter
```

As well as drop the duplicates and get the sums:

```
def drop_duplicates(df, ddf):
    memory_before = get_process_memory()
    start_time = time.time()
    df_dropped = df.drop_duplicates(subset='Hotel_Name')
    show_results(start_time, memory_before, 'Pandas Drop Duplicates')

    memory_after_pandas = get_process_memory()
    start_time = time.time()
    ddf_dropped = ddf.drop_duplicates(subset='Hotel_Name')
    show_results(start_time, memory_after_pandas, 'Dask Drop Duplicates')

    del df_dropped
    del ddf_dropped
    # return [df_dropped, ddf_dropped]
```

```
def get_sums(df, ddf):
    # df, ddf = drop_duplicates(df, ddf)
    memory_before = get_process_memory()
    start_time = time.time()
    df_sum = df['Total_Number_of_Reviews'].sum()
    show_results(start_time, memory_before, 'Pandas Sum (' + str(df_sum) + ')')

    memory_after_pandas = get_process_memory()
    start_time = time.time()
    ddf_sum = ddf['Total_Number_of_Reviews'].sum()
    show_results(start_time, memory_after_pandas, 'Dask Sum (' + str(ddf_sum) + ')')
```

I then saved the positive/negative rows:

```python
def save_csv():
    memory_before = get_process_memory()
    start_time = time.time()
    ddf = pd.read_csv('../../../DataSet/Hotel_Reviews.csv')
    ddf_pos = ddf.filter(['Positive_Review'])
    ddf_neg = ddf.filter(['Negative_Review'])
    ddf_pos.to_csv('./csv/Review_pos.csv')
    ddf_neg.to_csv('./csv/Review_neg.csv')
    show_results(start_time, memory_before, 'Dask Read, Drop and save to CSV')
```

Visualisation

I created a map using a Plotly Express Scatterbox map. In this map each hotel is a different dot. As can be seen in the screenshots in the Results section, I also have a filter where you're able to give a minimum/maximum rating. To get the correct data, I created a query that only returns the data you requested.

```python
fig = px.scatter_mapbox(df, lat="Lat", lon="Lng", hover_name="Hotel_Name", hover_data=["Average_Score", "Total_Number_of_Reviews"],
                        zoom=3, height=1000, size="Total_Number_of_Reviews", color="Average_Score", opacity=1, [redacted]="rdylgn")
fig.update_layout(mapbox_style="dark", mapbox_accesstoken="[redacted]")
fig.update_layout(
    autosize=True,
    margin={"r": 0, "t": 0, "l": 0, "b": 0})
return fig
```

This query also makes sure only one row is returned for each hotel, instead of one row per review.

```python
df = db.execute_query('all_data',     [
    {
        "$match": {
            "$and": [
                {
                    "Average_Score": {
                        "$gte": min_score
                    }
                },
                {
                    "Average_Score": {
                        "$lte": max_score
                    }
                }
            ]
        }
    },
    {
        "$group": {
            "_id": {
                "Lng": "$Lng",
                "Total_Number_of_Reviews": "$Total_Number_of_Reviews",
                "Lat": "$Lat",
                "Average_Score": "$Average_Score",
                "Hotel_Name": "$Hotel_Name"
            }
        }
    },
    {
        "$project": {
            "Hotel_Name": "$_id.Hotel_Name",
            "Lat": "$_id.Lat",
            "Lng": "$_id.Lng",
            "Total_Number_of_Reviews": "$_id.Total_Number_of_Reviews",
            "Average_Score": "$_id.Average_Score",
            "_id": 0
        }
    }
]
)
```

Whenever the rating slider is changed, the following callback gets executed, updating the map with the new data:

```
@app.callback(
    Output('map', 'figure'),
    [Input('rating_range', 'value')])
def update_map(value):
    return make_map(value[0], value[1])
```

To create the table with reviews, I have another callback that retrieves data about the hotel you clicked on, and displays it in a table:

```
data:
 hdf = db.execute_query('all_data', [{"$match": {'Hotel_Name': data['points'][0]['hovertext']}}])
```

```
columns = [{"name": i, "id": i} for i in hdf.columns]

return dash_table.DataTable(
    columns=columns,
    data=hdf.to_dict('records'),
    style_cell={'textAlign': 'left'},
    style_as_list_view=True
)
```

Results

Spark Logistic Regression

Number of rows	Accuracy Score	ROC-AUC	Train time
1000	87.90%	93.45%	00:00:57
5000	90.58%	95.55%	00:01:04
10000	91.84%	96.24%	00:01:00
50000	93.57%	98.05%	00:01:28
100000	94.08%	98.07%	00:01:54

Spark Support Vector Machine

Number of rows	Accuracy Score	ROC-AUC	Train time
1000	87.10%	93.82%	00:01:01
5000	91.11%	95.96%	00:01:30
10000	92.98%	97.21%	00:02:44
50000	93.73%	93.73%	00:07:31
100000	93.91%	97.86%	00:12:55

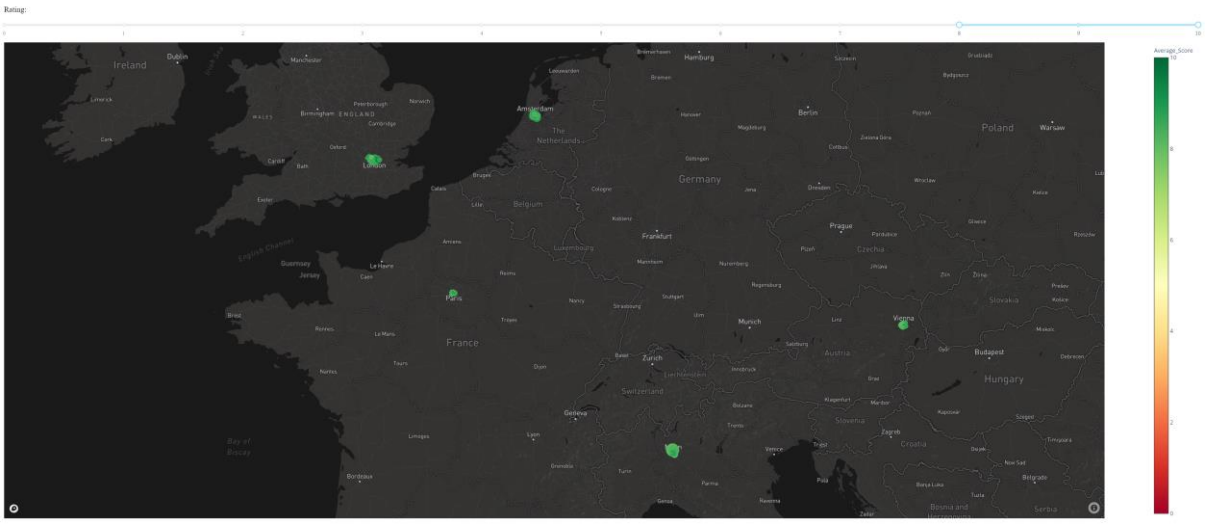
Dask vs Pandas

```
Pandas Load : memory before: 50M, after: 513M, consumed: 462M; exec time: 00:00:02
Dask Load : memory before: 458M, after: 460M, consumed: 2M; exec time: 00:00:00
Pandas Filter : memory before: 460M, after: 604M, consumed: 143M; exec time: 00:00:00
Dask Filter : memory before: 604M, after: 604M, consumed: 64K; exec time: 00:00:00
Pandas Drop Duplicates : memory before: 460M, after: 461M, consumed: 192K; exec time: 00:00:00
Dask Drop Duplicates : memory before: 461M, after: 461M, consumed: 8K; exec time: 00:00:00
Dask Read, Drop and save to CSV : memory before: 67M, after: 241M, consumed: 174M; exec time: 00:00:06
```

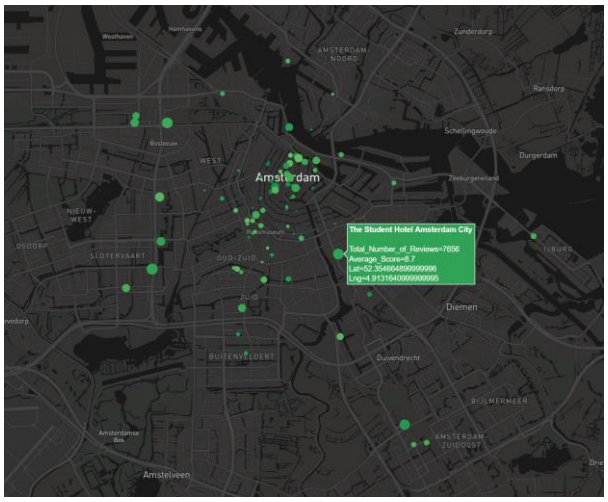
As you can see, the ram usage is way lower with Dask than Pandas, Dask had no problem dealing with dataframes of 7+ million rows, whereas Pandas already maxed out at half of that.

Visualisation

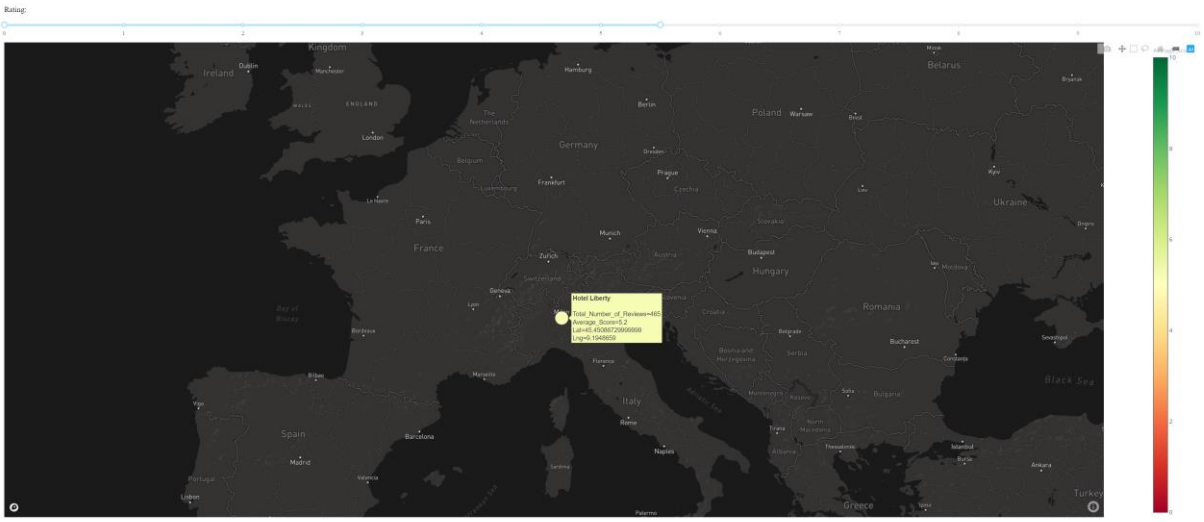
The visualization I made looks like this:



This shows all hotels in the dataset. The greener the hotel is, the higher the average rating is, the bigger the hotel's dot, the more reviews they have, as you can see in the example below, The Student Hotel Amsterdam has a rating of 8.7, which is why the circle is dark green, it also has one of the biggest circles in Amsterdam, because it has the most reviews.



The slider ontop can be used to filter the hotels based on their average rating. There's only 1 hotel in the dataset that has an unsatisfactory rating, which is Hotel Liberty in Milan:



If we click on a hotel, we can see the data about it, in this case for the hotel above:

Review	Sentiment	Reviewer_Score	Review_Date	Review_Word_Counts	Reviewer_Nationality	Total_Number_of_Reviews_Reviewer_Has_Given
Good location quiet if rooms are on the internal	1	4.6	6/4/2017	17	China	1
There is no wi fi in the rooms and no air conditi	0	4.6	6/4/2017	26	China	1
Great location with public transport links on doo	1	7.5	5/30/2017	24	United Kingdom	1
The hotel was advertised with air conditioning ho	0	7.5	5/30/2017	64	United Kingdom	1
Tram lines	1	3.3	5/29/2017	3	Ireland	2
There was no hot water in my room and staff were	0	3.3	5/29/2017	87	Ireland	2
Breakfast was good	1	6.3	5/25/2017	5	South Africa	4
NO wifi in room You have to go to portal for inte	0	6.3	5/25/2017	74	South Africa	4
The room was clean	1	4.2	5/21/2017	6	United Kingdom	1
Our bed wasn t made the next day The AC didn t wo	0	4.2	5/21/2017	55	United Kingdom	1
Location 5 mins walk to main train station Proper	1	7.5	5/19/2017	24	Singapore	12
Breakfast very simple and basic Room drapes are r	0	7.5	5/19/2017	14	Singapore	12
Suggested for non allergic people to mites fans o	1	6.3	5/12/2017	20	Italy	14
I can t see the four star rank I don t mean for t	0	6.3	5/12/2017	34	Italy	14
Hotel location is excellent Tram stop just outsid	1	6.3	5/3/2017	37	Malta	2
Hotel needs a good cleanup allover and breakfast	0	6.3	5/3/2017	18	Malta	2
Breakfast was ok	1	3.8	4/24/2017	5	Australia	3
Extremely dirty probably has not been cleaned pro	0	3.8	4/24/2017	43	Australia	3
Just the breakfast service	1	2.9	4/24/2017	6	India	3
When you pay that much of an amount with an expen	0	2.9	4/24/2017	67	India	3
Just the free breakfast	1	5	4/16/2017	6	United Kingdom	2
No WiFi at room The staff was not good when we as	0	5	4/16/2017	37	United Kingdom	2
I really unhappy with the room vessel and sink in	1	4.2	4/8/2017	33	Vietnam	10
The room not comfortable and the bathroom truly b	0	4.2	4/8/2017	11	Vietnam	10
Large room Decent breakfast	1	4.6	3/23/2017	5	United Kingdom	7
Badly in need of being refurbished Couldn t find	0	4.6	3/23/2017	38	United Kingdom	7
The stuff was extremely nice The breakfast was de	1	7.1	2/17/2017	30	Poland	1
The shower water pressure was absolutely missing	0	7.1	2/17/2017	40	Poland	1
the location	1	3.3	10/28/2016	3	Belgium	4
1 we didn t received whta we asked a room with a	0	3.3	10/28/2016	80	Belgium	4
Location	1	2.9	9/21/2016	2	Turkey	3
Rooms were extremely dirty and dark Bathroom was	0	2.9	9/21/2016	36	Turkey	3
Such friendly and helpful staff Short distance fr	1	8.8	7/24/2016	27	United Kingdom	2
Air conditioning	0	8.8	7/24/2016	3	United Kingdom	2
Location is good and you can easily walk everywhe	1	3.3	7/14/2016	13	United Kingdom	1

Conclusion

If I had to choose between the two models I trained with Spark, I'd go with Logistic Regression, since when I trained it with 100k reviews, the accuracy and ROC-AUC were quite high (94.08%/98.07%), while keeping the train time very low (00:01:54). The accuracy and ROC-AUC were similar with SVM, but the train time was 8x as shorter with Logistic Regression.

Reference list

Starmer, J. (2018, maart 5). *StatQuest: Logistic Regression*. Opgehaald van Youtube: https://youtu.be/yIYKR4sgzI8

Totta datalab. (2018, november 28). *Wat zijn support vector machines en hoe werken ze?* Opgehaald van Totta datalab: https://www.tottadatalab.nl/2018/11/28/support-vector-machines/