Predicting Parking Space Availability in Paderborn Technology Selection

Jan Lippert <ljan@mail.upb.de>

***Abstract*—The Parking Prediction system Paderborn is available at http://pppb.herokuapp.com/. It was written in Play! Scala and uses the haifengl.github.io/smile/ library. It was planned to predict the number of available parking spaces for the "Libori-Galerie" 15 minutes in advance, e.g. loading the web page in the browser will show the prediction. Currently, this functionality was not implemented yet as the quality of predictions is insufficient. Alternative models need to be investigated and implemented for the final system.**

I. Introduction

In urban areas, space and as such parking space is limited. Car owners often have to drive around to available parking spots. This leads to an unnecessary waste of time and additional air pollution.

According to multiple studies, the availability of parking data reduces the search time for a parking spot [1][2]. However, most systems only provide the current number free parking spots. In rush hours, this information can quickly get outdated and lead to driving around to find available parking spaces.

This report will describe how such a system could be implemented. Section II-A will show how the overall architecture of the system will look like. Section II-B will compare the different frameworks that were considered for the implementation.

II. Technology

A. Architecture

One important part in the choice of technology was the separation of the different concerns in the applications. There are multiple parts that play together: crawling, preprocessing, training, prediction, and finally evaluation.

Crawling has to be done on a regular basis. Therefore the chosen framework needs to support regular execution of jobs. When new entries are crawled, they are cleaned via the preprocessing module and inserted in the database. Since it may be necessary to update the model over time, the preprocessing module will also have a regular job that cleans existing entries by updating them to the most recent model.

B. Frameworks

To implement this project, Ruby on Rails, Python and Scala were considered. I used Ruby on Rails in previous projects and development is quite fast with this framework. However, I could only find a few libraries that deal with machine learning [3] [4]. Another choice of language was python. Python has quite a lot pf machine learning libraries and is also used in academics. However, I do not have much previous experience with python.

My final choice was the https://playframework.com with Scala. I did use this framework in previous projects and therefore was familiar with setting up background jobs and how to enable web access. Heroku also supports easy deployment for Play! applicationsinsert links in final report. One important factor of this choice was the type-safety of the Scala language and it's usage in machine learning and big datainsert link in final report. While the problem at hand is certainly not big data, the multitude of existing libraries helped a lot to get startedadd links to the libraries in final report.

I chose to use http://haifengl.github.io/smile/index.html, the "Statistical Machine Intelligence and Learning Engine". It was easy to use and documentation is quite extensive. In some cases, the API documentation even references the scientific papers the learning algorithm is based on.

III. Encountered Problems

A. Malfunction of the Parking Guidance System

One week after the first prototype of the crawler was online, https://www.paderborn.de/microsite/asp/parken_in_der_city/freie_Parkplaetze_neu.php was non-functional. The number of available parking spaces for the "Liborie-Galerie" was always set to 0.

At the same time, many of the other parking areas were switched to "Nicht im Parkleitsystem" (not part of the parking guidance system). Both of these issues were caused by a malfunction of the parking guidance system. Crawling was resumed normally after the parking guidance system was fixed by the provider.

On another note, string values were not expected for the Liborie-Galerie. This caused the crawler to crash on every crawl; the crawler was then adapted to be ore resilient to unexpected values: all non-integer values for free spaces will directly be dropped.

B. Heroku Free Dyno Limitations

After collecting data for some time, I did some correlation analysis on the data. I noticed that all entries had "$w_m = 1$" despite the app being online for more than a week. Further investigation showed that the data was only available for 3 different days.

The reason for this failure was the uninformed use of the Heroku Free package. Free dynos will sleep after 30 minutes

minutes of inactivity. After this was noticed, multiple services and workarounds were investigated.

However, most of the workarounds were from before 2015. In 2015, Heroku added the requirement that free dynos need to sleep 6 hours a day. Unfortunately the most promising service – http://kaffeine.herokuapp.com/ – is not functional anymore.

To keep the system running, a bash script was created which keeps the dyno awake by sending regular HEAD requests. Since the dyno is still required to sleep 6 hours a day, a sleeping time had be chosen. Luckily, the "Libori-Galerie" is closed from 2am to 8am and the dyno will rest in this period.

C. Free Database Limitations

At first, there were multiple problems in accessing the database The free tier only allows 20 concurrent connections to the Postgresql database. These connections were exhausted 5 minutes after the start of the application. This could be solved by configuring the application to restricting the number of database connections to 10 – when the application was configured to use 20 connections, problems persisted.

The next limit is in the size of the database. The free tier only allows 10000 rows. Insert actions will be disabled after the database has had more than 10000 for 7 days. This challenge will be circumvented by removing old data in regular intervals.

IV. Performance

As initially expected, the prediction accuracy is quite low. The regression tree has a mean absolute error of 106.09 when using 754 training examples and 83 test examples. The currently best result was achieved by using a Random Forest Classificator; using the same examples as before, the RFC had a mean absolute error of 64.36. This clearly shows that model and training method have to improved by a great deal before actually becoming useful.

To do so, the next part of the project will consist of trying out different models and experimenting with other features. Live predictions will be added soon with the warning that they are experimental and probably not usable yet.

References

[1] Y. Asakura and M. Kashiwadani, "Effects of parking availability information on system performance: a simulation model approach," in *Vehicle Navigation and Information Systems Conference, 1994. Proceedings., 1994*, pp. 251–254, Aug 1994.

[2] F. Caicedo, "Real-time parking information management to reduce search time, vehicle displacement and emissions," *Transportation Research Part D: Transport and Environment*, vol. 15, no. 4, pp. 228 – 234, 2010.

[3] J. Donaldson, "Machine learning on rails with ruby!," 2012. Available: https://blog.bigml.com/2012/07/06/machine-learning-on-rails-with-ruby/ [Accessed: 22.02.2017].

[4] L. Masini, "Machine learning made simple with ruby," Aug. 2015. Available: https://www.leanpanda.com/blog/2015/08/24/machine-learning-automatic-classification/ [Accessed: 22.02.2017].