Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

Shopping 2.0

Automated Shopping Lists

Bachelor Thesis

Jonas Kuratli

`kuratlij@ethz.ch`

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Philipp Brandes, Dr. Jochen Seidel
Prof. Dr. Roger Wattenhofer

September 21, 2015

Abstract

Shopping lists are still written mostly on paper, even though a high percentage of the population owns a smartphone. We aim at improving shopping experience with the use of a smartphone, more specifically by developing an app for Android smartphones which serves as a shopping list with extended functionality. The main improvement over common shopping lists is the automatic prediction of purchases by analyzing the user's previous shopping behaviour. We will evaluate the algorithm and show that it is possible to predict a part of user's shopping list, especially items bought regularly.

Contents

Abstract **i**

1 Motivation **1**

2 Related Work **2**

2.1 Recommender Systems 2

3 Predicting Shopping Lists **3**

3.1 Concept 3

3.2 Algorithm 4

4 Implementation **7**

4.1 Features 7

4.1.1 Sorting Predictions 7

4.1.2 Generic Items 10

4.1.3 Displaying Discounts 12

4.2 System Structure 12

4.3 Authentication 13

4.4 Communication 14

4.5 Client 14

4.6 Server 16

4.6.1 Data Retrieval 16

5 Tuning & Evaluation **18**

5.1 Parameter Tuning 18

5.2 Evaluation of Predictions 22

6 Future Work & Conclusion **25**

Bibliography **26**

CHAPTER 1

Motivation

Today, a big part of everyday life is made easier by the help of some electronical device. However, the common task of shopping is still largely paper-based. This leads to frustration, be it by forgetting to include that item you usually buy in your hand-written shopping list, by having to cross the whole store several times because the items on the list are noted randomly, or by having to check your supplies first to know what you are missing.

This thesis addresses these issues by offering an app for Android smartphones which serves as a regular shopping list with additional features. The app communicates with a server that excutes analysis functions and stores purchase data.

To help the user in compiling a list even though they do not know the exact supplies they have at home, the server analyzes previous purchases, identifies items bought regularly and uses this information to estimate whether the user has to buy that item because his supply is likely to run out.

To prevent unnecessary distance covered in the store, the system learns from app users by analyzing in what order they get items and infering a possible order in which items are placed in the store.

The resulting app improves shopping experience as it not only offers an easy way to keep track of the items in a paperless way but also addresses the above-mentioned issues and further offers easy extensibility.

Chapter 2

Related Work

2.1 Recommender Systems

The most widespread use of predicting shopping lists today are recommender systems. These learn from user preferences and try to find users with similar interests to recommend an item one of these users liked or bought to all of the other users. One of the most interesting applications is recommending new movies to users (e.g. done by Netflix [1]) or displaying items bought together (e.g. used by Amazon [2]).

These are, however, not suitable for the task tackled in this paper. The main conceptual difference lies in the fact that items bought in convenience stores such as Migros are bought every so often, while movies are typically watched once and items bought once, meaning they are only recommended once.

Implementation-wise, most recommender systems apply some sort of matrix factorization to the matrix containing known information such as movie ratings to obtain a guess for the missing entries. The success of this approach is based on the fact that for every row, some values are known (e.g. ratings for movies by one user) and the missing values in the row are then predicted, a task highly suitable for matrix factorization. However, the creation of a shopping list from scratch is equal to guessing the contents of a whole new row in a matrix, a task not suitable for any matrix factorization techniques.

The approach at predicting shopping lists in this paper is therefore different, and will be explained in-depth in the following chapter.

Chapter 3

Predicting Shopping Lists

To explain the prediction algorithm introduced in this section, Table 3.1 will be used. It features a collection of made-up receipts of a user who only buys the four items listed, with an x indicating that an item has been bought on a specific date at the amount noted in brackets. The algorithm tries to predict what the user should buy next (2015-03-12) and in what quantity.

Table 3.1: Example list of purchases of four different items

	Coca-Cola Zero	Ice Tea	Vollmilch	Chips
2015-02-03	x (2)	x (1)	x (2)	
2015-02-08	x (2)		x (3)	
2015-02-14	x (2)	x (1)	x (2)	x (2)
2015-02-15				x (4)
2015-02-18	x (1)			
2015-02-25	x (3)	x (1)	x (1)	
2015-03-01	x (1)		x (2)	
2015-03-07	x (3)	x (1)	x (3)	x (5)
2015-03-12	?	?	?	?

3.1 Concept

The main idea of this algorithm is to determine how regularly each item is bought and then check whether enough time has passed since the last purchase. To obtain this information, for each item the average a of the time difference between every pair of receipts containing that item is calculated. Next, the algorithm determines the time difference t between the time the function is called and the last known time the user bought the item (date & time). To check whether enough time has passed, the algorithm calculates the difference ratio r given by the following division:

$$r = \frac{t}{a}$$

It can be seen that r will increase linearly with the time passed since the last purchase, and items should most likely be proposed at a value around 1. For the example data introduced before, the values of a, t, and r can be seen in Table 3.2

Table 3.2: Values for a, t, and r

	Coca-Cola Zero	Ice Tea	Vollmilch	Chips
a	128	256	162	228
t	120	120	120	120
r	0.94	0.47	0.74	0.53

3.2 Algorithm

As a first step, the algorithm iterates over all receipts and excludes all items that were bought less than four times by the user. This adds a certain significance to the regularity of purchases of that item. On the downside, items the user just started buying regularly will only show up after some purchases.

Items with similar names are now merged into a single, generic item (For implementation details, see Section 4.1.2). This is to improve the quality of the data used, as for example two items called *Bio Vollmilch* and *Milch 1L* denote the same generic item *Milch*. Since the user selects one of the two by personal preference, it suffices to propose the generic item *Milch* and leave the choice of what item to buy up to the user. The mapping aids in deriving this generic proposal by checking each item's name against the database and, if the database holds an entry for a mapping of the item to a more generic item, applying said mapping.

Next, the algorithm iterates over the items determined before and then, for each of these items, iterates over all receipts containing the item, making note of the average time difference in hours between every pair of subsequent receipts, which yields the average time difference a. Additionally, the standard deviation σ of a is calculated. In the same iteration, the algorithm notes the average amount each item is bought at. This gives an intermediate result containing all items not excluded in the first step, along with values for r, a and σ for every item.

At this point a first threshold is applied. If the standard deviation is more than twice as high than the average a, the item will not be predicted. This rules out accidental predictions of items users do not seem to buy regularly, but are rather bought in a random fashion (e.g. for a party), and would therefore be hard to predict correctly. As can be seen in Table 3.3, none of the four example items will be excluded by this threshold.

Table 3.3: Values for a and σ for the four example items. All items are kept, as the standard deviation σ of the time difference between purchases is less than twice the average of said time difference for all items.

	Coca-Cola Zero	Ice Tea	Vollmilch	Chips
a	128	256	162	228
σ	29.07	13.86	71.67	288.50
a/σ	0.23	0.05	0.44	1.27

Finally, the before-mentioned ratio r is calculated and compared to the two input parameters r_{min} and r_{max}, which are bordering values for r. The lower limit's necessity is obvious, the upper limit however is less so. It is used to prevent items from being predicted over and over again. This is due to the fact that an item not bought for a long time (e.g. if the user does not buy it any more) will have a constantly increasing value of r. Setting an upper bound will, after this bound was crossed, exclude the item from all future predictions. The process of tuning these parameters is explained in Section 5.1.

If r lies between or is equal to one of the two values, the item is added to the result list. The amount the user should buy is calculated by multiplying the average amount the item was bought at by the value of r, rounding to the nearest integer.

After the algorithm checked all items a shopping list like in Table 3.4 is obtained, containing all items proposed to the user and the quantity in which they will be proposed.

Table 3.4: Decision on which items to propose in what quantity (assuming $r_{min} = 0.8$ and $r_{max} = 1.8$)

	Coca-Cola Zero	Ice Tea	Vollmilch	Chips
r	0.94	0.47	0.74	0.53
Average amount	2	1	2.17	3.67
Buy item (Y/N)	Y	N	N	N
Predicted amount	1.88 (2)	-	-	-

In a final step, the algorithm determines the order in which the items are likely to be placed in the store. This is done by learning from previous purchases. The algorithm analyzes in what order users check items and determines an approximate order in which the user will likely find the items in the store.

Implementation details are described in Section 4.1.1.

Pseudocode for this algorithm can be seen in Algorithm 1.

Algorithm 1 Basic Prediction

```
1: function PREDICT($r_{min}$, $r_{max}$)
2:     $items\_to\_check \leftarrow$ Items bought at least 4 times
3:     $generic\_items \leftarrow$ MERGE_SIMILAR_ITEMS($items\_to\_check$)
4:     $all\_receipts \leftarrow$ All receipts present in DB for user
5:     $t \leftarrow$ Time passed since last purchase
6:     $predictions \leftarrow [\,]$
7:     for $item$ in $items\_to\_check$: do
8:         $a \leftarrow$ Average time diff. between two purchases of $item$
9:         $\sigma \leftarrow$ Standard deviation of $a$
10:        $a_{amount} \leftarrow$ Average amount the item was bought at before
11:        if $\sigma > 2 \cdot a$ then
12:            $r \leftarrow t/a$
13:            if $r_{min} \leq r \leq r_{max}$ then
14:                $p\_amount \leftarrow a_{amount}/r$
15:                Append ($item$, $p\_amount$) to $predictions$
16:            end if
17:        end if
18:    end for
19:    $sorted\_predictions \leftarrow$ SORT_IN_STORE_ORDER($predictions$)
20:    return $sorted\_predictions$
21: end function
```

Chapter 4

Implementation

4.1 Features

4.1.1 Sorting Predictions

As briefly mentioned in Section 3.2, predictions proposed to the user are in an order that is likely to be close to the order in which the user will find the items proposed to them in the store. This feature is based on the assumption that the layout in all stores is approximately the same.

In order to sort predictions, the system learns from previous purchases. For every new purchase stored in the database (independent of the user), the system looks at the order in which the items were checked by the user and, for every pair of items, determines which of the two was checked later. Consider the four items in Table 4.1. The numbers in the table represent the order in which items were checked during different purchases.

Table 4.1: Numerical order in which items were checked during purchases, i.e. an entry of 1 for a purchase means that the corresponding item was checked first during that purchase (- = not bought)

Date	Tomaten	Coca-Cola	Milch	Brot
2015-02-03	1	4	2	3
2015-02-08	1	-	3	2
2015-02-14	1	2	3	-
2015-02-15	2	3	-	1
2015-02-18	-	2	1	-
2015-02-25	2	-	1	-
2015-03-01	2	1	-	3
2015-03-07	1	-	3	2
2015-03-12	1	4	2	3

For every purchase and every pair of items the item with the higher associated number in the ordering was checked later. This information is stored in a matrix M, as seen in Table 4.2, where for each pair $\{i, j\}$, the entry $M_{i,j}$ states the amount of times the item represented by row i was checked earlier than the item represented by column j.

Items that are checked as one of the last items in most purchases will have low entries in their respective column, while items that are checked at the beginning of purchases will have high entries.

Table 4.2: Matrix M: Stores how often items were checked later than other items. Each entry $M_{i,j}$ denotes the amount of purchases so far where the item of row i was checked later than the item of column j. Example: The item *Coca-Cola* was checked after the item *Tomaten* during 4 different purchases before.

	Tomaten	Coca-Cola	Milch	Brot
Tomaten	-	1	1	1
Coca-Cola	4	-	3	3
Milch	5	1	-	2
Brot	5	1	2	-

Next, for the set of items that should be ordered, the corresponding entries in the above-mentioned matrix are extracted into a new matrix N. For each row the entries are then scaled proportionally such that they scale up to 1, i.e. each entry $N_{i,j}$ is given by

$$N_{i,j} = \frac{M_{i,j}}{\sum\limits_{s \in S} M_{i,s}},$$

where S is the set of items to be ordered.

Items that were bought at the beginning in most purchases will, as before, have relatively high entries in their respective column.

Keeping the above conclusion in mind, the matrix is now represented as a state diagram (Figure 4.1), where the items denote the states (abbreviated by their first letter) and the entries in N denote the probabilities of state changes, i.e. $N_{i,j}$ denotes the probability that after state i, the system will be in j.

Table 4.3: Probability of state changes

	Tomaten	Coca-Cola	Milch	Brot
Tomaten	-	0.33	0.33	0.34
Coca-Cola	0.4	-	0.3	0.3
Milch	0.625	0.125	-	0.25
Brot	0.625	0.125	0.25	-

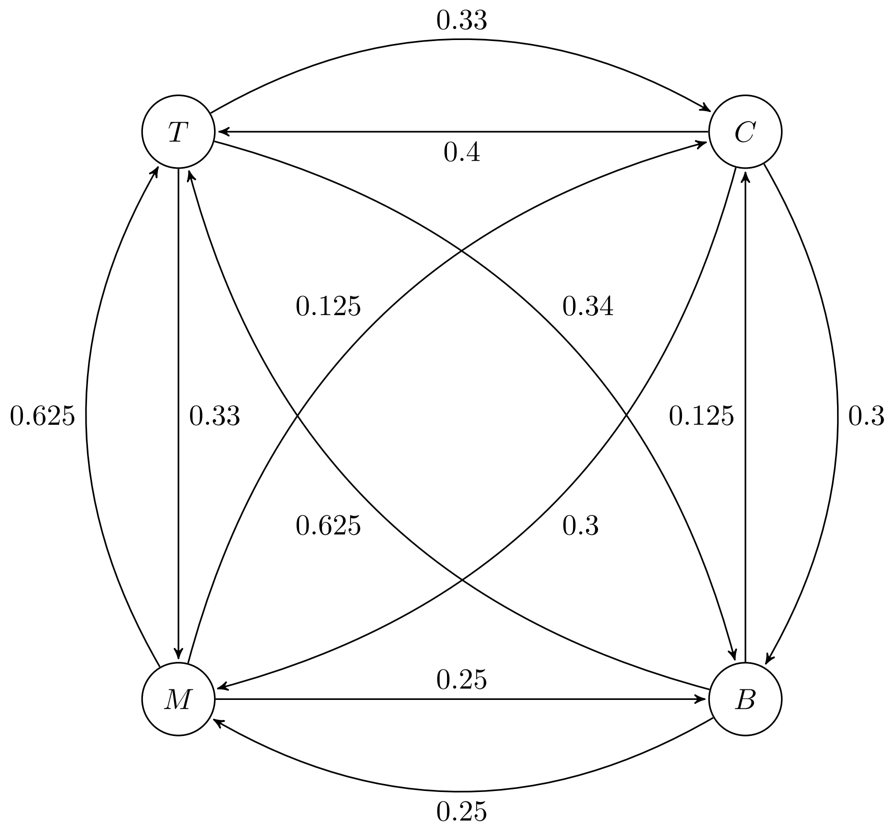


Figure 4.1: State Diagram showing the state change probabilities

As stated before, an item checked late in many purchases will have low entries in its respective column, therefore implying that the edges going into its state will be of lower weight than ingoing edges of other states. This item should at the same time show up late in the ordering that the algorithm determines. Items that are represented by states with ingoing edges of low weight should thus be placed near the end of the ordered list. In turn, items represented by a state with ingoing edges of high weight should be placed at the beginning of the list.

In order to obtain a measure for how high each state's ingoing weights are in comparison to the other states and in what order items should thus be placed, a Markov Chain is used to model the states and probabilities of state changes. For one, the state diagram obtained is a representation of a Markov Chain and therefore no additional work is required for creation. Additionally, the given restriction on the input data (all rows sum up to 1) leads to the fact that one can deduce steady-state probabilities of the Markov Chain obtained.

In a Markov Chain of the format explained above, the steady-state probabilities are denoted by a vector which, for each state i, denotes the probability that at a random time, the system will be in that state. It can be verified that a state with relatively high-weighted incoming edges is likely to have a high steady-state probability compared to the other states. Therefore, items checked early, whose states have, as explained above, high-weighted incoming edges, will have a higher steady-state probability and should be proposed early.

The algorithm thus orders the items by creating a Markov Chain as explained above, calculating the steady-state probabilities, ordering them by value and mapping them back to their respective items as seen in Table 4.4.

Table 4.4: Steady-state probability (SSP) for example data and the order in which items should be placed (*Rank* 1 through 4). The state T corresponding to the item $Tomaten$ has the highest steady-state probability and the item is therefore placed at the top of the list (*Rank* 1)

	SSP	*Rank*
Tomaten	0.360	1
Coca-Cola	0.177	4
Milch	0.230	3
Brot	0.233	2

4.1.2 Generic Items

As briefly mentioned in Section 3.2, users who need a specific generic item (e.g. $Milch$) do not always buy the exact same article in the store. The items *Bio Vollmilch* and *Milch 1L*, for example, both represent an item $Milch$ but have different names. Further, there may be users that do not always purchase items in the same size. *Ice Tea 1L* and *Ice Tea 2L*, for example, both denote the same item (*Ice Tea*) but in different sizes. Nevertheless, users should be sent predictions where the purchases of the two items were taken into account together.

To visualize the feature, consider the mock-up overview of purchases of a user for the four items mentioned above shown in Table 4.5.

Table 4.5: Example list of purchases of four different items

	Bio Vollmilch	M-Budget Milch	Ice Tea 1L	Ice Tea 2L
2015-02-03	x (2)			x (1)
2015-02-08	x (2)		x (2)	
2015-02-14		x (2)	x (1)	
2015-02-15				x (1)
2015-02-18	x (1)			
2015-02-25		x (3)	x (2)	
2015-03-01		x (1)	x (2)	
2015-03-07	x (2)			x (1)
2015-03-12	?	?	?	?

If the prediction algorithm was applied to the given table, it would yield the values in Table 4.6 for the difference ratio r and lead to an empty list of predictions. This is clearly not the desired result, as one can see that the user purchases the generic items *Milch* and *Ice Tea* regularly, but in different versions.

Table 4.6: The four items have low values of r. These would, depending on the lower bound for r, lead to an empty or only half-full prediction list, despite the fact that common sense dictates that both items should be proposed.

	Bio Vollmilch	M-Budget Milch	Ice Tea 1L	Ice Tea 2L
a	256	180	168	384
t	120	120	120	120
r	0.47	0.67	0.71	0.31
Buy item (Y/N)	N	N	N	N

The solution approach used is a mapping of item names given by receipts or by the user to more generic names (e.g. mapping the two *ICE TEA* items of different sizes to a generic item *ICE TEA* (with no notion of size)). This mapping is applied to all items a user has bought prior to calling the prediction algorithm from Section 3.2. The algorithm will then run on fewer items with more purchase data per item and will deliver more accurate predictions.

To determine this mapping, it proved to be easiest to run through the list of items and determine meaningful generic names by hand. These are then checked against all item names and, where one of the manually determined names is a substring of the item's name, a new entry in the table of mappings is created.

The above-mentioned mapping is implemented as a table with four columns, as shown in the example in Table 4.7. The columns *old_name* and *new_name* denote the old and the new, more generic name of the item, while *new_id* is, for all sets of items mapping to the same generic name, the id of one of the items in the set.

Table 4.7: Excerpt of the table containing all mappings

old_id	*new_id*	*old_name*	*new_name*
31	31	Bio Vollmilch	MILCH
41	31	M-Budget Milch	MILCH
59	59	Ice Tea 1L	ICE TEA
26	59	Ice Tea 2L	ICE TEA

The application of this mapping leads to a merge of the four initial items into just two items (*MILCH* and *ICE TEA*), which hold the purchase data of all items that were mapped to them. This leads to more exact knowledge of the user's shopping behaviour, as more data is available per item.

4.1.3 Displaying Discounts

The items that are currently discounted are obtained by using Scrapy [3] to crawl the website displaying all discounts. Items are loaded by emulating AJAX requests and extracting the necessary information from the responses. Next, each item is checked against the list of known generic items and, if a mapping exists, the item is renamed to the generic item's name. The item is then stored in the database.

When a user requests predictions, the prediction algorithm is run as usual to obtain a shopping list. For every item in the list, the known discounts are checked and if a discount exists for that item, this information is added to the list.

On the client side, when displaying the predictions, every item that is marked as discounted will feature a small icon (c.f. Figure 4.2).

4.2 System Structure

The system is implemented as a Client-Server structure (c.f. Figure 4.3). The client is an Android App running on the user's smartphone, while the server is based on the Django Web Framework [4]. Communication is based on JSON [5] and a REST [6] interface on the server side.

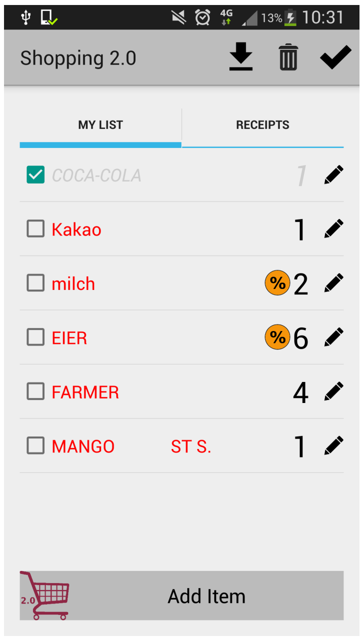


Figure 4.2: Discounts are indicated by an icon with a percentage sign

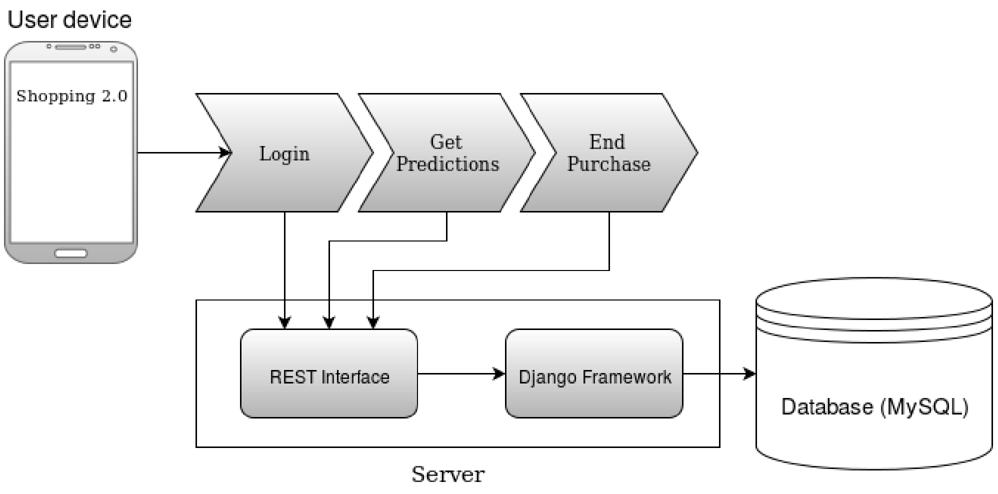


Figure 4.3: Architecture diagram of the System

4.3 Authentication

Users authenticate themselves with their Google Accounts. This is achieved by using Google Sign-In for Android [7]. The process is similar to the OAuth [8] authentication process. The app asks the user to log in with their Google account and, if consent is given, obtains a token from Google's OAuth servers. This token is then sent to the server, which verifies its validity. Next the necessary login information (mail address & unique id) is extracted and checked against the database.

If a user with the given data exists, a unique session id is created by the server and sent back to the user. All subsequent requests include said session id and are authenticated by it.

If no user is found, a new user is created and the App asks the user to enter the login credentials for his Cumulus account. This data is then sent to the server and stored. As above, a session id is created and sent to the user to finish the login process.

4.4 Communication

All communication between client and server is stateless in the sense that the server does not keep track of what request is received by which client. Requests are sent to the server's REST interface, with all additional data added in JSON format.

On the client side, requests are sent asynchronously to the corresponding interface, however the client does not block while waiting for a response. The response is, as soon as received, handled by a callback function.

Furthermore, communication is kept to a minimum. The only app functions triggering a request are logging in, requesting predictions, loading old receipts, and uploading shopping lists at the end of a purchase. All other functions are executed locally on the client side, and results are stored on the client's device.

4.5 Client

As mentioned before, the client is an Android App running on the user's smartphone. The client functionality consists mainly of offering a shopping list to the user. All calculations are done on the server, the app only displays the resulting predictions.

After successfully logging in, the user is presented a screen similar to the one depicted in Figure 4.4. Items displayed in red font are predictions loaded from the server, items in black are created by the user or predicted items that have been edited.

Predictions are loaded automatically when logging in, but can also be requested manually by pressing the button in the title bar. They are received in

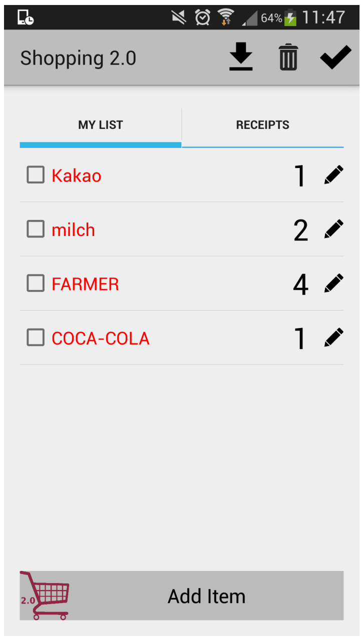


Figure 4.4: Screen presented to the user after Login

JSON format, converted into objects of the class *Item* and displayed in the list.

Any changes to the list (addition, editing or deletion of items) is stored locally on the user's device, but not uploaded to the server.

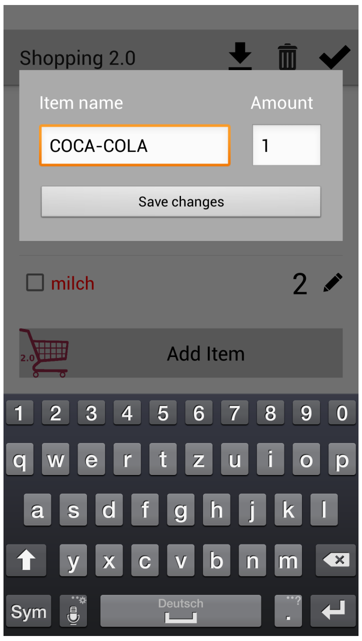
Items are checked or unchecked by either pressing the checkbox or the line the item is displayed on. Subfigure 4.5b shows a list with two items checked.

Editing items is achieved by pressing the pencil icon of the respective item (c.f. Figure 4.5), changing name and amount and confirming the change. The change is then reflected on the list.

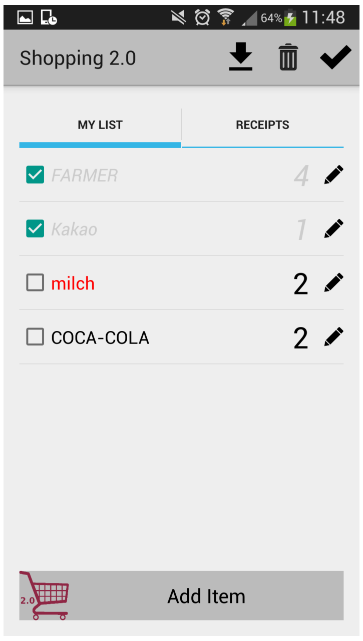
The list is sorted by first listing all checked items and then displaying the rest in the order they are proposed or were added.

The tab *Receipts* shows an overview over the user's previous purchases. Detailed information such as the items bought and the amount they were bought at is available by pressing the corresponding receipt (c.f. Figure 4.6). It is further possible to add previously bought items to the list by pressing the button next to it that shows a + sign.

Upon completion of the purchase, the user confirms this by pressing the black checkmark icon. The list is converted into JSON format and sent to the server. The server stores the data and then sends an acknowledgment to the App. As soon as that is received, the list is cleared and the user informed about the successful upload of the list.



(a) Edit Dialog



(b) List after edit

Figure 4.5: Editing items

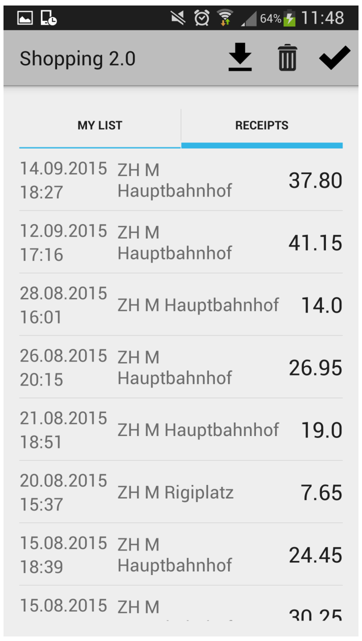
4.6 Server

The server is implemented in Python. The Django Web Framework [4] is used for generation and management of the server logic and the data models, as well as for communication (by offering REST interfaces). The requests are routed through the views of the interface to the respective functions.

4.6.1 Data Retrieval

All data about user purchases is obtained by downloading the receipts associated with the user's Cumulus account and stored by Migros. Migros offers an online platform [9] which displays the receipts of all purchases where the user presented his Cumulus card, along with what items were purchased at what amount and what price, as well as date, time and store where the purchase took place.

Whenever a user is authenticated as described in Section 4.3, the server gets and stores any new receipts added to the user's Cumulus account. This is achieved by opening a browser session using the python package *mechanize* [10], authenticating said session with the credentials stored in the database and emulating JavaScript to display and download any receipts not stored yet.



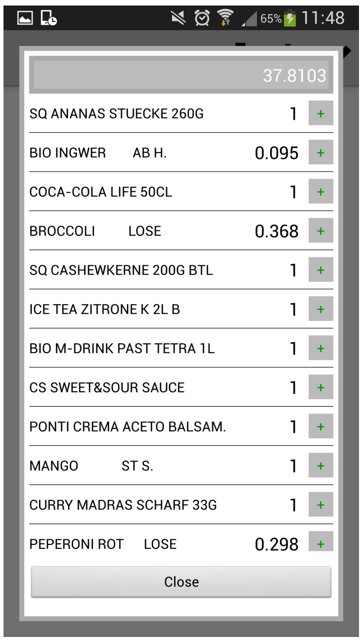
11:48

Shopping 2.0

MY LIST | RECEIPTS

14.09.2015 18:27	ZH M Hauptbahnhof	37.80
12.09.2015 17:16	ZH M Hauptbahnhof	41.15
28.08.2015 16:01	ZH M Hauptbahnhof	14.0
26.08.2015 20:15	ZH M Hauptbahnhof	26.95
21.08.2015 18:51	ZH M Hauptbahnhof	19.0
20.08.2015 15:37	ZH M Rigiplatz	7.65
15.08.2015 18:39	ZH M Hauptbahnhof	24.45
15.08.2015	ZH M	30.25

(a) Edit Dialog



11:48

37.8103

SQ ANANAS STUECKE 260G	1	+
BIO INGWER AB H.	0.095	+
COCA-COLA LIFE 50CL	1	+
BROCCOLI LOSE	0.368	+
SQ CASHEWKERNE 200G BTL	1	+
ICE TEA ZITRONE K 2L B	1	+
BIO M-DRINK PAST TETRA 1L	1	+
CS SWEET&SOUR SAUCE	1	+
PONTI CREMA ACETO BALSAM.	1	+
MANGO ST S.	1	+
CURRY MADRAS SCHARF 33G	1	+
PEPERONI ROT LOSE	0.298	+

Close

(b) List after edit

Figure 4.6: Editing items

Chapter 5

Tuning & Evaluation

5.1 Parameter Tuning

The bordering parameters r_{min} and r_{max} of the prediction algorithm are determined by evaluating predictions for different values. This is done by taking a subset of a user's receipts stored in the database and, for different parameter values, predicting the items appearing on the first half of the remaining receipts.

Note that this split is necessary, since the second half will be used as a test set to evaluate predictions (and therefore also the parameters tuned in this section), and should therefore differ from the test set used in this section. The predictions are then evaluated to determine the best values for the parameters. For this evaluation, the following three scores are taken into consideration:

1. Average amount of items predicted: How many items the predicted shopping lists contained on average. This score is necessary to prevent selection of parameters which would lead to lists of predictions with either too few or too many items.
2. Correct predictions: How many of the predicted items appeared on the actual receipt
3. Percentage of correct predictions: Percentage of correct predictions to average amount of items predicted

To determine r_{min}, r_{max} is set to a reasonable value of 1.8. Then, the evaluation function runs over the test set mentioned above for values of r_{min} ranging from 0.3 to 1.2, in steps of 0.1.

Figures 5.1 and 5.2 display the three scores for the range under test. Since the percentage of predictions that are correct stays almost the same, it seems best to set r_{min} to a value that leads to a reasonable amount of predictions, since neither two nor nine predictions appear good for usability. The value of r_{min} is therefore set to 0.7.

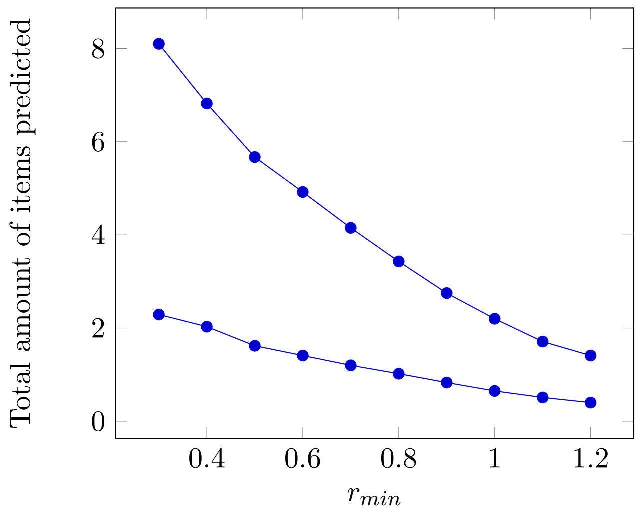


Figure 5.1: The upper line represents the average amount of items predicted, the bottom line shows the amount of correct predictions, i.e. items that were predicted and bought by the user. It can be seen that both the average amount of items predicted and the amount of correct predictions decreases with increasing values of t_{min}.

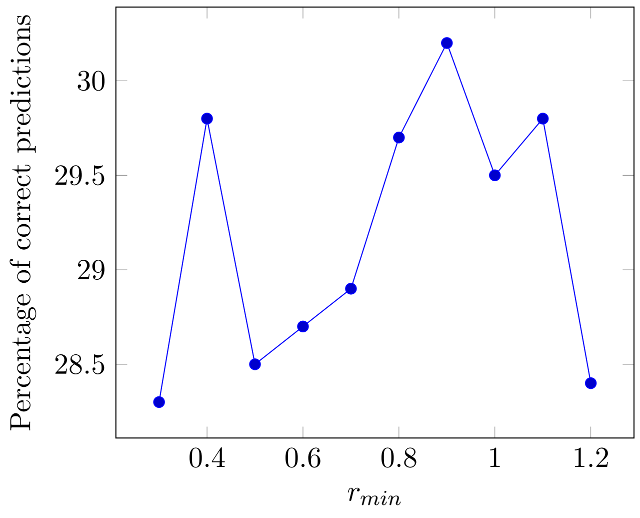


Figure 5.2: The percentage of correct predictions varies little for all values of r_{min} under test.

Determining a value for r_{max} is more complicated. A static upper bound is not the ideal solution, as items purchased very often (e.g. every three days) would cross the upper bound after a relatively short amount of time and would from then on not be proposed again.

Consider an example similar to the one used in Section 1, reduced to one item (*ICE TEA*), in Table 5.1. The item, as shown in Table 5.2 is bought every three days. Such a small difference between purchases leads to the difference ratio r growing quickly in few days time (e.g. $r = 2.28$ after only seven days). Any static upper bound on r in a sensible range will therefore be crossed in a relatively small period of time, meaning that items bought often will not be predicted anymore after only a few days.

It is also not a good option to increase the upper bound indefinitely, as for example an upper bound of $r_{max} = 3.0$ applied to an item every 20 days means that the item will be predicted until 60 days have passed since the last purchase, which decreases user-friendliness.

Table 5.1: Example list of regular purchases the item *ICE TEA*

	ICE TEA
2015-02-03	x (2)
2015-02-06	x (2)
2015-02-08	x (3)
2015-02-12	x (2)
2015-02-15	x (2)
2015-02-22	?

Table 5.2: The value of r increases quickly and crosses sensible upper bounds early

	ICE TEA
a	72
t	164
r	2.28

To fix this issue, the upper bound is implemented as a dynamic bound that is different for each item and is inversely proportional to the value of a for the item under test. This ensures that items with a low a will have a higher upper bound r_{max} on r, i.e. they will be predicted for a longer time.

We are therefore looking for an equation for the upper bound r_{max} depending on the average time difference a introduced in Section 3.2 of the form

$$r_{max} = \frac{c}{a^n} + b,$$

with three unknown parameters c, b, and n.

To determine good values for these parameters, n and c, b are in turn fixed and the other paramater(s) are optimized with regard to the percentage of correct predictions and the average amount of predictions. To visualize this process, Figures 5.3 and 5.4 show the resulting scores of an optimization step for n with fixed $c = 18$ and $b = 0.8$.

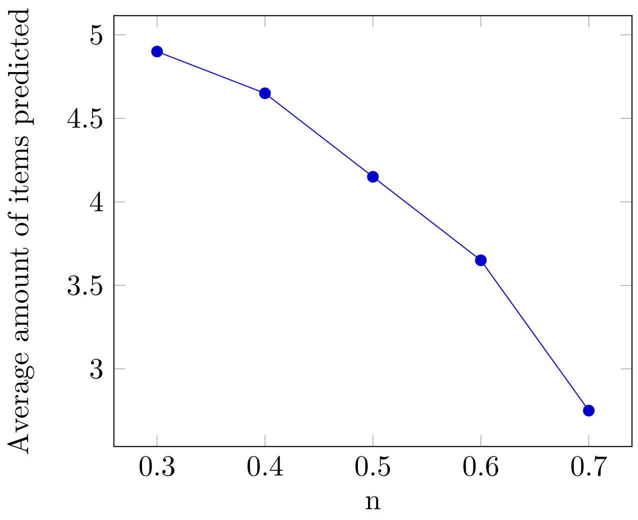


Figure 5.3: The average amount of items predicted decreases for increasing values of n for given parameters $c = 18$ and $b = 0.8$

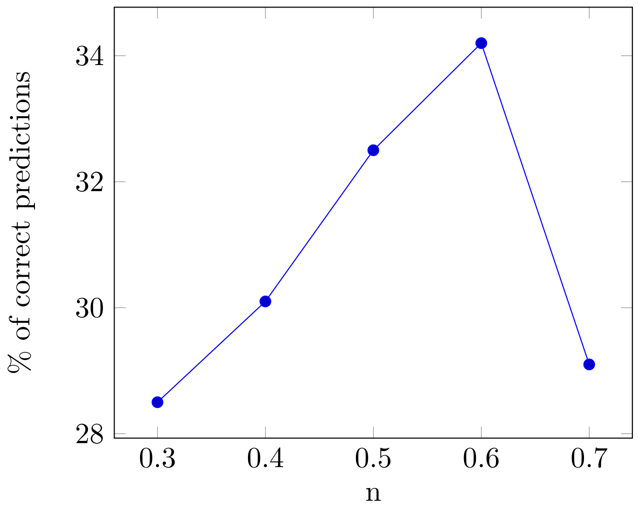


Figure 5.4: The percentage of predicted items that was bought reaches a maximum for $n = 0.6$

The values obtained in the end were $c = 17$, $n = 0.6$ and $b = 1$, leading to the following equation for determining the upper bound r_{max}:

$$r_{max} = \frac{17}{a^{0.6}} + 1.0,$$

where a is the average time difference between purchases of the item.

Adding this formula to the existing prediction algorithm and applying it to the example data results in a higher upper bound for the item *ICE TEA* and leads to the item being predicted.

The effectiveness of this dynamic bound is evaluated in Section 5.2.

5.2 Evaluation of Predictions

The prediction algorithm is evaluated by first running it without any features tuned on (i.e. evaluating the concept used), and then turning on features one by one. This approach helps in visualizing the effect the features described in Section 4.1 have on the predictions and thus also helps evaluating them and justifying their use in the algorithm

Evaluation of the predictions is done by taking all but the last 100 receipts of a user as a base for predictions and predicting each of the 100 receipts. The two scores used to evaluate the correctness of the predictions are:

1. The percentage of predicted items that showed up on the actual receipt
2. The amount of items that were bought by the user but not predicted by the algorithm.

The following four evaluations are run as described and the two scores are obtained for each one of them:

- Evaluation 1: The base algorithm runs without merging items into generic items and by using a static upper bound instead of a dynamic one.
- Evaluation 2: The dynamic upper bound as determined in Section 5.1 is added to the algorithm.
- Evaluation 3: The merging of items into generic items is turned on again, but the algorithm runs with a static upper bound.
- Evaluation 4: Both merging of items and the dynamic upper bound are turned on.

In Figures 5.5 and 5.6, the two scores are displayed for each of the four evaluations described above.

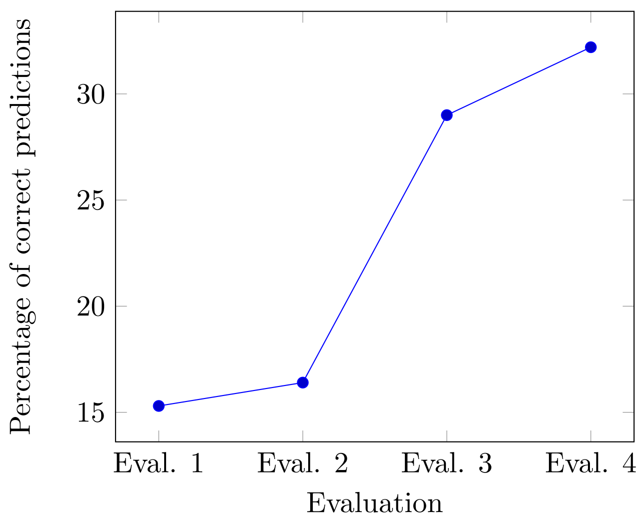


Figure 5.5: Both features lead to an increase of the percentage of correct predictions

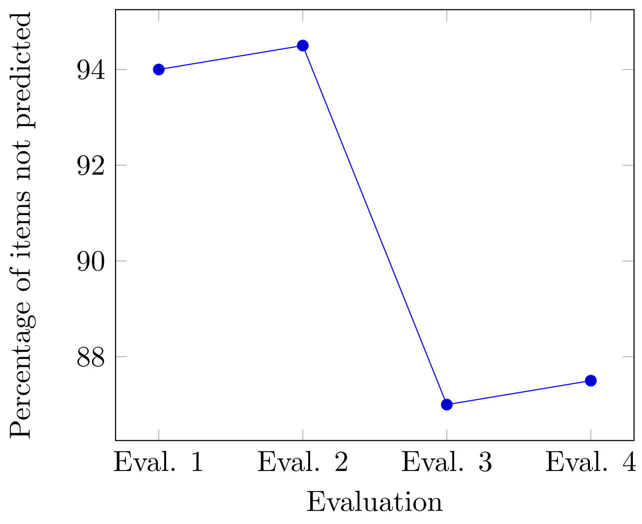


Figure 5.6: The percentage of bought items that were not predicted decreases when merging items

The results imply that merging items has a significant positive effect on both the percentage of correct predictions, which almost doubled (Evaluation 1 vs. Evaluation 3), and the percentage of bought items that were not predicted. Thus, the feature's inclusion in the algorithm is justified.

Analyzing the effect of the dynamic upper bound leads to an interesting notion. The percentage of correct predictions increases as desired, but the percentage of bought items that were not predicted increases as well. This is a result of the dynamic upper bound leading to less predictions per purchase (3.7 instead of 4.3 predictions per purchase on average). As the increase in the percentage of correct predictions is higher than the increase in items bought that were not predicted, the inclusion of this feature in the predictions algorithm is justified as well.

Chapter 6

Future Work & Conclusion

The prediction algorithm as well as the different features shown are by themselves a solid result, but can also be enriched further.

First, in Section 5.1, the value of the lower bound for the prediction algorithm was chosen by aiming at an average of four items per prediction. This value could also be determined dynamically for each user, for example by analyzing the amount of items the user buys on average per purchase. Especially for users with few, but big purchases, a higher amount of predictions may be desirable.

Next, it would be interesting to come up with a different approach at predicting items and evaluate it against the prediction algorithm presented in this paper. Alternatively, the algorithm could be extended with more features that would improve the predictions.

Further, the app only works when the user is connected to the internet. However not all stores offer good coverage. One could therefore extend the app by, for example, loading predictions and old receipts in a background service.

Finally, the current version of the app only collects the order in which the items were checked in, meaning a lot of usage data is not analyzed. Surely, additional functionality and possibly improved predictions could be derived from collecting more data.

All in all, the paper presents a solid structure for improving shopping experience by offering a digital shopping list with prediction of items and other features helping the user. These have been shown to have a significant positive effect. The presented solution could be improved easily by adding additional features or addressing some of the issues mentioned above.

Bibliography

[1] : Netflix recommendations. http://techblog.netflix.com/2012/04/netflix-recommendations-beyond-5-stars.html Accessed: 2015-09-20.

[2] Linden, G., Smith, B., York, J.: Amazon.com recommendations. In: IEEE Internet Computing January/February 2003. IEEE (2003) 76–80

[3] : Scrapy. http://scrapy.org/ Accessed: 2015-09-19.

[4] : Django Web Framework. https://www.djangoproject.com/ Accessed: 2015-09-20.

[5] : JavaScript Object Notation. http://www.json.org/ Accessed: 2015-09-20.

[6] : Representational State Transfer. https://de.wikipedia.org/wiki/Representational_State_Transfer Accessed: 2015-09-20.

[7] Google: Google Sign-In for Android. https://developers.google.com/identity/sign-in/android/ Accessed: 2015-09-16.

[8] IETF OAuth WG: Oauth 2.0. http://oauth.net/2/ Accessed: 2015-09-15.

[9] : Migros Cumulus. https://www.migros.ch/cumulus/de.html Accessed: 2015-09-19.

[10] Python Software Foundation: Mechanize 0.2.5. https://pypi.python.org/pypi/mechanize/ Accessed: 2015-09-15.