Skillability

HarvardX - PH125.9x Data Science Capstone

Giovanni Azua Garcia - giovanni.azua@outlook.com

Januar 06, 2020

Contents

List of Tables

List of Figures

Licenses, Terms of Service, Privacy Policy, and Disclaimer

Data license: The dataset files downloaded, extracted, transformed, assembled or processed in any form as part of this project are derived from Stack Overflow[1]'s and retain their original license: Attribution-Share Alike 4.0 International (CC BY-SA 4.0)[2].

Code license: The code delivered as part of this project is licensed under the GNU Affero General Public License (AGPL v3)[3].

Terms of Service and Privacy Policy: In this project we use anonymised user data and the Google Maps API therefore we're also bound by Google's Terms of Service and Google's Privacy Policy.

DISCLAIMER Third-Party Trademark Notice: All third-party trademarks referenced in this report (i.e. tags or skills), whether in logo form, name form or product form, or otherwise remain the property of their respective owners, and are used here only to refer the credentials and proficiency of the Stack Overflow users in using the technology, products or supporting solutions. The use of these trademarks in no way indicates any relationship between the author of this report and their respective owners. The description of the capabilities regarding any of the listed trademarks does not imply any relationship, affiliation, sponsorship or endorsement and reference to those shall be considered nominative fair use under the trademark law.

[1] https://archive.org/details/stackexchange
[2] https://creativecommons.org/licenses/by-sa/4.0/
[3] https://www.gnu.org/licenses/agpl-3.0.en.html

Introduction

If you ever programmed, faced a technical question and "Googled it", it's needless to say that you have already probably landed in the Stack Overflow[4] site. Stack Overflow is a platform aimed at programmers of all levels for asking and answering technical questions. The platform was created in 2008[5] and it's the most popular site as part of the Stack Exchange Network[6]. The platform offers multiple features, the most popular one being the ability to up vote and down vote user posts (either questions or answers) contributing to the posting user's overall reputation. Based on reputation, the plaform enables users to reach different priviledges such as the ability to vote, comment and edit posts. Furthermore, users are awarded "badges" (i.e. achievements) that relate to: the overall reputation, answers, questions or even the frequency of use of the site. The author of this work is an avid user of the platform[7] since its inception and has periodically used it more towards asking rather than answering technical questions. A result of the present work, is to more strongly consider using the site for **answering questions** rather than just asking.

Very luckily for us the Stack Overflow data is available for download[8] and in different formats, opening the door for conducting truly interesting data analysis with many applications in the context of the recruitment industry such as: resume (or CV) compilation[9], job candidate shortlisting, job candidate assessment, staff skills assessment, technology trends analysis, and many more.

Project goals

We first focus in delivering a fully automated method and R code to download, extract, process and clean the Stack Overflow dataset that's directly applicable to any other Stack Exchange Network site data. The dataset we compiled is **anonymised** i.e only the artificial integer key `userId` is stored. After a basic exploratory data analysis of the full dataset we move to the following data analysis use-cases. The first two objectives are covered as part of the Data exploration and visualization section. The last objective is covered in the Modeling approach and Method implementation sections.

[4] https://www.stackoverflow.com
[5] https://en.wikipedia.org/wiki/Stack_Overflow
[6] https://stackexchange.com/sites#
[7] https://stackoverflow.com/users/1142881/
[8] https://archive.org/details/stackexchange
[9] https://www.kickstarter.com/projects/1647975128/one-thousand-words-cv-1kwcv/

The What: skills and technology trends

Questions in Stack Overflow contain one or more tags e.g. java. For professionals who work in programming these tags are skills and they are listed as such in a resume (or CV)[10]. Therefore, in this analysis step we'd like to find the top skills by frequency of use, establish a proximity measure and discover skill groups. We're also very interested in discovering what major technology trends exist and their importance.

To this end, the top tags (or skills) are first selected. The key modeling approach (in NLP terms) is to view questions as "documents" and skills as "words" and count how many times skills occur pair-wise together in the same questions, therefore we generate a skills co-occurrence matrix. We then compute Principal Component Analysis (PCA) on the scaled co-occurrence matrix. The first two principal components reveal what are the skill groups that explain most of the variance in the data or how we like to call it the main "technology trends". We visualize the top and bottom ends of these two principal components. The remaining PCA components reveal other skill groups.

This first analysis step enpower us to answer many practical questions e.g.

- As a programmer: what are the main technology trends at the moment and which one shall I invest learning on?
- As a company: we'd like to build a new product, which technology stack should we use? check the top components for the most popular stacks in the area required.
- As a resume (or CV) compilation service: suggest candidates with skills they may have overlooked to include in their CV and are among the most important e.g. when listing `java`, suggest also `java-ee`.

Note that applying this method in a rolling time window fashion e.g. every year compute this analysis for a time window of three years ending at the given date; will reveal the industry changes in technology trends over time.

The Where: putting it in geographical context

Here we search for all Stack Overflow users in Switzerland, find their top technology skills looking into the tags linked to their top answers by score or otherwise top questions by score, and link these top skills to the top technology trends revealed in the first few PCA components of the previous analysis. We then use Google's Geocoding API and Maps Static API to compute the user locations (i.e. longitude and latitude) and extract a map of Switzerland respectively. Finally we put the main technology trends revealed by the previous analysis into geographical context. Note that due to the Google Maps Platform Terms of Services the geocoding results may not be cached, therefore to be able to execute and reproduce the results in this section you'd need a valid Google API key see Get an API Key and make it available in your environment as `GOOGLE_API_KEY`. However, the few API calls needed will easily fit cost-free within a free trial version of the Google Maps API.

You may wonder why Switzerland? because it's where the author lives and Switzerland is a relatively small country which is nice in order to keep the geocoding costs low i.e. we need to call Google's Geocoding API for every[11] user located in Switzerland. This second step enables answering very practical questions e.g.

[10] https://www.dropbox.com/s/6t7mq5zcztarah4/1kwcv_prototype_Giovanni.pdf?dl=1

[11] Actually all the distinct user locations i.e. about four hundreds

- As a programmer: which locations should I consider to find jobs that match my main areas of expertise?
- As a company: where do we find relevant partners and support on the technology areas we need?
- As a recruitment company: where should we look for talent?

The How: rating user skills

The author of this work has in the past applied to jobs with listing describing requirements that include one or a few skills for which he had no previous experience. For example, he was recently rejected while applying for a position that required knowledge of Tableau. Indeed, he had no previous experience on this particular skill so it wasn't listed in his CV; however, he has extensive experience in data visualization using e.g. `d3.js` and `ggplot2`. His `sql` skill level is well above average too, therefore he intuitively "felt" that he would have nevertheless been a great match for that position and that's why he applied in the first place.

This project reminded of another anecdote, a collegue that was very unsure which graduate program to pursue. Surprisingly he decided to ask the admissions secretary and she recommended that he would be better off going for a master in Computational Biology and so he did. If a secretary can do it, can we not also trust a machine learning model to recommend what future career to pursue given your skill ratings?

In this final step a user skill `ratings` dataset derivation is designed, constructed and modeled to solve the ultimate task covered in this project: to predict how good a candidate would be in a skill for which there is no previous evidence. That's it, we present and implement a recommender system to predict user-skill ratings using the collaborative filtering approach. More specifically, we'll apply the model-based approach using low-rank matrix factorization (LRMF) and two implementation variations of the stochastic gradient descent (SGD) algorithm. The first algorithm is based on the classic SGD with multiple improvements for faster and better convergence e.g. cast the `P` and `Q` matrices of the SGD algorithm in column major ordering to match the R's default matrix memory ordering. The second algorithm is an R-based lock-free parallel multi-core SGD variation of the first featuring a speed up of roughly 2x with comparable high quality out-of-sample RMSE result and with potential for higher speed ups. However, we first navigate through a simpler baseline model implementation based on isolating the different biases or effects and we'll there outline some interesting findings.

What used to be just an intuitive "feeling" was happily confirmed by the model employed in this project as it predicted the author's rating on `tableau` to be well above average. What's the moral of the story? hiring personnel shouldn't be too hasty dismissing candidates whose skills don't match the job requirements exactly but first consult a Machine Learning model like the one we built in this project.

This last analysis enables us to answer many practical questions too e.g.

- As a programmer: given my current skill ratings, in what technologies am I predicted to perform way average?
- As a company: Can we reorganize and optimize our skills distribution per department without firing / hiring anyone?

- As a recruitment company: candidate X doesn't explicitly list required skill Y in her CV, however our model predicted her to be a perfect match for that job.

Chapter 1

Data analysis

1.1 Data import

The script `create_dataset.r` contains all the code to automatically download, extract, parse, clean and construct all the complete dataset required for analysis e.g. the `ratings`. The script is large and only the most important points will be covered; however, the script is very well structured and commented. Running `create_dataset.r` the first time may take several hours and require large amounts of free disk disk space. Furthermore the script requires running in an Unix-like environment[1] that has the following tools available: `wc`, `split`, `awk`, `7z`, `rename`, `mv`, `grep` and `time`. The script will automatically create and populate the relative folders `data/7z/*` containing the downloaded files; the folder `data/xml/*` containing the extracted XML files and finally the folder `data/rds/*` containing the generated dataset files[2]. Note that running `create_dataset.r` **is optional** as the final `rds` files are readily available under the relative folder `data/rds/*` in the project's GitHub repository https://github.com/bravegag/HarvardX-Skillability.

Name	Compressed	Uncompressed	Description
'stackoverflow.com-Badges.7z'	254.5MB	4.0GB	All badge assignments.
'stackoverflow.com-Posts.7z'	15.3GB	76.5GB	All the question and answer posts.
'stackoverflow.com-Tags.7z'	817.0kb	5.1MB	All the tags.
'stackoverflow.com-Users.7z'	529.3MB	3.7GB	All the users.

The extracted XML file where up to 76.5GB in size. Several methods were tested to load, parse and extract the data from such big files and the best solution found was a combination of the following points[3]:

1. Splitting the huge files into smaller ones (split into as many files as there are cores available), loading and parsing the files in parallel. Note that the split files are temporarily written to the relative `data/xml` directory.
2. Using `readr::read_lines_chunked` to read chunks of XML, keeping the memory footprint low as each core will process a bounded chunk of XML.

[1] It was tested in Ubuntu 18.04 with 32GB RAM, a 6-core Intel i7-4960X and a SSD drive.
[2] Available in the project's GitHub page: https://github.com/bravegag/HarvardX-Skillability
[3] See functions `downloadExtractAndProcessXml(...)` and `extractDataFromXml2(...)`

3. The trick to turn these smaller XML chunks of rows into a valid XML was to wrap them within `<xml>...</xml>` tags[4].
4. Finally use the package `xml2` for parsing, extracting and consolidating the data into tibbles which are later stored as `rds` files.

The function `extractDataFromXml2(...)` was made generic; it takes a `mapping` parameter that identifies which XML attributes to read and what column names they should be mapped to in the resulting tibble.

Note that the huge Posts XML file contains both questions and answers and they're automatically segregated by grepping for attributes that would only be contained in either e.g. only questions contain the attribute `AnswerCount` so we do `system(command=sprintf("time grep \"AnswerCount\" %s/Posts.xml > %s/Questions.xml", xmlDir, xmlDir))`.

1.2 Data cleaning

The data cleaning steps were also covered as part of the `create_dataset.r` implementation. The cleaning process removes rows with missing important XML attributes e.g. answer posts with missing "foreign key" `questionId`. Several data transformations are applied too e.g. the questions attribute `tags` has HTML entity separators which are transformed into pipe separated[5]. The cleaning outcome is briefly summarized in the following table:

Dataset	**Before**	**After**	**Description**
tags	56.5k rows	56.5k rows	Unchanged.
users	~11.37m rows	~200k rows	Keep only users having reputation greater than 999 or are located in Switzerland.
badges	~12.59m rows	~12.59m rows	Unchanged.
questions	~18.59m rows	~5.39m rows	Keep only questions answered or created from the users selection and in the later case with a score greater than 0.
answers	~28.25m rows	~4.83m rows	Keep only answers created from the users selection, with score greater than 2 and having a valid answerId.

We notice that only 1.75% of the users are participative. The vast majority of users only seem to create a handful of posts and use the site in "read-only" mode i.e. not making any posts. Read-only users are not interesting for the different analyses and were therefore excluded.

1.3 Data exploration and visualization

1.3.1 Description of the dataset

We load the bundled `rds` data files using the following code:

[4] Credits given to the answer of https://stackoverflow.com/questions/59329354 for coining the idea.
[5] See `create_dataset.r` lines #548 and #549.

```r
# load the Users, Questions, Answers, Badges and Tags data files
users <- readObjectByName("Users")
questions <- readObjectByName("Questions")
answers <- readObjectByName("Answers")
badges <- readObjectByName("Badges")
tags <- readObjectByName("Tags")
```

The `users` frame includes all the users we have selected for analysis, each row is uniquely identified by the key `userId` which is used to link with other tables. The column `creationDate` timestamp represents the time when the user account was first created. Note that `location` is free text which we use later as input to the Google Geocoding API for generating geographic coordinates:

```r
prettyPrint(head(glimpse(users)))
```

```
## Observations: 200,613
## Variables: 8
## $ userId         <int> 1142881, 1, 2, 3, 4, 5, 8, 9, 11, 13, 17, 19, 20, 22, ...
## $ reputation     <dbl> 9689, 58233, 5532, 15096, 31470, 47714, 1787, 20938, 4...
## $ creationDate   <dttm> 2012-01-11 09:46:39, 2008-07-31 14:22:31, 2008-07-31 ...
## $ lastAccessDate <dttm> 2019-11-30 20:00:59, 2019-11-15 23:50:12, 2019-11-27 ...
## $ location       <chr> "Leimbach, Switzerland", "El Cerrito, CA", "Corvallis,...
## $ views          <int> 1406, 516045, 25890, 25860, 77070, 12709, 7213, 5686, ...
## $ upvotes        <int> 2351, 3377, 655, 7587, 804, 785, 12, 47, 0, 5206, 885,...
## $ downvotes      <int> 42, 1311, 88, 100, 96, 34, 9, 4, 0, 210, 216, 13, 38, ...
```

userId	reputation	creationDate	lastAccessDate	location	views	upvotes	downvotes
1142881	9689	2012-01-11 09:46:39	2019-11-30 20:00:59	Leimbach, Switzerland	1406	2351	42
1	58233	2008-07-31 14:22:31	2019-11-15 23:50:12	El Cerrito, CA	516045	3377	1311
2	5532	2008-07-31 14:22:31	2019-11-27 20:35:05	Corvallis, OR	25890	655	88
3	15096	2008-07-31 14:22:31	2019-11-26 20:34:32	Raleigh, NC, United States	25860	7587	100
4	31470	2008-07-31 14:22:31	2019-11-25 23:15:40	New York, NY	77070	804	96
5	47714	2008-07-31 14:22:31	2019-11-26 01:17:06	San Diego, CA	12709	785	34

The `questions` frame contains all the questions we have selected and each row is uniquely identified by the key `questionId` which is used to link with other tables. Note the `tags` column will be used extensively in this work; it has during the data cleaning phase already been preprocessed to pipe separated from HTML encoded entities. The column `acceptedAnswerId` identifies the accepted answer among all that corresponds to answers' `answerId`:

```r
prettyPrint(head(glimpse(questions)), latex_options = c("striped", "scale_down"))
```

```
## Observations: 5,393,900
## Variables: 11
## $ questionId       <int> 4, 6, 9, 11, 13, 14, 16, 17, 19, 24, 25, 34, 36, 39,...
## $ acceptedAnswerId <int> 7, 31, 1404, 1248, NA, NA, 12446, 26, 531, 49, 14439...
## $ creationDate     <dttm> 2008-07-31 21:42:52, 2008-07-31 22:08:08, 2008-07-3...
## $ score            <dbl> 645, 290, 1754, 1461, 597, 405, 132, 181, 315, 167, ...
## $ viewCount        <int> 45103, 18713, 574705, 152779, 182042, 126872, 82717,...
## $ userId           <int> 8, 9, 1, 1, 9, 11, 2, 2, 13, 22, 23, NA, 32, 33, 37,...
## $ lastActivityDate <dttm> 2019-10-21 14:03:54, 2019-07-19 01:43:04, 2019-11-2...
## $ tags             <chr> "c#|floating-point|type-conversion|double|decimal", ...
## $ answerCount      <int> 13, 6, 61, 37, 24, 10, 7, 9, 23, 6, 9, 8, 8, 2, 8, 2...
## $ commentCount     <int> 3, 0, 5, 3, 10, 4, 0, 3, 16, 0, 0, 0, 2, 0, 0, 0, 0,...
## $ favoriteCount    <int> 49, 11, 438, 537, 147, 57, 14, 20, 80, 25, 7, 2, 25,...
```

<table>
<tr><th>questionId</th><th>acceptedAnswerId</th><th>creationDate</th><th>score</th><th>viewCount</th><th>userId</th><th>lastActivityDate</th><th>tags</th><th>answerCount</th><th>commentCount</th><th>favoriteCount</th></tr>
<tr><td>4</td><td>7</td><td>2008-07-31 21:42:52</td><td>645</td><td>45103</td><td>8</td><td>2019-10-21 14:03:54</td><td>c#|floating-point|type-conversion|double|decimal</td><td>13</td><td>3</td><td>49</td></tr>
<tr><td>6</td><td>31</td><td>2008-07-31 22:08:08</td><td>290</td><td>18713</td><td>9</td><td>2019-07-19 01:43:04</td><td>html|css|internet-explorer-7</td><td>6</td><td>0</td><td>11</td></tr>
<tr><td>9</td><td>1404</td><td>2008-07-31 23:40:59</td><td>1754</td><td>574705</td><td>1</td><td>2019-11-27 07:20:41</td><td>c#|.net|datetime</td><td>61</td><td>5</td><td>438</td></tr>
<tr><td>11</td><td>1248</td><td>2008-07-31 23:55:37</td><td>1461</td><td>152779</td><td>1</td><td>2019-05-26 02:31:53</td><td>c#|datetime|time|datediff|relative-time-span</td><td>37</td><td>3</td><td>537</td></tr>
<tr><td>13</td><td>NA</td><td>2008-08-01 00:42:38</td><td>597</td><td>182042</td><td>9</td><td>2019-05-14 16:02:10</td><td>html|browser|timezone|user-agent|timezone-offset</td><td>24</td><td>10</td><td>147</td></tr>
<tr><td>14</td><td>NA</td><td>2008-08-01 00:59:11</td><td>405</td><td>126872</td><td>11</td><td>2019-10-01 17:18:01</td><td>.net|math</td><td>10</td><td>4</td><td>57</td></tr>
</table>

The `answers` frame contains all the answers also uniquely identified by the key `answerId`. Note that we can find the tags or skills linked to an answer by joining with the `questions` frame via the "parent" `questionId` and reading the `tags` column:

```
prettyPrint(head(glimpse(answers)), latex_options = c("striped", "scale_down"))
```

```
## Observations: 4,830,000
## Variables: 7
## $ answerId         <int> 7, 12, 22, 26, 27, 29, 30, 33, 44, 45, 49, 51, 52, 5...
## $ questionId       <int> 4, 11, 9, 17, 11, 13, 25, 14, 39, 39, 24, 36, 34, 34...
## $ creationDate     <dttm> 2008-07-31 22:17:57, 2008-07-31 23:56:41, 2008-08-0...
## $ score            <dbl> 433, 326, 37, 136, 29, 117, 37, 463, 17, 55, 60, 25,...
## $ userId           <int> 9, 1, 17, 48, 17, 19, 13, 13, 35, 39, 43, 17, 23, 34...
## $ lastActivityDate <dttm> 2019-10-21 14:03:54, 2018-01-12 16:10:22, 2008-08-0...
## $ commentCount     <int> 0, 11, 0, 0, 0, 5, 0, 4, 0, 0, 1, 0, 0, 0, 3, 6, 0, ...
```

answerId	questionId	creationDate	score	userId	lastActivityDate	commentCount
7	4	2008-07-31 22:17:57	433	9	2019-10-21 14:03:54	0
12	11	2008-07-31 23:56:41	326	1	2018-01-12 16:10:22	11
22	9	2008-08-01 12:07:19	37	17	2008-08-01 15:26:37	0
26	17	2008-08-01 12:16:22	136	48	2016-06-02 05:55:17	0
27	11	2008-08-01 12:17:19	29	17	2008-08-01 13:16:49	0
29	13	2008-08-01 12:19:17	117	19	2008-08-01 12:19:17	5

The `badges` frame contains the user badge assignments (linked via the foreign key `userId`), for example, the `gold` badge "Populist" is one of the hardest for an user to get and requires outscoring an already accepted answer with score of more than 10 by more than 2x of the accepted answer[6]:

```
prettyPrint(head(glimpse(badges)), latex_options = c("striped"))
```

```
## Observations: 12,597,503
## Variables: 4
## $ userId <dbl> 3718, 3893, 4591, 2635, 1113, 164, 5246, 509, 5024, 1284, 2907...
## $ badge  <fct> Teacher, Teacher, Teacher, Teacher, Teacher, Teacher, Teacher,...
## $ date   <dttm> 2008-09-15 08:55:03, 2008-09-15 08:55:03, 2008-09-15 08:55:03...
## $ class  <fct> bronze, bronze, bronze, bronze, bronze, bronze, bronze, bronze...
```

userId	badge	date	class
3718	Teacher	2008-09-15 08:55:03	bronze
3893	Teacher	2008-09-15 08:55:03	bronze
4591	Teacher	2008-09-15 08:55:03	bronze
2635	Teacher	2008-09-15 08:55:03	bronze
1113	Teacher	2008-09-15 08:55:03	bronze
164	Teacher	2008-09-15 08:55:03	bronze

Finally the tags data frame contains the all the unique tags along with their use counts:

[6] See https://stackoverflow.com/help/badges/62/populist

```r
prettyPrint(head(glimpse(tags)), latex_options = c("striped"))
```

```
## Observations: 56,525
## Variables: 3
## $ tagId <int> 1, 2, 3, 4, 5, 8, 9, 10, 12, 14, 16, 17, 18, 19, 21, 22, 23, 26...
## $ tag   <chr> ".net", "html", "javascript", "css", "php", "c", "c#", "c++", "...
## $ count <dbl> 289949, 863524, 1909662, 612173, 1320075, 316522, 1363768, 6445...
```

tagId	tag	count
1	.net	289949
2	html	863524
3	javascript	1909662
4	css	612173
5	php	1320075
8	c	316522

1.3.2 Quick exploration

Let's explore some interesting facts from the data we have, the top ranking question, answer, user, tags (i.e. skills) and the top ten gold badges. The top ranking answer applies to the top ranking question and they relate to `c++`, `performance` and code `optimization`. By looking at the top ten gold badges it shows that being awarded with a "Great Question"[7] is harder than for a "Great Answer"[8], it would seem from this result that users tend to up vote more answers than questions:

```r
# what's the question with highest score?
prettyPrint(
  questions %>%
    top_n(1, score) %>%
    select(questionId, acceptedAnswerId, tags, score, answerCount, favoriteCount,
           viewCount)
)
```

questionId	acceptedAnswerId	tags	score	answerCount	favoriteCount	viewCount
11227809	11227902	java|c++|performance|optimization|branch-prediction	23665	26	10754	1424807

```r
# what's the answer with highest score?
prettyPrint(
  answers %>%
    top_n(1, score) %>%
    select(answerId, questionId, score, commentCount, creationDate)
, latex_options = c("striped"))
```

answerId	questionId	score	commentCount	creationDate
11227902	11227809	30924	81	2012-06-27 13:56:42

```r
# what's the top user?
prettyPrint(
  users %>%
```

[7]See https://stackoverflow.com/help/badges/22/great-question
[8]See https://stackoverflow.com/help/badges/25/great-answer

```
    top_n(1, reputation) %>%
    select(userId, reputation, creationDate, location, upvotes, downvotes)
)
```

userId	reputation	creationDate	location	upvotes	downvotes
22656	1147559	2008-09-26 12:05:05	Reading, United Kingdom	16489	6951

```
# what are the top ten tags / skills?
prettyPrint(
  topTenSkills <- tags %>%
    top_n(10, count) %>%
    arrange(desc(count)) %>%
    rename(skill=tag)
, latex_options = c("striped"))
```

tagId	skill	count
3	javascript	1909662
17	java	1613332
9	c#	1363768
5	php	1320075
16	python	1297831
1386	android	1237204
820	jquery	971432
2	html	863524
10	c++	644544
58338	ios	615326

```
# what are the top ten gold badges hardest to get i.e. with fewer users awarded?
prettyPrint(
  badges %>%
    filter(class == "gold") %>%
    group_by(badge) %>%
    summarise(awarded = n()) %>%
    top_n(10, -awarded) %>%
    arrange(awarded)
, latex_options = c("striped"))
```

badge	awarded
Sheriff	42
Illuminator	122
Legendary	278
Reversal	292
Lifeboat	818
Publicist	1270
Marshal	2843
Copy Editor	3327
Socratic	4061
Stellar Question	7681

The following statistics reveal the highly skewed nature of the data (the mean is far from the median in most cases):

```r
# what's the average user reputation?
prettyPrint(
  users %>%
    summarise(mean=mean(reputation), median=median(reputation))
, latex_options = c("striped"))
```

mean	median
5499.315	2164

```r
# what's the average number of questions per user?
prettyPrint(
  questions %>%
    group_by(userId) %>%
    summarise(n = n()) %>%
    ungroup() %>%
    summarise(mean=mean(n), median=median(n))
, latex_options = c("striped"))
```

mean	median
5.353162	1

```r
# what's the average number of answers per user?
prettyPrint(
  answers %>%
    group_by(userId) %>%
    summarise(n = n()) %>%
    ungroup() %>%
    summarise(mean=mean(n), median=median(n))
, latex_options = c("striped"))
```

mean	median
26.25799	8

```r
# what's the average number of answers per question?
prettyPrint(
  questions %>%
    summarise(mean=mean(answerCount), median=median(answerCount))
, latex_options = c("striped"))
```

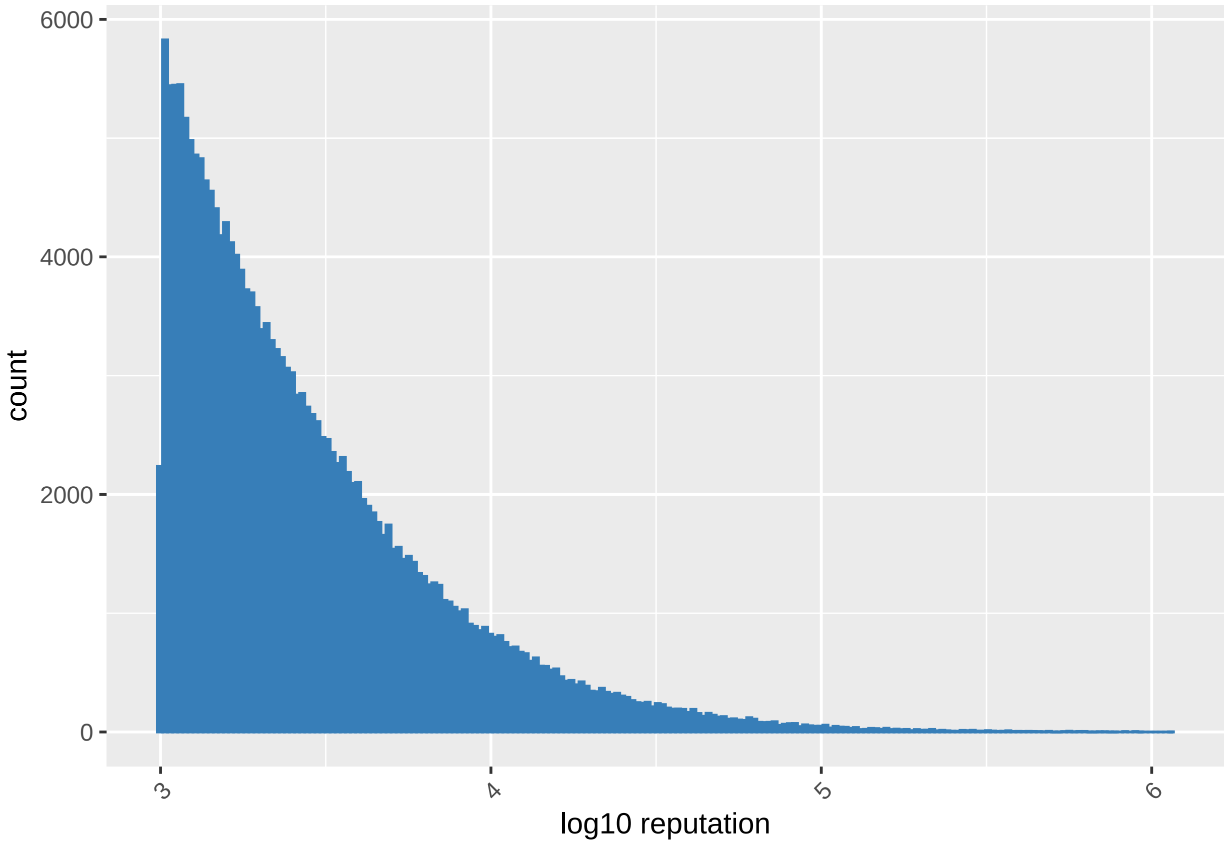
mean	median
2.278197	2

In the following listing we apply the `log10` transformation[9] to the users reputation and plot its histogram, the plot confirms that the users reputation is positively skewed. Remember that we set the user selection criteria to be: users with reputation greater than 999 or located in Switzerland,

[9] We preferred to work with the `log10` for reputation because it's a nicer scale, an order-invariant transformation, and easier to interpret than natural `log`.

so there we have some of the users located in Switzerland to the left of $\log10(999) \approx 3$ and we exclude them first:

```
users %>%
  filter(reputation > 999) %>%
  mutate(reputation=log10(reputation)) %>%
  ggplot(aes(reputation)) +
  geom_histogram(bins = 200, colour="#377EB8", fill="#377EB8") +
  xlab("log10 reputation") +
  theme(plot.title = element_text(hjust = 0.5),
        legend.text = element_text(size=12),
        axis.text.x = element_text(angle = 45, hjust = 1))
```



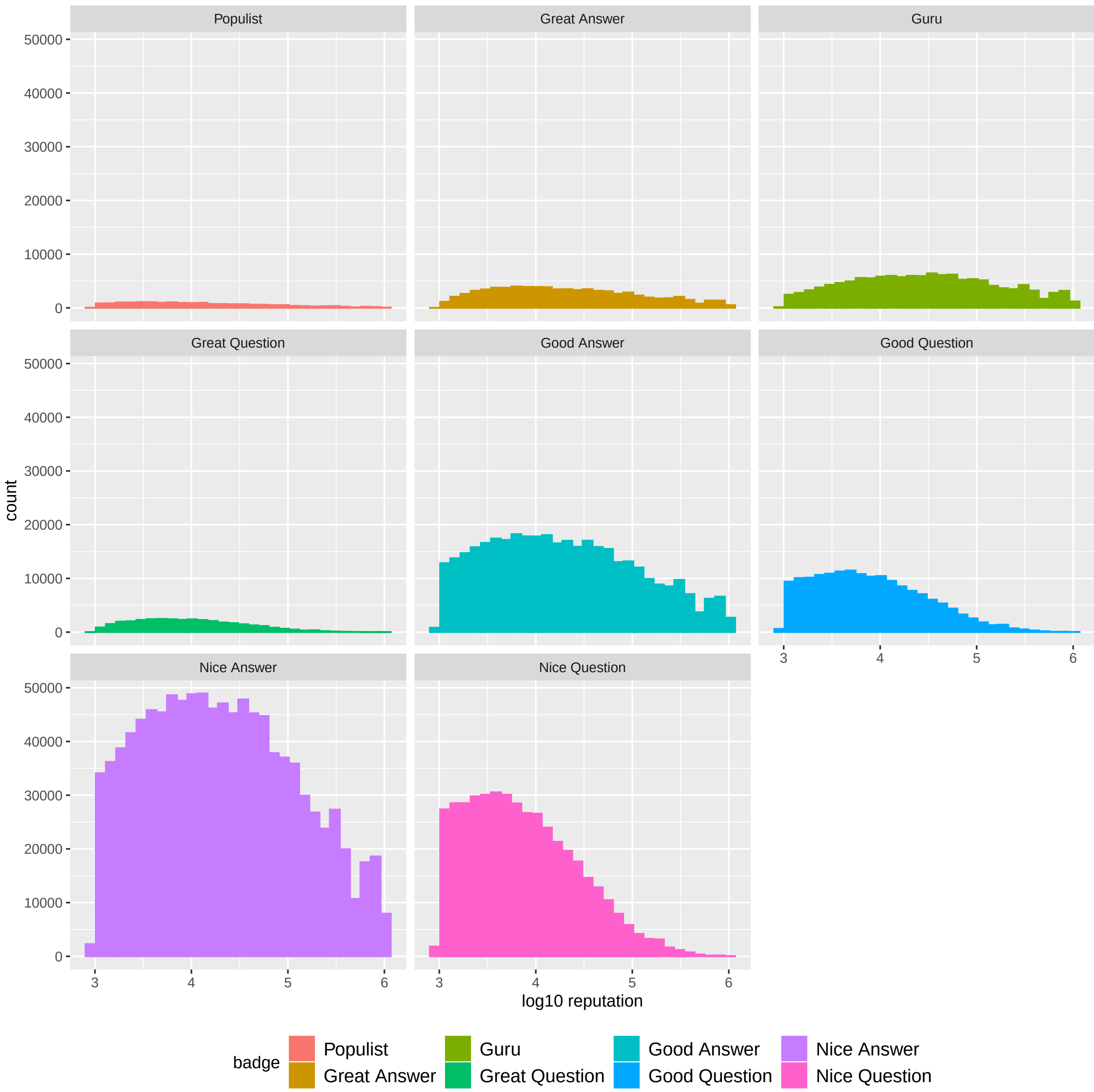
Now, if we split the users per badges[10] then the histograms look a bit nicer i.e. no longer so skewed but still asymmetrical and quite departed from a normal distribution:

```
# histogram of the log10-transfored of users reputation per badge and
# excluding users with less than 999 reputation
users %>%
  filter(reputation > 999) %>%
  mutate(reputation=log10(reputation)) %>%
  inner_join(badges %>%
               select(userId, badge) %>%
               filter(badge %in% c("Populist", "Great Answer", "Guru", "Great Question",
                                   "Good Answer", "Good Question", "Nice Answer",
                                   "Nice Question")), by="userId") %>%
  mutate(badge=factor(badge, levels=c("Populist", "Great Answer", "Guru",
                                      "Great Question", "Good Answer", "Good Question",
                                      "Nice Answer", "Nice Question"))) %>%
```

[10] We chose only the badges relevant for our analysis see https://stackoverflow.com/help/badges

```
ggplot(aes(reputation, group=badge, color=badge, fill=badge)) +
geom_histogram() +
xlab("log10 reputation") +
theme(legend.position="bottom", legend.text=element_text(size=3)) +
theme(plot.title = element_text(hjust = 0.5),
      legend.text = element_text(size=12)) +
facet_wrap(~badge)
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



1.3.3 The What: skills and technology trends

In the following listing we want to find the main technology trends and how skills group together. To this end we first select the top 2k skills by frequency of tagging, compute their pair-wise co-occurrence matrix and run PCA on it.

```r
# select the top 2k tags/skills by count
mainSkills <- tags %>%
  top_n(2000, count) %>%
  rename(skill=tag) %>%
  arrange(desc(count))

# what's the proportion to the total?
100*sum(mainSkills$count)/sum(tags$count)
```

```
## [1] 82.33486
```

```r
# select a smaller questions subset matching the main tags
# to get the results faster ...
questionSkills <- questions %>%
  filter(score > 9 & viewCount > 99 & answerCount > 1)
# takes ~35s
tic(sprintf('separating rows with %d', nrow(questionSkills)))
questionSkills <- questionSkills %>%
  select(questionId, tags) %>%
  separate_rows(tags, sep="\\|") %>%
  rename(skill=tags) %>%
  inner_join(mainSkills, by="skill") %>%
  arrange(desc(count)) %>%
  select(questionId, skill)
toc()
```

```
## separating rows with 490245: 36.253 sec elapsed
```

```r
# takes ~15m if TRUE
if (FALSE) {
  tic(sprintf('computing co-occurrence matrix with %d question-skill',
              nrow(questionSkills)))
  X <- crossprod(table(questionSkills[1:2]))
  diag(X) <- 0
  toc()
  saveObjectByName(X, "XCo-occurrence")
}
X <- readObjectByName("XCo-occurrence")

# how sparse is it?
sum(X == 0)/(dim(X)[1]^2)
```

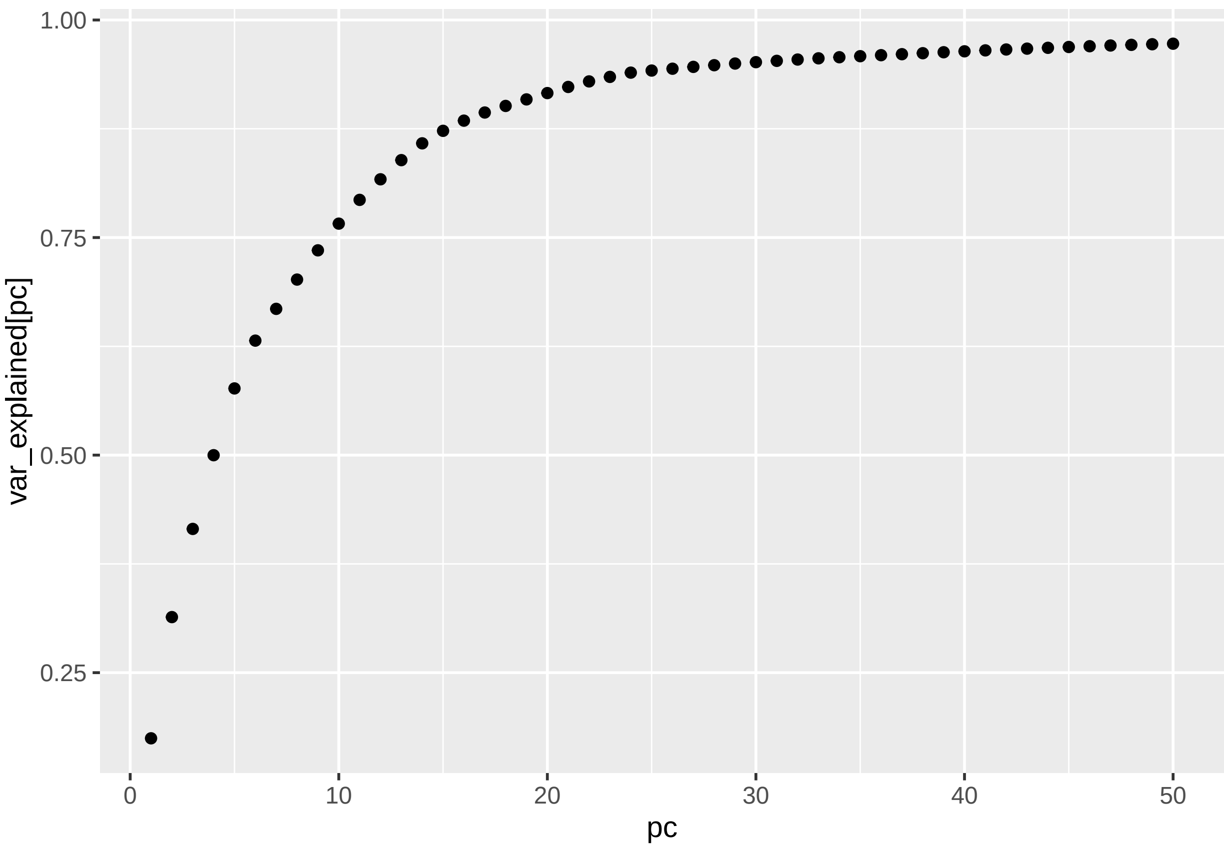
```
## [1] 0.9299415
```

```r
# compute PCA
pca <- prcomp(X)
```

At this point we can plot the cumulative variability explained up to each principal component. We see that the first four components explain 50% of the variance:

```r
# let's consider the first 50 components only
pc <- 1:50

# plot the variability explained
var_explained <- cumsum(pca$sdev^2 / sum(pca$sdev^2))
```

```r
qplot(pc, var_explained[pc])
```

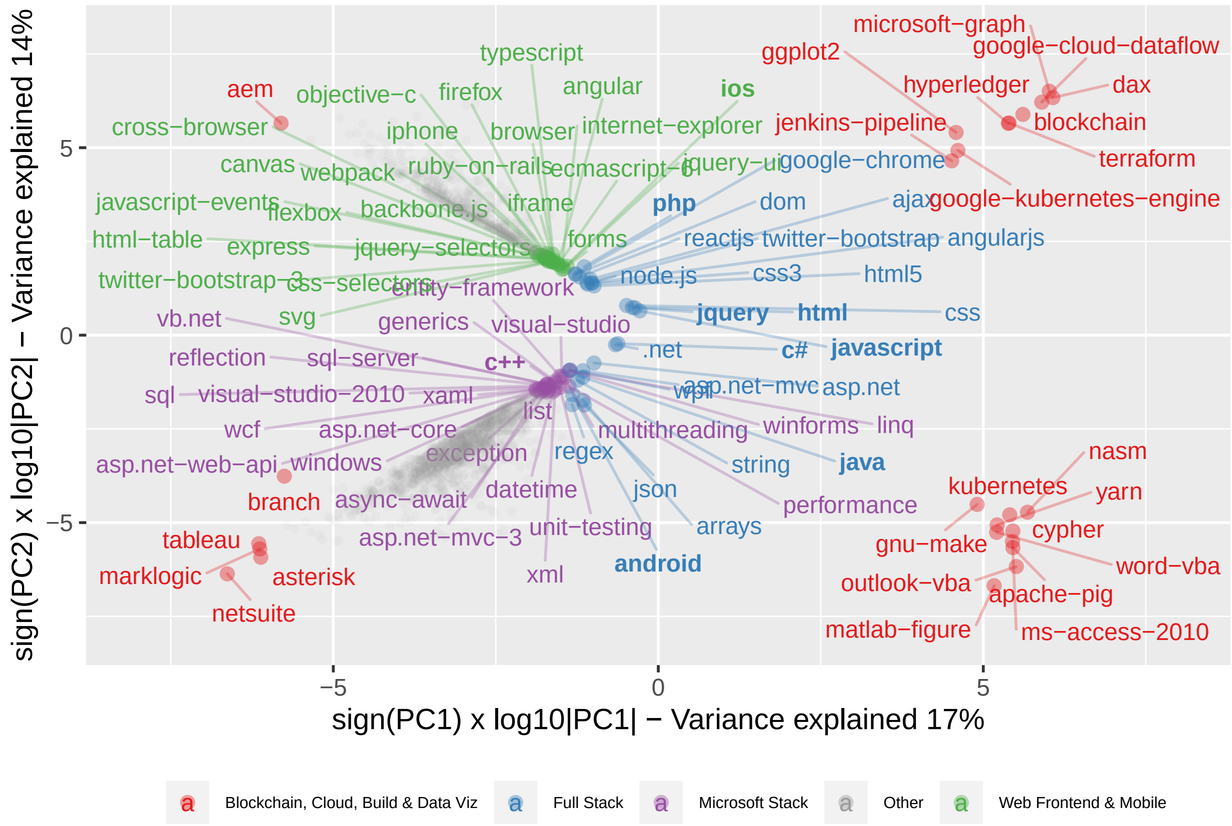


The following plot reveals the main technology trends. The top skills are highlighted in bold font-face. The top and bottom ends of the first principal component reveal “Blockchain, Cloud, Build and Data Visualization” (in red) and “Full Stack” (in blue) respectively. While the top and bottom ends of the second principal component reveal “Web Frontend & Mobile” (in green) and “Microsoft Stack” (in purple) respectively. Note that the technology trends were named e.g. “Microsoft Stack” after reviewing all the skills found in those segments and assigning a more general conceptual trend but the resulting groupings are not exact e.g. `c++` appears in the second component “Microsoft Stack” while the third component groups together **mostly** “Python & C++” skills e.g. `python`, `c++11`, `stl`, `boost`, `qt`, `visual-c++`, etc.

The grouping here is very interesting, The link between cloud and build tools is clear, since most of the cloud technologies are related to and require building and deploying software. Furthermore, it would seem that blockchain software is linked to deploying software in the cloud; likewise there seems to be a link between software deployment, cloud technologies, generating reports and data visualization.

```r
portable.set.seed(1)
highlightLog %>%
  filter(pc %in% c(1, 2)) %>%
  ggplot(aes(PC1, PC2, label=skill, colour=Technology)) +
  geom_jitter(alpha = 0.4, size = 2) +
  theme(legend.position="bottom", plot.title = element_text(hjust = 0.5),
        legend.text=element_text(size=6), legend.title = element_blank()) +
  guides(fill = guide_legend(nrow=2)) +
  xlab(sprintf("sign(PC1) x log10|PC1| - Variance explained %d%%",
               round(100*pca$sdev[1]^2 / sum(pca$sdev^2)))) +
```

```r
  ylab(sprintf("sign(PC2) x log10|PC2| - Variance explained %d%%",
               round(100*pca$sdev[2]^2 / sum(pca$sdev^2)))) +
  geom_text_repel(aes(fontface=fontface), segment.alpha = 0.3, size = 3,
                  force = 7, nudge_x = 0.1, nudge_y = 0.1, seed = 1) +
  scale_colour_manual(values = colorSpec) +
  scale_x_continuous(limits=c(-8, 8)) +
  scale_y_continuous(limits=c(-8, 8)) +
  geom_jitter(data = nonHighlightLog, aes(PC1, PC2), alpha = 0.05, size = 1)
```



1.3.4 The Where: putting it in geographical context

The following code, using a valid environment `GOOGLE_API_KEY`[11] will match all users with location Switzerland and compute their geographic coordinates using Google's geocoding API. We note that Switzerland has a large expatriate english-speaking technology community plus four official Swiss languages, therefore we filter for user `location` containing the Swiss country code `ch` or `switzerland` written in english or written using any of the four official Swiss languages: in German `schweiz`, Italian `svizzera`, French `suisse` and Romansh `svizra` respectively:

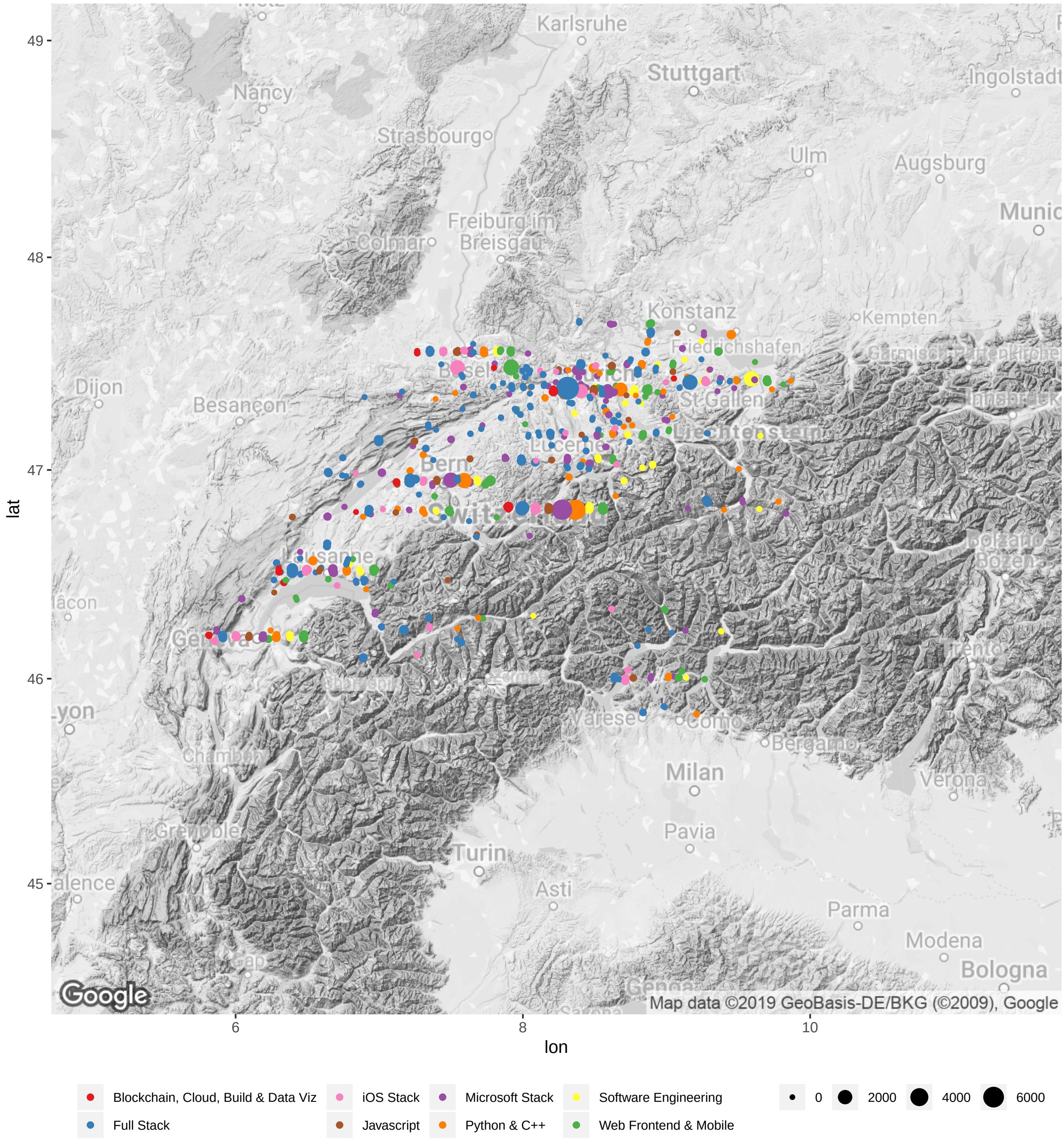
```r
# do this only if the file isn't there to avoid costly Google's geocoding calls
if (!file.exists(filePathForObjectName("UsersCH"))) {
  # the environment variable GOOGLE_API_KEY is required or simply copy-paste your
  # google key instead. To obtain a google key, follow the steps outlined here:
  # https://developers.google.com/maps/documentation/javascript/get-api-key
  register_google(key=Sys.getenv("GOOGLE_API_KEY"))

  # get users whose location is Switzerland only
```

[11]See instructions here to get a free trial Google API key https://developers.google.com/maps/documentation/javascript/get-api-key

```r
  usersCh <- users %>%
    filter(str_detect(tolower(location),
                      "(\\bch\\b|switzerland|schweiz|svizzera|suisse|svizra)")) %>%
    arrange(desc(reputation))

  # get the unique locations and avoid duplicate calls e.g. "Zurich, Switzerland"
  swissLocations <- usersCh %>%
    select(location) %>%
    unique()
  # WARNING! this code paired with a valid GOOGLE_API_KEY may cost money!
  swissLocations <- mutate_geocode(swissLocations, location = location)
  usersCh <- usersCh %>%
    left_join(swissLocations, by="location")

  # write the usersCh to disk
  saveObjectByName(usersCh, "UsersCH")
}
usersCh <- readObjectByName("UsersCH")
stopifnot(nrow(usersCh) == 4235)
```

The following plot depicts the technology trends discovered in the previous analysis and now shown in geographical context for Switzerland. We note that Zurich is becoming a true technology center in Europe as all the trends are there. The most prominent data point by score in Switzerland was reached by an user located in Zurich posting on Full Stack development. We can also see that the east and south of Switzerland i.e. the Tessin region has much lower activity technology-wise therefore it wound't be a wise decision looking for technology jobs there. The data points appearing in the center of Switzerland correspond to users who were not precise in providing their specific location i.e. they specified their locations to "Switzerland" and that's the center of Switzerland but technology-wise we should not expect to find anything in the middle of the mountains. Geneve city was surprisingly less active in quantity and quality or may be that users there didn't provide their location precisely enough. The city of Bern shows two high scoring data points connected to the technology trends Microsoft Stack and Python & C++ respectively. We can also note several users in isolated Swiss regions working on `ios` i.e. potentially building iPhone applications in remote areas which would make sense.

```r
# plot the top technology trends in Geo-context in Switzerland
ggmap(map) +
  scale_colour_manual(values = colorSpec) +
  geom_point(data=usersChTop, aes(x=lon, y=lat, colour=Technology, size=score),
             position = position_jitterdodge(jitter.width=0.01, jitter.height=0.01,
                                             seed=1)) +
  theme(plot.title = element_text(hjust = 0.5), legend.text=element_text(size=8),
        legend.title = element_blank(), legend.position="bottom")
```



The top ten most prominent post data points in Switzerland are revealed using the following code:

```
prettyPrint(
  usersChTop %>%
    top_n(10, score) %>%
    arrange(desc(score))
)
```

questionId	userId	score	skill	Technology	type	location	lon	lat
208105	9021	7913	javascript	Full Stack	answer	Zürich, Switzerland	8.541694	47.37689
522563	19082	5655	python	Python & C++	answer	Switzerland	8.227512	46.81819
522563	19082	5655	list	Microsoft Stack	answer	Switzerland	8.227512	46.81819
2594829	27535	2825	sql	Microsoft Stack	answer	Switzerland	8.227512	46.81819
7892334	13302	2418	sql-server	Microsoft Stack	answer	Bern, Switzerland	7.447447	46.94797
7892334	13302	2418	tsql	Python & C++	answer	Bern, Switzerland	7.447447	46.94797
8710619	521799	2397	java	Full Stack	answer	St. Gallen, Schweiz	9.376717	47.42448
8710619	521799	2397	casting	Software Engineering	answer	St. Gallen, Schweiz	9.376717	47.42448
805547	24587	2289	ios	Web Frontend & Mobile	answer	Liestal, Schweiz	7.733427	47.48661
805547	24587	2289	objective-c	Web Frontend & Mobile	answer	Liestal, Schweiz	7.733427	47.48661
805547	24587	2289	cocoa-touch	iOS Stack	answer	Liestal, Schweiz	7.733427	47.48661

1.3.5 The How: rating user skills

One of the biggest challenges in this project was without any doubt to come up with a clean approach to assign skill ratings to users. First because there is no explicit link between users and skills and second because there isn't any apparent way to quantify a rating for a given user and skill. From the data exploration we know that questions contain `tags` (i.e. skills) and they also contain the posting `userId`. We also know that answers link to the parent question via the `questionId` and to the posting user via the `userId`. Therefore, through questions and answers we can link users and skills; namely the questions asked by an user: `user -> question -> tags` and the answers posted by an user: `user -> answer -> question -> tags`.

But what about the ratings? This is where the `badges` dataset comes into play. Badges[12] are awarded to users for different reasons including how good an answer or question is, this "how good" is backed up by a quantity which is the answer or question score (i.e. originating from the up or down votes), and thus we have a possible solution. The idea for filling the ratings would be to follow the same ordering provided by the badges system which is categorized with three "quality" class levels: `gold`, `silver` and `bronze`. We'd intuitively assume that e.g. an user that posted an answer which was awarded with `gold` for a question related to certain skills should be rated higher in those skills than an user asking a `silver` question on those same skills. But, would this badges ordering ensure significantly higher quality users? This is what we're about to find out in the following exploratory and visualization analysis.

We'd like to validate the hypothesis of whether the user quality would be significantly higher or better given that he has been awarded a certain level badge. One possible way to do this is to use the user reputation which is an overall quantity calculated independently of specific questions and answers. We wouldn't want a model fed with "lucky" users landing with a very high rating for a skill. We'd also like to validate the ordering i.e. is the reputation of users awarded with gold answer badges in average significantly higher than that of users with silver answer or question badges?

Here we take a look at the average reputations for users who have been granted the different badges of interest. The result gives us a rough idea of the order we are after. Note that we previously saw the users reputation to be highly skewed so we choose the median as measure of central tendency instead of the mean. The result reveals that users granted with answer badges depict in average higher reputations than those granted question badges. We also note that users granted gold badges have in average higher reputation than that of users granted silver

[12] For a detailed description of the badge system see https://stackoverflow.com/help/badges.

badges, similarly users granted silver badges tend to have higher reputations in average than users granted bronze badges.

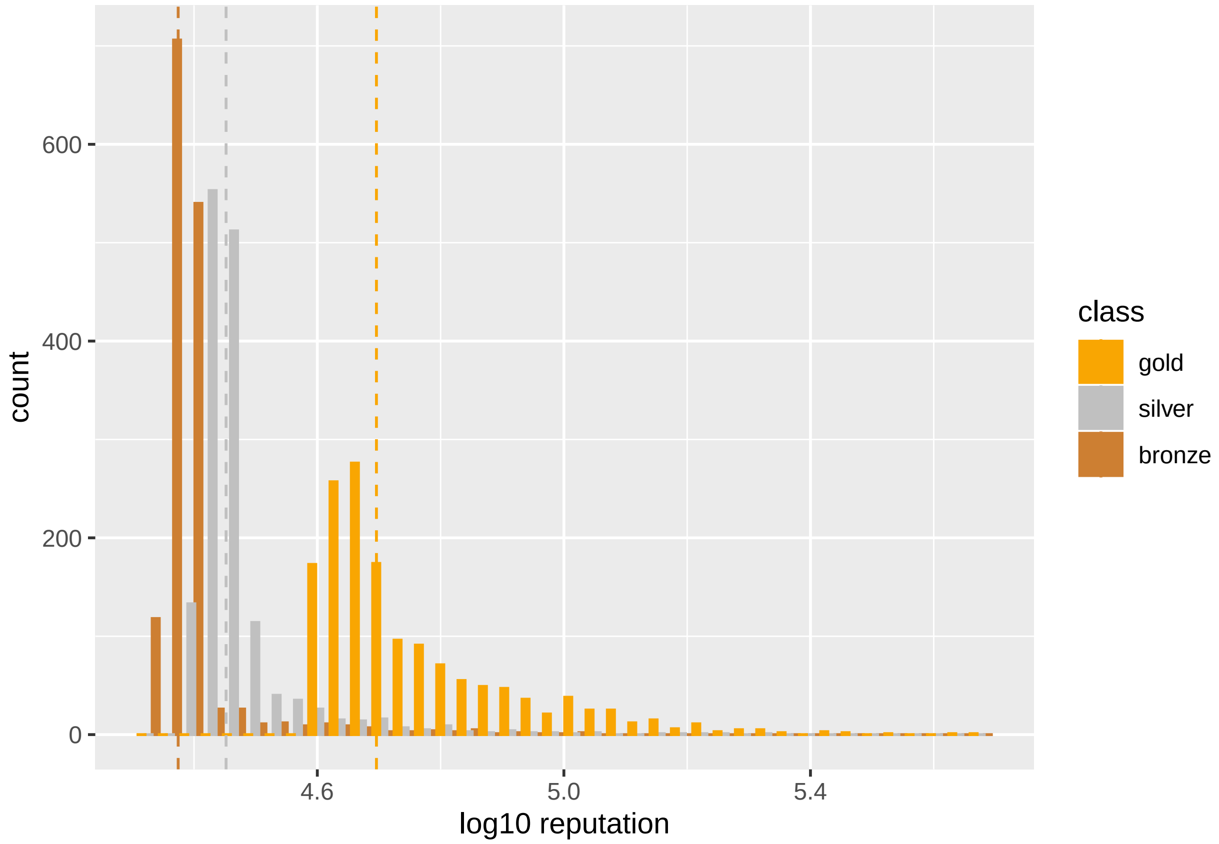
```r
# checkout the average reputation ordering by badge to get an idea though
# this is not the exact final ordering used due to the exclusion system.
prettyPrint(
  users %>%
    inner_join(badges %>% select(userId, class, badge) %>% unique(), by="userId") %>%
    filter(badge %in% badgesOrder) %>%
    group_by(class, badge) %>%
    summarise(avg_reputation=median(reputation)) %>%
    arrange(desc(avg_reputation))
, latex_options = c("striped"))
```

class	badge	avg_reputation
gold	Populist	6158.5
gold	Great Answer	5866.0
silver	Guru	5460.0
gold	Great Question	5178.0
silver	Good Answer	3146.5
silver	Good Question	3031.0
bronze	Nice Answer	2580.0
bronze	Nice Question	2544.0

Note that since the `log` transformation preserves the order of the data i.e. if $x > y$ then $\log(x) > \log(y)$ and brings it to a nicer scale (e.g. for plotting) we're going to do conduct the following analysis using a `log10` transformation of the users reputation.

The following plot reveals the user reputation ordering difference between gold, silver and bronze for answer badges. We see that the average within each group matches the expected level of the class i.e. users granted gold answer badges depict higher reputation average than those granted with silver and bronze answer badges. The vertical dashed lines show the median for each class:

```r
# create color specification for the different badges
colorSpec <- c("#f9a602", "#c0c0c0", "#cd7f32")
names(colorSpec) <- c("gold", "silver", "bronze")

summaryRep <- comp %>%
  group_by(class, badge) %>%
  summarise(median=median(reputation))
comp %>%
  filter(badge %in% c("Great Answer", "Good Answer", "Nice Answer")) %>%
  ggplot(aes(reputation, colour = class, fill = class, group = badge)) +
  xlab(label = "log10 reputation") +
  geom_histogram(position = "dodge", bins = 40) +
  scale_fill_manual(values = colorSpec) +
  scale_colour_manual(values = colorSpec) +
  geom_vline(data=summaryRep %>% filter(badge %in% c("Great Answer", "Good Answer",
                                                      "Nice Answer")),
             aes(xintercept=median, color=class), linetype="dashed")
```



The results depicted above are in a way incomplete. In order to assign skill ratings to users we need to also exclude users that were previously rated on those skills i.e. there should be no duplicate and ambiguous rating for the same user and skill. But how do we choose the rating among all possible? We can agree on the following strategy: an user reaching a higher rating level with respect to a skill, such higher rating takes precedence over other possible ratings on that skill. That's it, an user is rated with the highest rating we have observed among all posts that relate to that skill. Therefore, we find the users in the highest badge and class level for a skill and assign the highest rating e.g. 5.0. Then, excluding those users that were already rated in those skills, we find the users in the next highest level for that skill and assign the second highest rating e.g. 4.5 and so on. Therefore, we'd implement an iterative exclusion process that grants user with ratings from 5.0 to 1.5 in steps of 0.5 i.e. `seq(5.0, 1.5, by=-0.5)`.

We'd like to assses the significance of average reputation differences per badge and class group and for the top N users using this exclusion process. We choose the top N samples per group because this is the only way (relatively quick computing time-wise) to ensure that users from the top of higher-rated groups do not slip randomly into lower-rated groups. We also prefer not to do the exclusion process exhaustively for all the data because it would take a long time to compute. We should note that picking the top N users by reputation in each badge and class group invalidates the independence assumption required to resort to the Cental Limit Theorem (CLT)[13] for doing statistical inference i.e. building confidence intervals to compare two (actually all groups pair-wise) population averages i.e. in this case we're not choosing in-group samples randomly. Furthermore, for skewed data, the measure of central tendency of choice is the sample median and standard errors can not be calculated for distribution-free statistics. That said, and in order to compare the reputation medians among the different groups, we construct the 95% confidence intervals using the nonparametric bias-corrected and accelerated bootstrap interval BC_a, a statistically robust algorithm for producing highly accurate confidence

[13] https://en.wikipedia.org/wiki/Central_limit_theorem

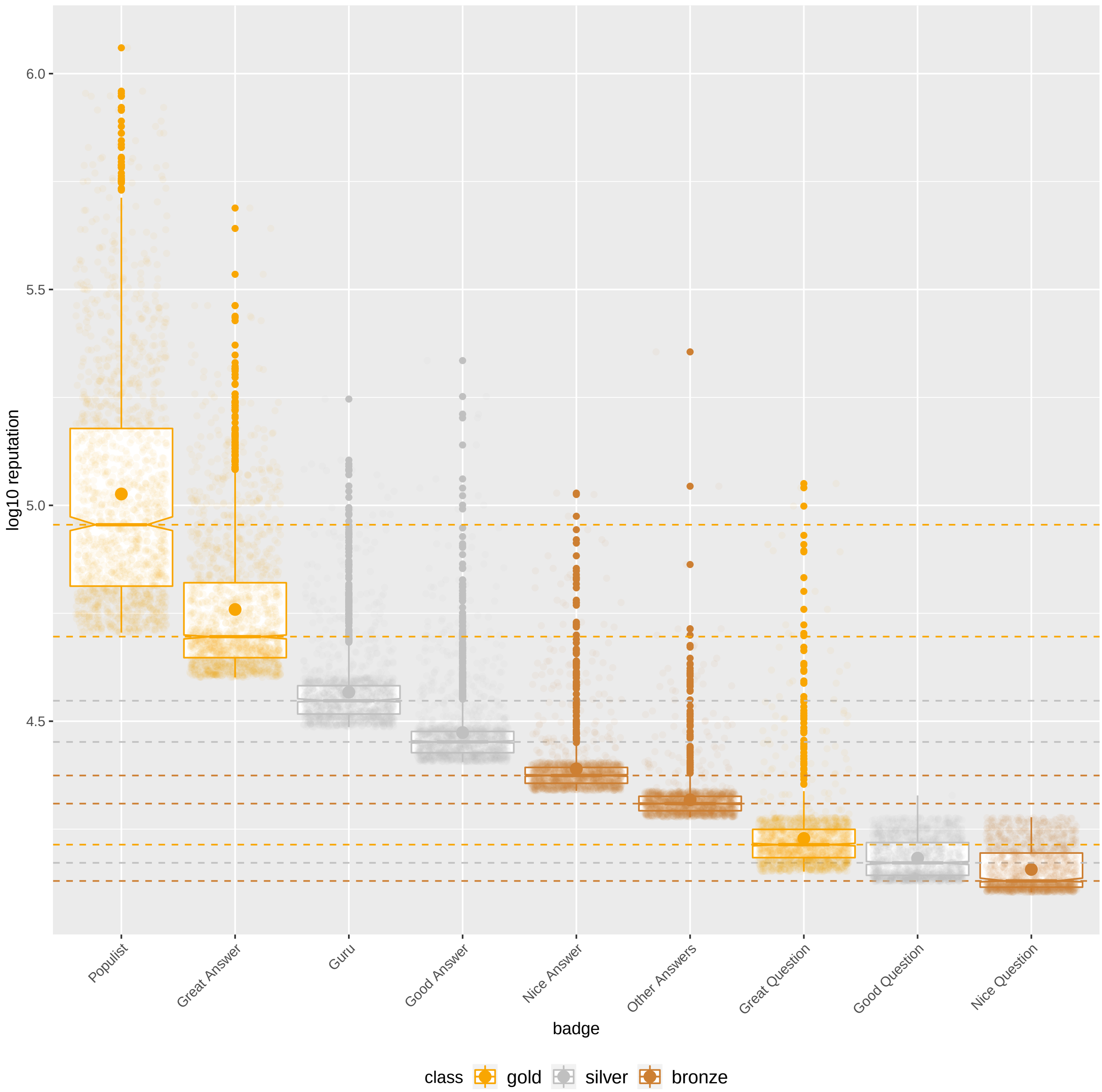
limits from a boostrap distribution see (DiCiccio and Efron 1996) and (Davison and Hinkley 1997).

The following listing compares the median of the reputation for the different groups and for the top N users using the BC_a bootstrap method. We required extending `ggplot2` with custom notches (a confidence interval feature for boxplots) computed using the BC_a method[14]. The boxplots are also enriched with the mean statistic (the solid circle shape) and dashed lines through the median for each group. With the guiding help of the dashed lines we observe that no confidence interval notches overlap and therefore the median reputation for each group can be assumed to be significantly different, higher (i.e. better) or lower than the others. A new non official Stack Overflow badge was introduced with name "Other Answers" to account for answers with scores below the levels that are officially awarded by Stack Overflow but still relevant to our ratings because answers, despite lower in score, still carry more weight than questions:

```
# bootstrap the median using 2x replications for the 1.5k users
B <- 2*N

# extend ggplot with a new boxplot stat_summary function to use bootstrapped BCa
# for the notch confidence intervals
bootNotch <- function(values) {
  # usual quantile values
  res = data.frame(t(boxplot(values, plot=FALSE)$stats))
  colnames(res) = c("ymin","lower","middle","upper","ymax")

  # bootstrap and get lower + upper notches
  ci <- boot.ci(boot(values, statistic = function(x, index) median(x[index]), R=B),
                type="bca")
  res$notchlower = ci$bca[4]
  res$notchupper = ci$bca[5]
  return(res)
}

# since we're extending the boxplot, need to provide outliers calculation as well
outlierNotch <- function(values) {
  return(boxplot(values, plot=FALSE)$out)
}

# compute a summary frame of the data containing mean and median
summaryRep <- comp %>%
  group_by(badge, class) %>%
  summarise(mean=mean(reputation),
            median=median(reputation))

# set the seed again (we need it here to get predictable bootstrap results)
portable.set.seed(1)
# plot the badge & class combinations to match an ordering for the ratings
comp %>%
  left_join(summaryRep, by=c("class", "badge")) %>%
  ggplot(aes(x=badge, y=reputation, colour=class, group=badge)) +
  ylab(label = "log10 reputation") +
  theme(legend.position="bottom", plot.title = element_text(hjust = 0.5),
        legend.text=element_text(size=12),
        axis.text.x = element_text(angle = 45, hjust = 1)) +
  stat_summary(fun.data = bootNotch, geom = "boxplot", notch = T) +
  stat_summary(fun.y = outlierNotch, geom = "point") +
  stat_summary(fun.y = mean, geom = "point", shape = 20, size = 5) +
```

[14] See the question https://stackoverflow.com/questions/59504775/

```r
  scale_colour_manual(values = colorSpec) +
  geom_hline(data=summaryRep, aes(yintercept = median, color = class),
             linetype = "dashed") +
  geom_jitter(alpha=0.05)
```



No other ordering of badges and class levels produces the monotonically decreasing median and mean reputation distribution for the users depicted in the figure above, therefore we use that ordering as reference to build our ratings dataset. Also note that answer badges are higher than question badges in average reputation regardless of the class level. The final ordering is summarized in the following table[15]. Note that we filter the last three rating levels “Other Questions” by `viewCount` being greater than 2.5k views, and thus we only consider question that have attracted certain level of interest:

[15] For more information on the Stack Overflow badges see https://stackoverflow.com/help/badges

Badge	Class	Rating	Conditions
Populist	gold	5.0	
Great Answer	gold	5.0	
Guru	silver	4.5	
Good Answer	silver	4.5	
Nice Answer	bronze	4.0	
Other Answers		3.5	Answers with score between [5, 10)
Other Answers		3.5	Answers with score between [0, 5)
Great Question	gold	3.0	
Good Question	silver	2.5	
Nice Question	bronze	2.5	
Other Questions		2.5	Questions with score between [5, 10) & viewCount > 2500
Other Questions		2.0	Questions with score between [2, 5) & viewCount > 2500
Other Questions		1.5	Questions with score between [0, 2) & viewCount > 2500

The code for generating the `ratings` dataset is implemented in the second half of the `create_dataset.r` script. Assembling the `ratings` dataset proved to be a very challenging task on its own too because it required first replicating the badge selection criteria documented in the definition of the badges and more importantly because of bringing the tags (or skills) into first normal form[16] i.e. tags are pipe-separated and we need them in a tag per row format instead, to be able to use relational operators. For this purpose, we used the function `tidyr::separate_rows` e.g. `tidyr::separate_rows(tags, sep="\\|")`. However, the expansion of millions of rows leads to hundreds of millions of rows which in a single thread would take a very long time to compute and easily overflow my 32GB system RAM. Therefore we again leveraged on the multi-core parallel architecture[17] and a blocking strategy to come up with a parallel and memory-bound divide and conquer approach which can be found in the `create_dataset.r` function `parBlockSeparate(...)` implementation. We can control the computation time and memory used with the parameters `ncore` and `blockSize` respectively.

We successfully compiled the `ratings` dataset and it includes the following main columns: `userId`, `skill` and `rating`; other columns were included for debugging, traceability[18] and as model features e.g. `firstPostDate` which will be discussed later.

```r
# read the ratings dataset
ratings <- readObjectByName("Ratings")

prettyPrint(head(glimpse(ratings)))
```

```
## Observations: 5,442,922
## Variables: 7
## $ userId        <dbl> 1, 1, 1, 1, 1, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13, ...
## $ skill         <chr> "database-design", "linq", "sql", "sql-server", "tags",...
## $ firstPostDate <dttm> 2008-09-07 19:17:04, 2008-08-11 19:23:47, 2008-09-07 1...
## $ creationDate  <dttm> 2008-09-07 19:17:04, 2008-12-01 04:56:38, 2008-09-07 1...
## $ postId        <int> 48692, 330073, 48692, 330073, 48692, 2383642, 1100426, ...
## $ postType      <fct> answer, answer, answer, answer, answer, answer, answer,...
## $ rating        <dbl> 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5...
```

[16] See https://en.wikipedia.org/wiki/First_normal_form

[17] Using `parallel::mclapply(...)` see https://stat.ethz.ch/R-manual/R-devel/library/parallel/html/mclapply.html

[18] To find which question or answer lead an user to receive a given rating.

userId	skill	firstPostDate	creationDate	postId	postType	rating
1	database-design	2008-09-07 19:17:04	2008-09-07 19:17:04	48692	answer	5
1	linq	2008-08-11 19:23:47	2008-12-01 04:56:38	330073	answer	5
1	sql	2008-09-07 19:17:04	2008-09-07 19:17:04	48692	answer	5
1	sql-server	2008-08-14 03:13:56	2008-12-01 04:56:38	330073	answer	5
1	tags	2008-09-07 19:17:04	2008-09-07 19:17:04	48692	answer	5
13	add	2010-03-05 00:03:25	2010-03-05 00:03:25	2383642	answer	5

We can see the measures of central tendency and dispertion of the ratings using the following code:

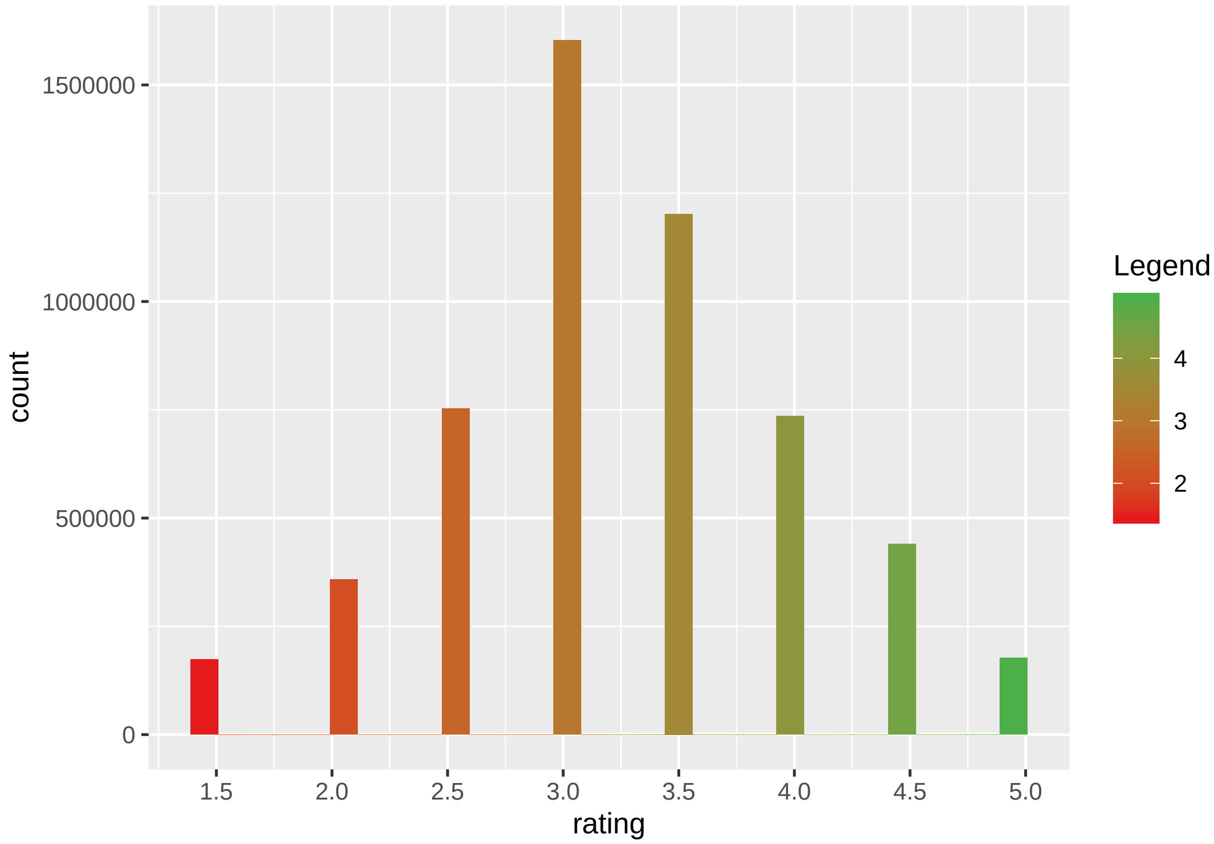
```r
# check the average
prettyPrint(
  ratings %>%
    summarise(median = median(rating), mean = mean(rating), sd = sd(rating))
, latex_options = c("striped"))
```

median	mean	sd
3	3.249211	0.782492

The following histogram plot depicts the distribution of the `ratings` dataset. We see that the lowest rating is 1.5 and goes by steps of 0.5 all the way to 5.0:

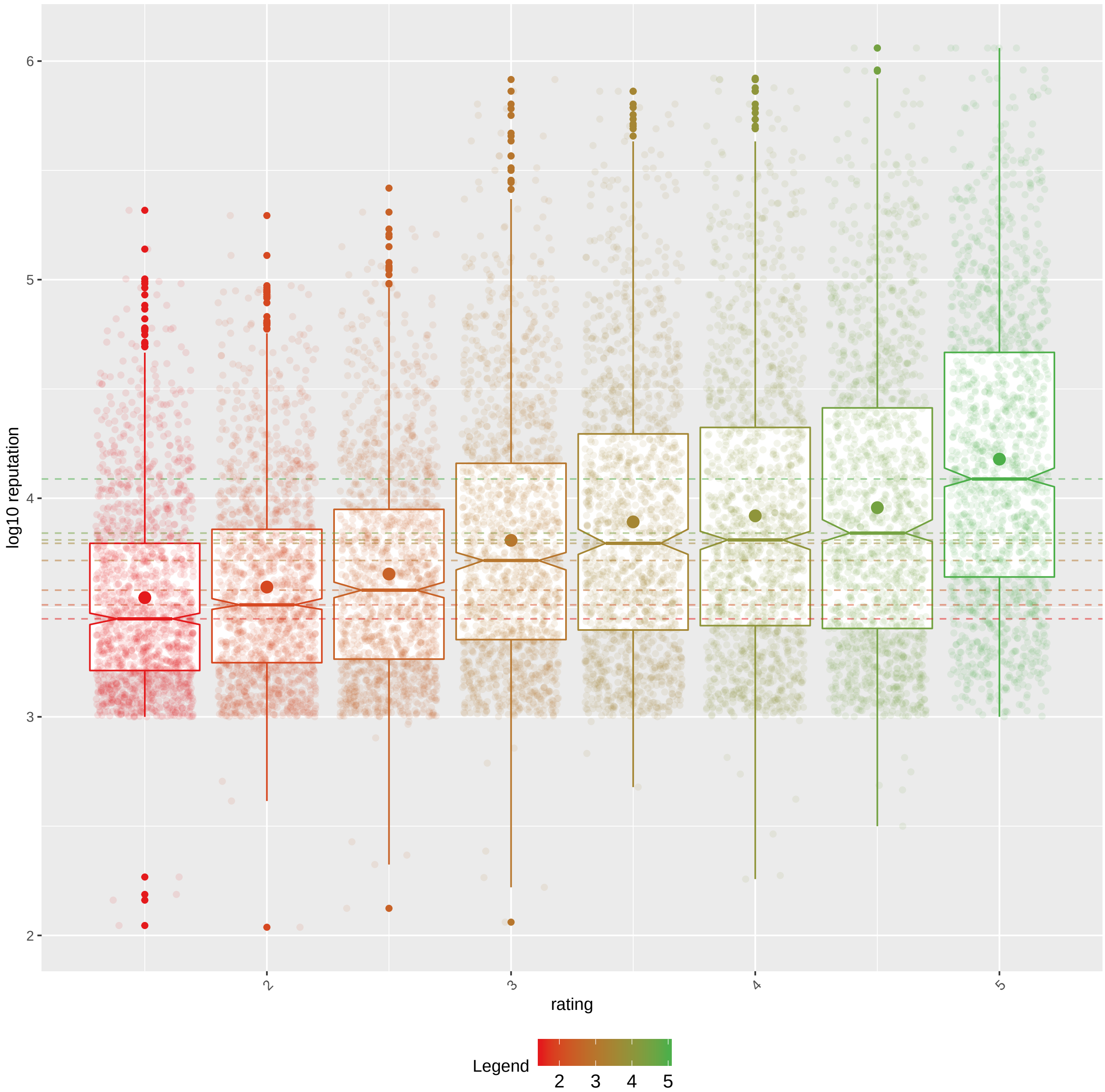
```r
# checkout the ratings histogram, it's nicely bell shaped
ratings %>%
  ggplot(aes(rating, fill=..x..)) + geom_histogram() +
  scale_x_continuous(breaks = seq(1.5, 5, by=0.5)) +
  scale_fill_gradient("Legend", low = "#E41A1C", high = "#4DAF4A")
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



At this point we can also assess how significantly different our final ratings are with respect to the users average reputation. Now we can take N random samples from each rating group since the exclusion process was achieved exhaustively. We again use the bias-corrected and accelerated bootstrap interval BC_a to check for significance. We see that several rating groups do not seem to be significatively different with respect to user reputation averages; namely, the BC_a confidence interval notches clearly overlap for ratings 3, 3.5, 4, and 4.5:

```
# set the seed again
portable.set.seed(1)
comp %>%
  ggplot(aes(x=rating, y=reputation, colour=rating, group=rating)) +
  ylab(label = "log10 reputation") +
  theme(legend.position="bottom", plot.title = element_text(hjust = 0.5),
        legend.text=element_text(size=12),
        axis.text.x = element_text(angle = 45, hjust = 1)) +
  stat_summary(fun.data = bootNotch, geom = "boxplot", notch = T) +
  stat_summary(fun.y = outlierNotch, geom = "point") +
  stat_summary(fun.y = mean, geom = "point", shape = 20, size = 5) +
  scale_color_gradient("Legend", low = "#E41A1C", high = "#4DAF4A") +
  geom_hline(data=summaryRep, aes(yintercept = median, color = rating),
             linetype = "dashed", alpha=0.5) +
  geom_jitter(alpha=0.1)
```



Visually comparing the confidence intervals for possible overlap is a necessary condition but often not a sufficient condition of significance. Therefore, we run the Wilcoxon rank sum test[19] (see Bauer 1972) for independent samples i.e. to determine the significance of the difference in population median between unpaired groups. The Wilcoxon test is nonparametric and doesn't require the normality assumption of the data which is exactly our case here. The following listing executes the Wilcoxon test on all ratings pair-wise. The results reveals that the median of the reputations for the groups corresponding to rating pairs `3.5-4.0` and `4.0-4.5` are not significatively different, whereas all other rating pairs are significatively different at 95% confidence level. However, we note that pair `4.0-4.5` is significatively different at 82% confidence level:

```
# generate all possible pair-wise rating combinations
c <- combn(seq(1.5, 5.0, by=0.5), m=2)
# print the pairs whose p-value is greater than 0.05 i.e.
# the median difference between groups is not significative at
```

[19]https://stat.ethz.ch/R-manual/R-devel/library/stats/html/wilcox.test.html

```r
# 95% confidence
for (i in 1:ncol(c)) {
  res <- comp %>%
    filter(rating == c[1,i] | rating == c[2,i]) %>%
    wilcox.test(reputation~rating, data=., paired=F, conf.int=T)
  if (res$p.value > 0.05) {
    cat(sprintf(paste0("Median rep. for users with rating ",
                       "%.1f and %.1f is NOT sig. ",
                       "diff. with p-value=%.5f\n"),
                c[1,i], c[2,i], res$p.value))
  }
}
```

```
## Median rep. for users with rating 3.5 and 4.0 is NOT sig. diff. with p-value=0.42574
## Median rep. for users with rating 4.0 and 4.5 is NOT sig. diff. with p-value=0.17431
```

Chapter 2

Modeling approach

In this section we'll build a recommender system for predicting user-skill ratings using the collaborative filtering (CF) technique. We'll explore a simpler baseline method that discounts multiple effects or bias `b`'s and then move to the more advanced model-based latent factor low-rank matrix factorization (LRMF) method implemented using the stochastic gradient descent (SGD) algorithm. The loss function we'll employ to evaluate the predictions in both cases is the root mean squared error (RMSE) where $r_{i,j}$ is the true rating and $\widehat{r_{i,j}}$ is our predicted user skill rating for user i and skill j, we'll reuse the `Metrics::rmse(...)` implementation:

$$\text{RMSE} = \sqrt{\frac{1}{N}\sum_{i,j}(\hat{r}_{i,j} - r_{i,j})}$$

Let's get some basic information about the `ratings` dataset e.g. sparsity. We have 199.8k users and 2k skills[1] and a highly sparse dataset i.e. only about 1.4% of all the possible user skill ratings:

```
# what's the number of unique users, skills and how sparse is it?
prettyPrint(
  ratings %>%
    summarise(users = n_distinct(userId),
              skills = n_distinct(skill),
              n = n()) %>%
    mutate(sparsity=sprintf("%.1f%%", 100*n / (users*skills))) %>%
    select(users, skills, sparsity)
, latex_options = c("striped"))
```

users	skills	sparsity
199840	2000	1.4%

We start by separating the `ratings` dataset into training and test sets as shown in the following listing. We will use the train set for calibration (i.e. cross validation) and training and the test set exclusively for out-of-sample evaluation of the models:

```
# split the ratings dataset into separate train and test sets
portable.set.seed(1)
testIndex <- createDataPartition(y = ratings$rating, times = 1,
                                 p = 0.1, list = FALSE)
trainSet <- ratings[-testIndex,]
tmp <- ratings[testIndex,]
```

[1]These are the top 2k skills we have used before that account for 82.3% of the total taggings.

```r

# make sure userId and skill in test set are also in train set
testSet <- tmp %>%
  semi_join(trainSet %>% select(userId) %>% unique(), by="userId") %>%
  semi_join(trainSet %>% select(skill) %>% unique(), by="skill")

# add rows removed from test set back into the train set
removed <- tmp %>%
  anti_join(testSet, by=c("userId", "skill"))

trainSet <- trainSet %>%
  bind_rows(removed)
rm(tmp, removed)

# test the results, the two sets must add up
stopifnot(nrow(trainSet) + nrow(testSet) == nrow(ratings))

# how many rows in the training set?
nrow(trainSet)
```

```
## [1] 4898883
```

```r
# how many rows in the test set?
nrow(testSet)
```

```
## [1] 544039
```

2.1 Baseline

To get a benchmark and idea for what level of RMSE we can reach, we start by exploring the simpler but very effective baseline model. In our model we'll account for the ratings' gobal mean μ, user effects b_i, skill effects b_j, and in addition a possible way to model temporal effects: smoothing the amount of week blocks since an user first posted about a specific skill to the time of the post rating or, put more intuitively, the elapsed time since an user started gaining experience in a skill to the time of the current rating. We call this effect the user skill experience effect b_e. We'll also employ regularization λ to penalize the contribution of effects with few elements i.e. the penalized least squares. Our baseline model is then specified using the following cost function:

$$J_{\text{baseline}} = \frac{1}{N}\sum_{i,j}(r_{i,j} - (\mu + b_i + b_j + f_{\text{smooth}}(b_e)))^2 + \lambda\left(\sum_i b_i^2 + \sum_j b_j^2\right)$$

We start building the baseline model with just the average. Note that we are using only the training set:

```r
# global average
mu <- mean(trainSet$rating)
# compute predictions and RMSE
rmseResults <- tibble(method = "Just the average",
                      RMSE = Metrics::rmse(trainSet$rating, mu))
prettyPrint(
  rmseResults
, latex_options = c("striped"))
```

method	RMSE
Just the average	0.7824687

We extend our baseline model to account for the user effects b_i and using a $\lambda = 4$[2]:

```
# set the regularization parameter
lambda <- 4
# compute regularized user effects
userEffects <- trainSet %>%
  group_by(userId) %>%
  summarize(b_i = sum(rating - mu)/(n() + lambda))
# compute predictions and RMSE
predictedRatings <- trainSet %>%
  left_join(userEffects, by='userId') %>%
  mutate(pred=mu + b_i) %>%
  pull(pred)
rmseResults <- bind_rows(rmseResults,
                         tibble(method="Regularized User Effects",
                                RMSE = Metrics::rmse(predictedRatings,
                                                     trainSet$rating)))
prettyPrint(
  rmseResults
, latex_options = c("striped"))
```

method	RMSE
Just the average	0.7824687
Regularized User Effects	0.6442360

We extend the baseline model to account for the skill effects too b_j:

```
# compute regularized skill effects
skillEffects <- trainSet %>%
  left_join(userEffects, by='userId') %>%
  group_by(skill) %>%
  summarize(b_j = sum(rating - (mu + b_i))/(n() + lambda))
# compute predictions and RMSE
predictedRatings <- trainSet %>%
  left_join(userEffects, by='userId') %>%
  left_join(skillEffects, by='skill') %>%
  mutate(pred=mu + b_i + b_j) %>%
  pull(pred)
rmseResults <- bind_rows(rmseResults,
                         tibble(method="Regularized User + Skill Effects",
                                 RMSE = Metrics::rmse(predictedRatings,
                                                      trainSet$rating)))
prettyPrint(
  rmseResults
, latex_options = c("striped"))
```

[2] A few values of λ were manually tried and $\lambda = 4$ was the best.

method	RMSE
Just the average	0.7824687
Regularized User Effects	0.6442360
Regularized User + Skill Effects	0.6304935

At this point we are in a place to account for more interesting effects. We consider the timestamp `firstPostDate` when an user made a post connected to a skill for the first time. We include this timestamp in every rating since it's an attribute that applies to every unique `userId`, `skill` combination and it's the input needed for generating the new feature: elapsed time in week blocks since the first time the user posted about the rated skill. The elapsed time in week blocks is calculated using the code `week_block_30 = ceiling(as.duration(firstPostDate %--% creationDate) / dweeks(weeksBlock)))` with the help of `lubridate` package's functions `lubridate::as.duration`[3], the interval creation operator `%--%`[4] and duration in weeks `lubridate::dweeks`[5].

The following plot computes the effects b_e after accounting for all other previous effects and illustrates how strong this gained experience temporal effect is. We note how the effect has a relatively big range and a potential impact to the RMSE of approximately 0.4. We should also note how interesting this is, from the time the user first starts looking at a skill, the effect starts increasing until it reaches a peak on approximately 2.5 times 30 week blocks i.e. approximately 1.5 years; according to our plot below this is the average experience time needed for an user to reach her peak rating for a skill. After the peak rating is reached, we notice that the experience gained effect reaches a plateau and starts to fluctuate meaning users either: continue improving that skill or move on to other skill topics or simply the skill becomes irrelevant. There could also be a changing job or projects effect involved:

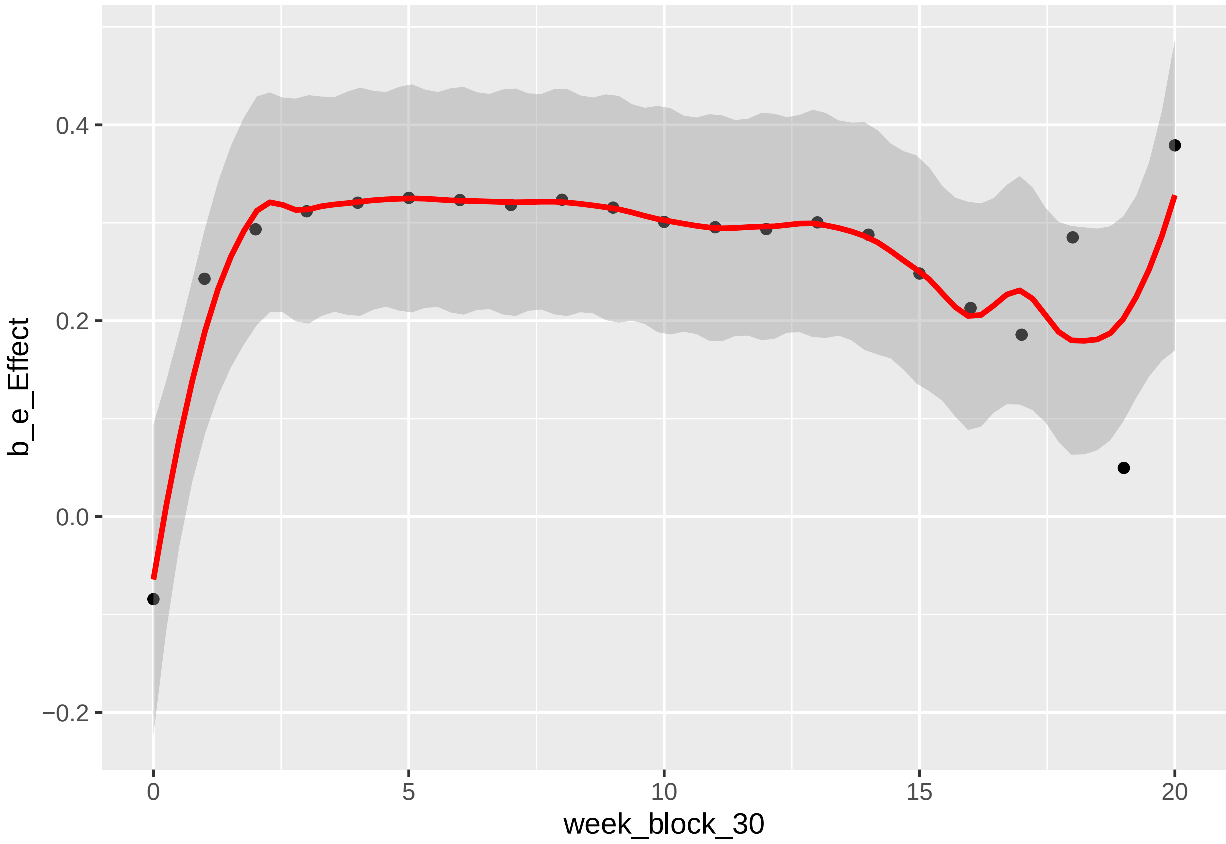
```
# show the effects of number of week blocks since first post
weeksBlock <- 30
# this week blocks corresponds approximately to 7 months
round(weeksBlock / 4.34524)
```

```
## [1] 7
```

```
# show the gaining experience over time effects
trainSet %>%
  left_join(skillEffects, by='skill') %>%
  left_join(userEffects, by='userId') %>%
  mutate(residual=rating - (mu + b_i + b_j)) %>%
  mutate(week_block_30 = ceiling(
    as.duration(firstPostDate %--% creationDate) / dweeks(weeksBlock))) %>%
  arrange(desc(week_block_30)) %>%
  group_by(week_block_30) %>%
  summarise(b_e_Effect=mean(residual)) %>%
  ggplot(aes(week_block_30, b_e_Effect)) + geom_point() +
  geom_smooth(color="red", span=0.3, method.args=list(degree=2))
```

```
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```

[3] See https://lubridate.tidyverse.org/reference/duration.html
[4] See https://lubridate.tidyverse.org/reference/interval.html
[5] See https://lubridate.tidyverse.org/reference/duration.html



We then add the b_e effect to our model:

```
# fit a loess smoothing to model the "experience gained" temporal effect
weeksBlockFit <- trainSet %>%
  left_join(userEffects, by='userId') %>%
  left_join(skillEffects, by='skill') %>%
  mutate(residual=rating - (mu + b_i + b_j)) %>%
  mutate(week = ceiling(
    as.duration(firstPostDate %--% creationDate) / dweeks(weeksBlock))) %>%
  group_by(week) %>%
  summarise(residual=mean(residual)) %>%
  loess(residual~week, data=., span=0.3, degree=2)

# compute predictions and RMSE
predictedRatings <- trainSet %>%
  left_join(userEffects, by='userId') %>%
  left_join(skillEffects, by='skill') %>%
  mutate(week = ceiling(
    as.duration(firstPostDate %--% creationDate) / dweeks(weeksBlock))) %>%
  mutate(pred=mu + b_i + b_j + predict(weeksBlockFit, .)) %>%
  pull(pred)
rmseResults <- bind_rows(rmseResults,
                         tibble(method="Regularized User + Skill + Experience Effects",
                                RMSE = Metrics::rmse(predictedRatings,
                                                     trainSet$rating)))
prettyPrint(
  rmseResults
, latex_options = c("striped"))
```

method	RMSE
Just the average	0.7824687
Regularized User Effects	0.6442360
Regularized User + Skill Effects	0.6304935
Regularized User + Skill + Experience Effects	0.6126687

We reached an impressive RMSE=`0.6126687` in-sample. Let's see how our baseline model performs out-of-sample using the test set:

```
## TEST SET ACCESS ALERT! accessing the test set to compute RMSE.
predictedRatings <- testSet %>%
  left_join(userEffects, by='userId') %>%
  left_join(skillEffects, by='skill') %>%
  mutate(week = ceiling(
    as.duration(firstPostDate %--% creationDate) / dweeks(weeksBlock))) %>%
  mutate(pred=mu + b_i + b_j + predict(weeksBlockFit, .)) %>%
  pull(pred)
rmseValue <- Metrics::rmse(predictedRatings, testSet$rating)
cat(sprintf("baseline RMSE on test data is %.9f\n", rmseValue))
```

```
## baseline RMSE on test data is 0.632744751
```

```
# check that we get reproducible results
stopifnot(abs(rmseValue - 0.632744751) < 1e-9)
```

We reached an impressive out-of-sample RMSE=`0.632744751`. Note that we haven't calibrated this baseline model properly, we've simply manually experimented with the hyper-parameters: λ, `weeksBlock`, loess `span` and `degree` and found the following best hyper-parameters combination for illustrative purposes:

- Lambda $\lambda = 4$
- `weeksBlock=30` number of week blocks between user's first exposure to the skill and the time of the rating.
- Loess temporal model smoothing parameter `span=0.3`.
- Loess temporal model `degree=2`.

Although we find this modeling approach truly fascinating, we could keep on exploring, adding features and accounting for more interesting effects that would lower our RMSE even further[6]; adding so many features increases model complexity and hinders usability in practice. Notice that in order to predict user-skill ratings using our baseline model we require the user to inform us with the elapsed time in 30 week blocks since she first started looking at a skill, as this is a required predictor variable of our model. At this point we have a baseline, reference RMSE and we can move on to a more advanced model that only requires the past ratings, nothing else, to learn and to make predictions.

2.2 Low-Rank Matrix Factorization

In this section we present a recommender system for predicting user skill ratings using the collaborative filtering (CF)[7] technique. More specifically we'll implement the model-based low-rank matrix factorization (LRMF) method see (Koren 2008) and (Koren, Bell, and Volinsky

[6] For example, add as feature the number of user posts for a skill or the number of answers to questions ratio for a skill, etc.

[7] https://en.wikipedia.org/wiki/Collaborative_filtering

2009). The principle is that there are latent structures in the data that once revealed, we have a low-dimensional representation we can use to make automatic predictions, in this case user skill rating predictions. The low-dimensional representation we obtain using LRMF is equivalent to computing the singular value decomposition SVD[8] on the dense ratings matrix representation. However, this later approach is impractical and prohibitive when the dimensions of the dense representation are too large and the system is very sparse as it's in our case.

Our algorithm will learn and encode a low-dimensional representation of the ratings within two matrices P and Q. P is a matrix of K latent rows (or features) and N columns corresponding to each distinct user. While Q is a matrix of K latent rows and M columns corresponding to each distinct skill. Note that we have encoded the two matrices in such a way that all matrix computations are done on the columns i.e. the dimensions corresponding to users and skills. Doing so we match R's default[9] column-major[10] order to achieve the best possible performance i.e. operate on contiguous memory and avoid costly memory striding operations. Our LRMF cost function is then defined as follows:

$$J_{P,Q} = \sum_{i,j} \left(r_{u,i} - P_i^T Q_j \right)^2 + \lambda \left(\sum_i \parallel P_i \parallel^2 + \sum_j \parallel Q_j \parallel^2 \right)$$

In order to find the P and Q that minimize our cost function $J_{P,Q}$ we use the stochastic gradient descent (SGD) algorithm. The gradient descent updates are found by deriving our cost function with respect to P_i (i.e. user i) and Q_j (i.e. skill j) respectively:

$$\begin{aligned}
\epsilon_{i,j} &= r_{i,j} - P_i^T Q_j \\
J_{P,Q} &= \sum_{i,j} \epsilon_{i,j}^2 + \lambda \left(\sum_i \parallel P_i \parallel^2 + \sum_j \parallel Q_j \parallel^2 \right) \\
&\underset{P,Q}{\operatorname{argmin}} \sum_{i,j} \epsilon_{i,j}^2 + \lambda \left(\sum_i \parallel P_i \parallel^2 + \sum_j \parallel Q_j \parallel^2 \right) \\
\frac{\partial J}{\partial P_i} &= -2\epsilon_{i,j} Q_j + 2\lambda P_i = -2(\epsilon_{i,j} Q_j - \lambda P_i) \Rightarrow \Delta P_i = \gamma(\epsilon_{i,j} Q_j - \lambda P_i) \\
\frac{\partial J}{\partial Q_j} &= -2\epsilon_{i,j} P_i + 2\lambda Q_j = -2(\epsilon_{i,j} P_i - \lambda Q_j) \Rightarrow \Delta Q_j = \gamma(\epsilon_{i,j} P_i - \lambda Q_j)
\end{aligned}$$

where γ is the learning rate. Therefore, in every SGD step the following P and Q updates are executed:

$$\begin{aligned}
P_i &= P_i + \gamma(\epsilon_{i,j} Q_j - \lambda P_i) \\
Q_j &= Q_j + \gamma(\epsilon_{i,j} P_i - \lambda Q_j)
\end{aligned}$$

[8] https://en.wikipedia.org/wiki/Singular_value_decomposition
[9] See https://cran.r-project.org/web/packages/reticulate/vignettes/arrays.html
[10] See https://en.wikipedia.org/wiki/Row-_and_column-major_order

Chapter 3

Method implementation

At this point we're ready to introduce the LRMF implementation described in section Low-Rank Matrix Factorization. We have identified the following model hyper-parameters:

- K: the number of features or latent dimensions.
- $\gamma_{\max}$: the maximum learning rate.
- λ: the regularizaton parameter.
- σ: standard deviation of the standard normal random initialization for P and Q i.e. for the model to learn.

Our implementation of the classic SGD algorithm is described as follows:

1. Pre-process the ratings to standard scale or z-scores saving μ and σ.
2. Initialize the matrices P and Q to be as close as possible to the learning goal (Rialland 2019).
3. For `maxIter` iterations of the algorithm run a number of batch updates and check the RMSE. If the RMSE worsens after the batch iterations then halve γ (see Rialland 2019) otherwise increase it more slowly to a maximum of $\gamma_{\max}$.
4. Run `batchIter` iterations of batch updates using `batchSize` random samples.
5. Store the final P and Q as part of the fit object and use it to make predictions.

In the second step of the algorithm, the matrix is initialized to be as close as possible to the learning goal[1] (Rialland 2019); this idea proved very useful in reaching very fast convergence. However, note that we have a specific representation of P and Q to align our matrix operations workload with R's default column-major ordering:

$$P^T Q = \begin{bmatrix} 1 & u_1 & 0 \\ 1 & u_2 & 0 \\ \vdots & \vdots & \vdots \\ 1 & u_i & 0 \\ \vdots & \vdots & \vdots \\ 1 & u_N & 0 \end{bmatrix} \times \begin{bmatrix} s_1 & s_2 & \cdots & s_j & \cdots & s_M \\ 1 & 1 & \cdots & 1 & \cdots & 1 \\ 0 & 0 & \cdots & 0 & \cdots & 0 \end{bmatrix} = \begin{bmatrix} & \vdots & \\ \cdots & s_j + u_i & \cdots \\ & \vdots & \end{bmatrix}$$

We also implemented a lock-free multi-core parallel[2] variation of the classic SGD algorithm that trains faster. The key idea is to work in parallel on mutually exclusive subsets of the P and Q

[1] See https://github.com/Emmanuel-R8/HarvardX-Movielens/raw/master/MovieLens.pdf

[2] Using `parallel::mclapply(...)` see https://stat.ethz.ch/R-manual/R-devel/library/parallel/html/mclapply.html

matrices, creating as many mutually exclusive subset combinations as there are cores available. We achieve that by first creating a blocks specification[3] e.g. for a 3-core system we have:

```
ncores <- 3
bi <- rep(0:(ncores-1), ncores)
bj <- (bi + rep(0:(ncores-1), 1, each = ncores)) %% ncores
blocks <- tibble(bi=bi, bj=bj)
prettyPrint(
  blocks
, latex_options = c("striped"))
```

bi	bj
0	0
1	1
2	2
0	1
1	2
2	0
0	2
1	0
2	1

The blocks specification above give us 3 different block groups (every 3 consecutive rows) in i (users) and j (skills) to run in parallel and mutual exclusion. Therefore, selecting rows as described in the following listing we can filter the matrices P and Q for users and skills that match those blocks respectively, can each be assigned to a separate core and executed in parallel:

```
# first group of mutually exclusive blocks
prettyPrint(blocks %>% slice(1:3), latex_options = c("striped"))
```

bi	bj
0	0
1	1
2	2

In this case the first core will work on users and skills whose `i %% ncores == 0 & j %% ncores == 0` respectively, the second core will work on `i %% ncores == 1 & j %% ncores == 1` and the third core on `i %% ncores == 2 & j %% ncores == 2`. However, doing so will only train those block combinations. Therefore at every iteration of the algorithm and before running the batch updates, we randomly choose which of the three block groups to use, either: `blocks[1:3,]`, `blocks[4:6,]` or `blocks[7:9,]` and each core will run batch updates in a loop ensuring maximum CPU utilization. We would of course like to run long batches in parallel but doing so may lead the SGD algorithm advancing in the wrong direction without a chance for correction.

The following code listing corresponds to the core of the SGD algorithm implementation as part of the function `lrmf$fit(...)` and including both, the classic and parallel variations. We iterate `maxIter` times as shown in line `#1`. Lines `#3` through `#18` correspond to the classic implementation. There we have the gradient P and Q updates and all operations are computed

[3]See https://stackoverflow.com/questions/59154906

on the matrix columns as discussed before. Lines #20 through #80 correspond to the parallel implementation. There we pick a random block group and run `parIter` iterations on it. In lines #54 through #69 we accumulate the set of distinct updated indexes for the users and skills and return them along with the gradient updates. Finally the lines #75 through #80 “reduce” or consolidate the updates from all blocks into P and Q.

```
1  for (iter in 1:maxIter) {
2    if (ncores == 1) {
3      for (iter2 in 1:batchIter) {
4        # choose a random batch of samples
5        samples <- x %>% sample_n(batchSize)
6
7        # get hold of the indexes
8        i <- samples$i
9        j <- samples$j
10
11       # compute the residuals
12       epsilon <- samples$rating_z - colSums(P[,i]*Q[,j])
13
14       # partial derivatives w.r.t. P and Q
15       P_upd <- (P[,i] + gamma*(Q[,j]*epsilon[col(Q[,j])] - lambda*P[,i]))
16       Q[,j] <- (Q[,j] + gamma*(P[,i]*epsilon[col(P[,i])] - lambda*Q[,j]))
17       P[,i] <- P_upd
18     }
19   } else {
20     stopifnot(nrow(blocks) == ncores^2)
21
22     # pick a random block group
23     g <- sample(0:(ncores-1), 1)
24
25     # process the selected block group
26     res <- mclapply((g*ncores + 1):((g + 1)*ncores), mc.cores = ncores,
27                     mc.set.seed = TRUE,
28       FUN = function(b) {
29         # select the subset of samples corresponding to this block
30         blockSamples <- x %>%
31           filter(bi == blocks[b,]$bi & bj == blocks[b,]$bj)
32
33         # keep track of the updated columns
34         ii <- NULL
35         jj <- NULL
36
37         # run multiple batches on this block
38         for (iter2 in 0:(parIter-1)) {
39           samples <- blockSamples %>% sample_n(batchSize)
40
41           # get hold of the indexes
42           i <- samples$i
43           j <- samples$j
44
45           # compute the residuals
46           epsilon <- samples$rating_z - colSums(P[,i]*Q[,j])
47
48           # partial derivatives w.r.t. P and Q
49           P_upd <- (P[,i] + gamma*(Q[,j]*epsilon[col(Q[,j])] - lambda*P[,i]))
50           Q[,j] <- (Q[,j] + gamma*(P[,i]*epsilon[col(P[,i])] - lambda*Q[,j]))
51           P[,i] <- P_upd
52
```

```
            # accumulate the changed indexes
            if (is.null(ii)) {
              ii <- i
            } else {
              ii <- c(ii, i)
            }

            if (is.null(jj)) {
              jj <- j
            } else {
              jj <- c(jj, j)
            }
          }

          ii <- sort(unique(ii))
          jj <- sort(unique(jj))

          # output the updated user and skill columns
          return(list(Pii=P[,ii],Qjj=Q[,jj],ii=ii,jj=jj))
        })

      # consolidate updates into the P and Q matrices
      for (k in 1:length(res)) {
        l <- res[[k]]
        P[,l$ii] <- l$Pii
        Q[,l$jj] <- l$Qjj
      }
    }
```

Our SGD implementation is integrated with the popular machine learning R package `caret`[4] as a custom `lrmf` model[5], and thus we take advantage of all the caret infrastructure for calibration, training and prediction in addition to making our code easier to understand, maintain and reuse.

[4] See https://cran.r-project.org/web/packages/caret/

[5] See using your own model in train https://topepo.github.io/caret/using-your-own-model-in-train.html

Chapter 4

Results

Now we're ready to run our model implementation, first we need to create a calibration set (small subset of the training set):

```
tic("collecting the calibration set of 2k samples")
# set the seed again
portable.set.seed(1)
calibrationSet <- trainSet %>%
  sample_n(2000)
toc()
```

```
## collecting the calibration set of 2k samples: 2.263 sec elapsed
```

```
# how many distinct skills and users in the calibration set?
cat(sprintf("The calibration set contains %d unique skills and %d users\n",
            length(unique(calibrationSet$skill)),
            length(unique(calibrationSet$userId))))
```

```
## The calibration set contains 893 unique skills and 1955 users
```

Then we calibrate our model (i.e. run cross-validation) using the `caret` package to find the best hyper-parameters, we also check for reproducible results:

```
tic('calibrating the LRMF model')
# set the seed again
portable.set.seed(1)
control <- trainControl(method = "cv",
                        search = "grid",
                        number = 10,          # use 10 K-folds in cross validation
                        p = .9,               # use 90% of training and 10% for testing
                        allowParallel = T,    # execute CV folds in parallel
                        verboseIter = T)
cvFit <- train(x = calibrationSet,
               y = calibrationSet$rating,
               method = lrmf,
               trControl = control,
               ncores = 1,
               maxIter = 100,
               batchIter = 50,
               batchSize = 100)
```

```
## Aggregating results
## Selecting tuning parameters
## Fitting K = 10, maxGamma = 0.1, lambda = 0.05, sigma = 0.05 on full training set
```

```
toc()
```

```
## calibrating the LRMF model: 60.381 sec elapsed
```

```
## The bestTune model found is:
stopifnot(cvFit$bestTune$K == 10)
stopifnot(cvFit$bestTune$gamma == 0.1)
stopifnot(cvFit$bestTune$lambda == 0.05)
stopifnot(cvFit$bestTune$sigma == 0.05)
```

We can then train our model on all training data[1]. We train both the parallel and classic models using the exact same number of sample updates 27k (i.e. `maxIter*batchIter`) to compare them. The parallel is trained with half the `maxIter` compensated with doubling the `batchIter`. We see that the parallel implementation arrives to comparable RMSE in about half the time than the classic. However, by halving the number of iterations, the parallel version has half the opportunities to correct the gradient direction and this is confirmed in the convergence plot as we see it's RMSE more bumpy compared to the classic:

```
tic('training LRMF on the full training set - parallel')
# set the seed again
portable.set.seed(1)
fitPar <- train(x = trainSet,
                y = trainSet$rating,
                method = lrmf,
                trControl = trainControl(method = "none"),
                tuneGrid = cvFit$bestTune,
                trackConv = T,
                iterBreaks = 1,
                verbose = F,
                ncores = ncores,
                maxIter = 125,
                batchIter = 216)
ticTocTimes <- toc()
```

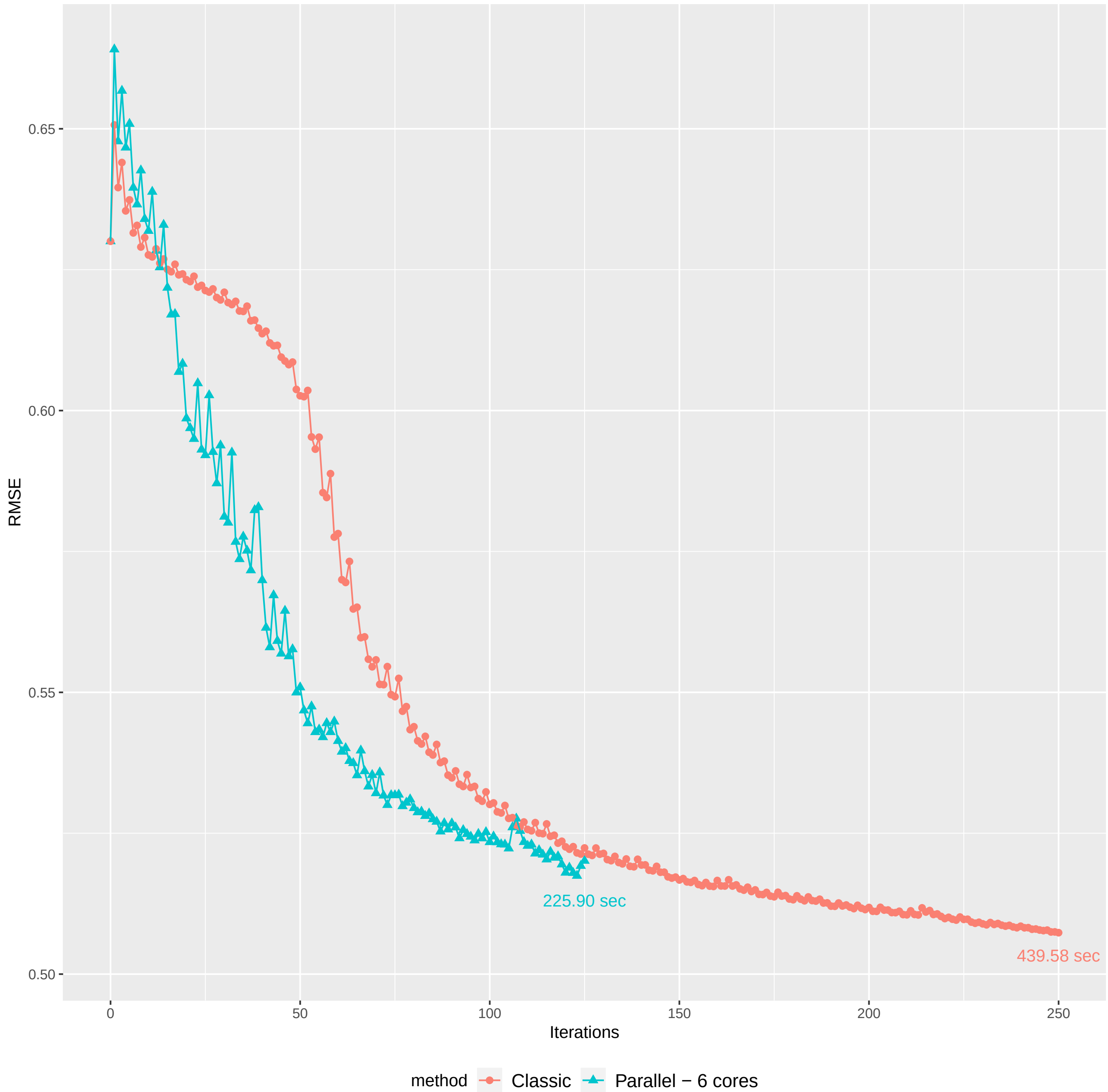
```
## training LRMF on the full training set - parallel: 225.9 sec elapsed
```

```
elapsedPar <- ticTocTimes$toc[[1]] - ticTocTimes$tic[[1]]

# save the RMSE history
rmseHist <- fitPar$finalModel$rmseHist %>%
  add_column(method=sprintf("Parallel - %d cores", ncores))

tic('training LRMF on the full training set - classic')
# set the seed again
portable.set.seed(1)
fitSeq <- train(x = trainSet,
                y = trainSet$rating,
                method = lrmf,
                trControl = trainControl(method = "none"),
                tuneGrid = cvFit$bestTune,
                trackConv = T,
                iterBreaks = 1,
                verbose = F,
                ncores = 1,
                maxIter = 250,
                batchIter = 108)
ticTocTimes <- toc()
```

[1] Actually we train the LRMF model on a small fraction ~0.5% of the training set.

```
## training LRMF on the full training set - classic: 439.578 sec elapsed
```

```
elapsedSeq <- ticTocTimes$toc[[1]] - ticTocTimes$tic[[1]]

# save the RMSE history
rmseHist <- rmseHist %>%
  bind_rows(fitSeq$finalModel$rmseHist %>%
              add_column(method="Classic"))

# plot the convergence for the model on the full training data
colorSpec <- c("salmon", "turquoise3")
names(colorSpec) <- c("Classic", sprintf("Parallel - %d cores", ncores))
rmseHist %>%
  ggplot(aes(iter, rmse, color=method, group=method)) +
  geom_point(aes(shape=method), size=2) +
  geom_line() +
  scale_colour_manual(values = colorSpec) +
  theme(plot.title = element_text(hjust = 0.5), legend.text=element_text(size=12)) +
  xlab("Iterations") + ylab("RMSE") +
  annotate("text", x = 125, colour = colorSpec[2],
           y = rmseHist %>%
             filter(method == sprintf("Parallel - %d cores", ncores)) %>%
             last() %>%
             pull(rmse) - 0.007,
           label = sprintf("%.2f sec", elapsedPar)) +
  annotate("text", x = 250, colour = colorSpec[1],
           y = rmseHist %>%
             filter(method == "Classic") %>%
             last() %>%
             pull(rmse) - 0.004,
           label = sprintf("%.2f sec", elapsedSeq)) +
  theme(legend.position="bottom", plot.title = element_text(hjust = 0.5),
        legend.text=element_text(size = 12))
```



At this point, we're ready to test our trained model in the out-of-sample test set. We reached an RMSE=`0.654579092` and RMSE=`0.635590026` for the parallel and classic respectively. Note that we didn't improve over the simpler baseline model in the out-of-sample but we arrived very close:

```r
## TEST SET ACCESS ALERT! accessing the test set to compute RMSE.
predictedRatings <- predict(fitPar, testSet)
rmseValue <- Metrics::rmse(predictedRatings, testSet$rating)
cat(sprintf("RMSE on test data is %.9f\n", rmseValue))
```

```
## RMSE on test data is 0.654579092
```

```r
# check that we get reproducible results
stopifnot(abs(rmseValue - 0.654579092) < 1e-9)

## TEST SET ACCESS ALERT! accessing the test set to compute RMSE.
predictedRatings <- predict(fitSeq, testSet)
```

```r
rmseValue <- Metrics::rmse(predictedRatings, testSet$rating)
cat(sprintf("RMSE on test data is %.9f\n", rmseValue))
```

```
## RMSE on test data is 0.635590026
```

```r
# check that we get reproducible results
stopifnot(abs(rmseValue - 0.635590026) < 1e-9)
```

Conclusion

Bibliography

Bauer, David F. 1972. "Constructing Confidence Sets Using Rank Statistics." *Journal of the American Statistical Association* 67 (339). Taylor & Francis: 687–90. https://doi.org/10.1080/01621459.1972.10481279.

Davison, Anthony, and D. Hinkley. 1997. "Bootstrap Methods and Their Application." *Journal of the American Statistical Association* 94 (January). https://doi.org/10.2307/1271471.

DiCiccio, Thomas J., and Bradley Efron. 1996. "Bootstrap Confidence Intervals." *Statist. Sci.* 11 (3). The Institute of Mathematical Statistics: 189–228. https://doi.org/10.1214/ss/1032280214.

Koren, Yehuda. 2008. "Factorization Meets the Neighborhood: A Multifaceted Collaborative Filtering Model." In *Proceedings of the 14th Acm Sigkdd International Conference on Knowledge Discovery and Data Mining*, 426–34. KDD '08. New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/1401890.1401944.

Koren, Yehuda, Robert Bell, and Chris Volinsky. 2009. "Matrix Factorization Techniques for Recommender Systems." *Computer* 42 (8). Washington, DC, USA: IEEE Computer Society Press: 30–37. https://doi.org/10.1109/MC.2009.263.

Rialland, Emmanuel. 2019. "HarvardX - Ph125.9x Data Science: Capstone - Movielens." https://github.com/Emmanuel-R8/HarvardX-Movielens/raw/master/MovieLens.pdf.