# Skillability

HarvardX - PH125.9x Data Science Capstone

Giovanni Azua Garcia - giovanni.azua@outlook.com

January 07, 2020

# Contents

# List of Tables

# List of Figures

# Licenses, Terms of Service, Privacy Policy, and Disclaimer



**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 would be 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 interested in discovering what major technology trends exist and their importance: in other words, how those skill clusters have evolved over time.

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, for example:

- 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 Maps Static API and Geocoding API to extract a map of Switzerland and compute the user locations (i.e. longitude and latitude) 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

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

call Google's Geocoding API for every[11] user located in Switzerland. This second step enables answering very practical questions, for example:

- 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[12]. 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 regarding a collegue that was very unsure which graduate program to pursue. Surprisingly he decided to ask the admissions secretary who recommended that he would be better off going for a master in Computational Biology, and so he did. In addition to asking someone without a specialised scientific knowledge, could 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 to fit this model. 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 there we'll outline some interesting findings.

What used to be just an intuitive feeling was 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 could be made more efficient by broadening a search through related skills which don't readily match the job requirements by consulting a Machine Learning model like the one we built in this project.

This last analysis enables us to answer many practical questions too, for example:

[11] Actually all the distinct user locations i.e. about four hundreds
[12] See https://www.tableau.com/

- As a programmer: given my current skill ratings, in what technologies am I predicted to perform above 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` dataset. The script is large and only the most important points will be covered; however, the script is 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 shown in Table 1.1; the folder `data/xml/*` will contain the extracted XML files and finally the folder `data/rds/*` will contain the final `rds` 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.

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(...)`

Table 1.1: Summary of Stack Overflow 7z dataset files

| 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. |

Table 1.2: Summary of the data cleaning process

| 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. |

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 Table 1.2.

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 because although they contribute to voting, we consider the lack of quality questions and answers to be equivalent to `NA`s and they were therefore excluded.

[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.

Table 1.3: Users dataset structure

| 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 |

## 1.3 Data exploration and visualization

### 1.3.1 Description of the dataset

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

```
# 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` dataset depicted in Table 1.3 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:

```
prettyPrint(head(glimpse(users)),
            caption = "Users dataset structure")
```

```
## Observations: 200,630
## 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, ...
```

The `questions` dataset depicted in Table 1.4 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`:

```
prettyPrint(head(glimpse(questions)),
            latex_options = c("striped", "scale_down"),
            caption = "Questions dataset structure")
```

```
## Observations: 5,393,941
## 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...
```

Table 1.4: Questions dataset structure

<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>

Table 1.5: Answers dataset structure

| 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 |

```
## $ 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,...
```

The `answers` dataset depicted in Table 1.5 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"),
            caption = "Answers dataset structure")
```

```
## Observations: 4,830,031
## 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, ...
```

The `badges` dataset depicted in Table 1.6 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"),
            caption = "Badges dataset structure")
```

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

Table 1.6: Badges dataset structure

| 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 |

Table 1.7: Tags dataset structure

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

```
## Observations: 12,597,762
## Variables: 4
## $ userId <int> 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...
```

Finally the tags dataset depicted in Table 1.7 contains the all the unique tags along with their use counts:

```
prettyPrint(head(glimpse(tags)),
            latex_options = c("striped"),
            caption = "Tags dataset structure")
```

```
## 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...
```

### 1.3.2 Quick exploration

Let's explore some interesting facts from the data we have, that is: 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`. The top ten gold badges reveal that being awarded with a "Great Question"[7] is harder than for a "Great Answer"[8], and it's no wonder why, since answers receive in average twice as many up votes as questions:

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

```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 %>%
    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 |

```r
# 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 |

```r
# 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 |

```r
# compare the average scores i.e. up votes for answers vs. questions
prettyPrint(
  questions %>%
    summarise(postType='question',avg_score=mean(score)) %>%
    bind_rows(answers %>%
      summarise(postType='answer',avg_score=mean(score)))
, latex_options = c("striped"))
```

| postType | avg_score |
|---|---|
| question | 6.178113 |
| answer | 12.422926 |

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(median=median(reputation), mean=mean(reputation))
, latex_options = c("striped"))
```

| median | mean |
|---|---|
| 2164 | 5498.885 |

```r
# what's the average number of questions per user?
prettyPrint(
  questions %>%
    group_by(userId) %>%
```

```
    summarise(n = n()) %>%
    ungroup() %>%
    summarise(median=median(n), mean=mean(n))
, latex_options = c("striped"))
```

| median | mean |
|---|---|
| 1 | 5.353187 |

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

| median | mean |
|---|---|
| 8 | 26.2563 |

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

| median | mean |
|---|---|
| 2 | 2.278193 |

In Figure 1.1 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, 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))
```

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

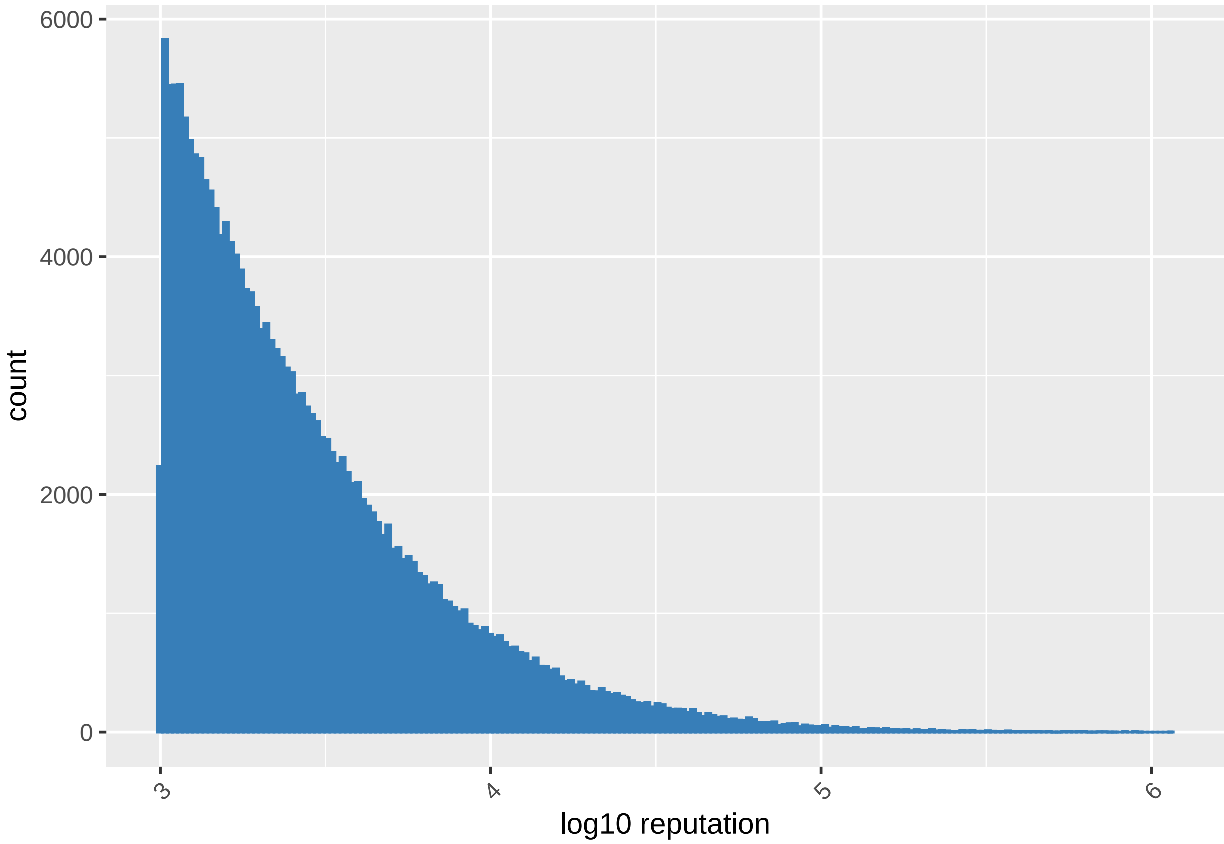


Figure 1.1: Histogram of users reputation

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:

```r
# 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"))) %>%
  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`.
```

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


Figure 1.2: Histogram of users reputation per badge

### 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 2000 skills by frequency of tagging, compute their pair-wise co-occurrence matrix and run PCA on it.

```
# 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?
```

```r
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 490247: 35.469 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)
```

Figure 1.3 depicts the cumulative variability explained up to each principal component. We see that the first four components explain 50% of the variance, and only the first 30 components are required to explain ~95% 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))
qplot(pc, var_explained[pc])
```

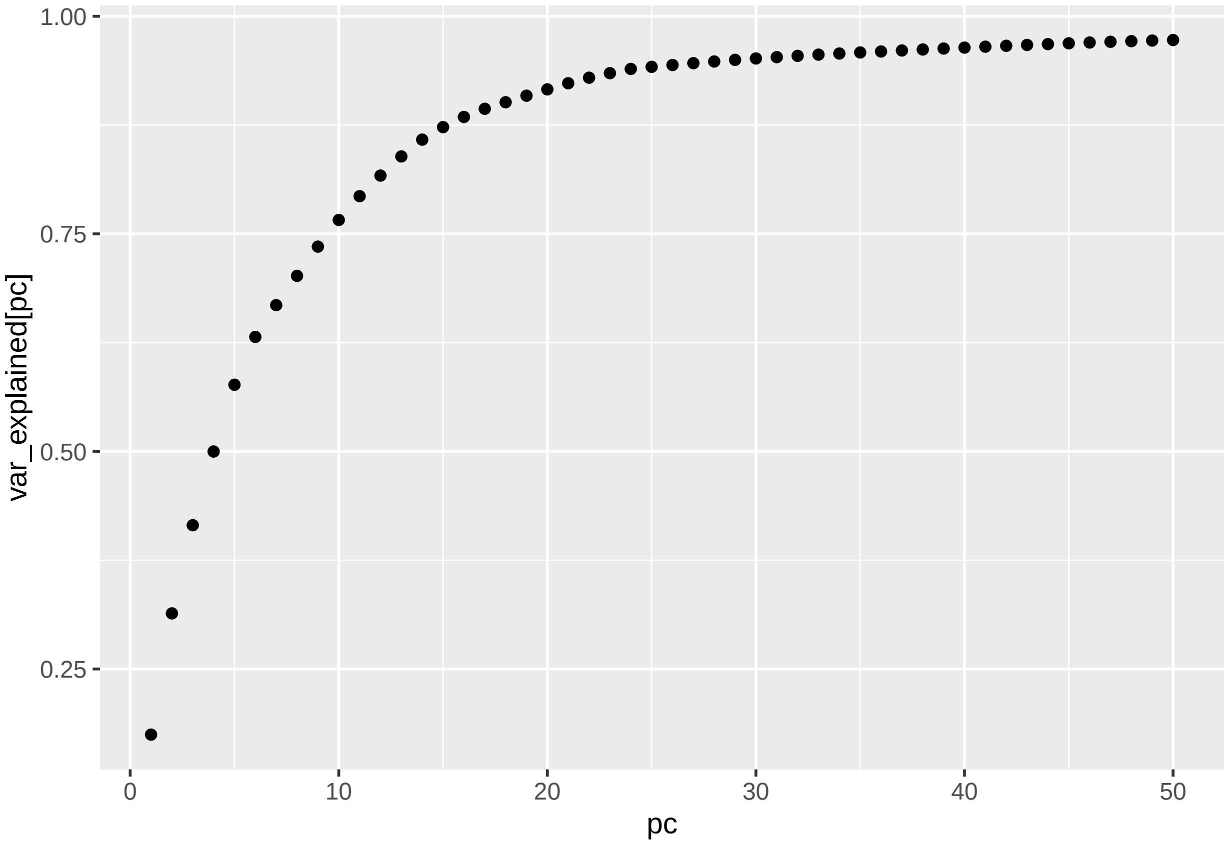


Figure 1.3: Variance explained up to each principal component

Figure 1.4 reveals the skill groups that explain most of the variance in the data or how we prefer to call it, the main technology trends. The top ten 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.

```
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%%",
```

```r
                round(100*pca$sdev[1]^2 / sum(pca$sdev^2)))) +
  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)
```

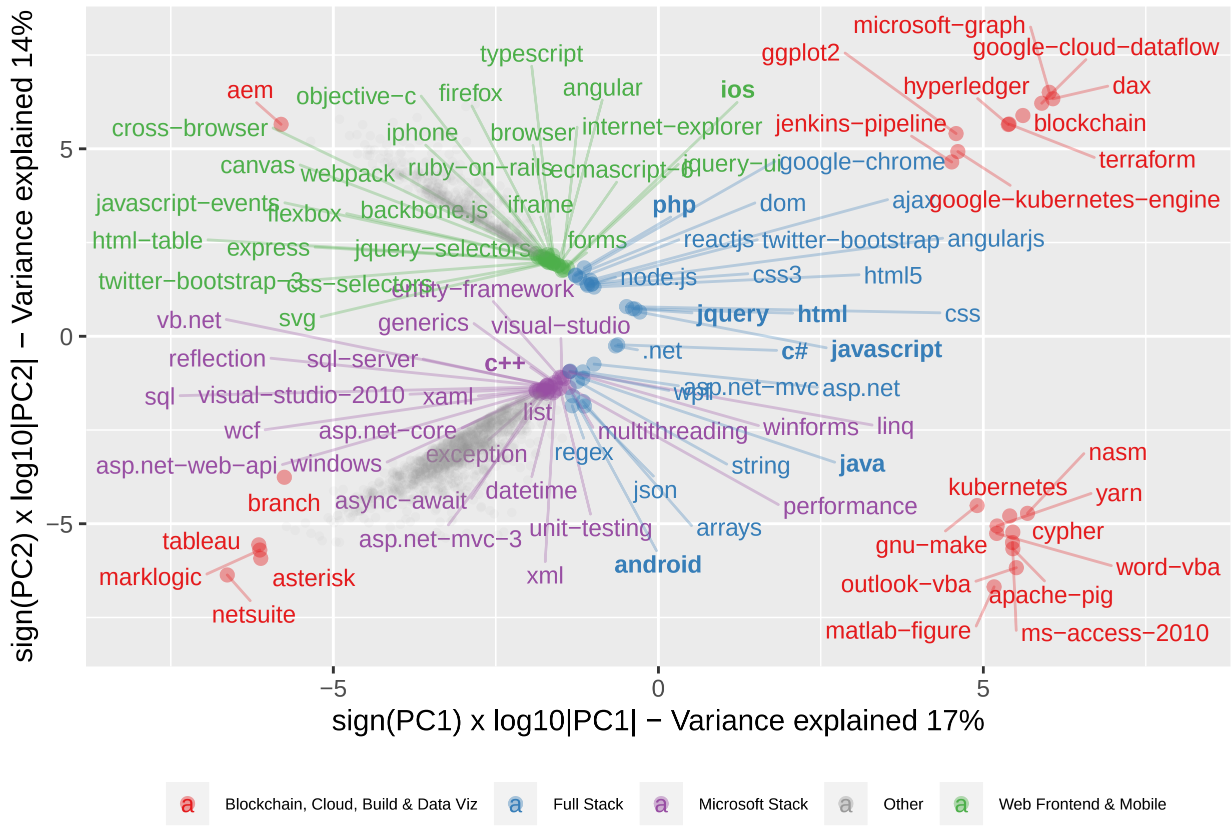


Figure 1.4: Top skills in each direction of the first two Principal Components PC1 and PC2

### 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 located in 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: German `schweiz`, Italian `svizzera`, French `suisse` and Romansh `svizra`:

```r
# do this only if the file isn't there to avoid costly Google geomapping 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"))
```

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

```r

  # get users whose location is Switzerland only
  usersCh <- users %>%
    filter(str_detect(tolower(location),
                      "(\\bch\\b|switzerland|schweiz|svizzera|suisse|svizra)")) %>%
    arrange(desc(reputation))

  # get the unique locations so that we 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")
# expected number of users located in Switzerland
stopifnot(nrow(usersCh) == 4258)
```

Figure 1.5 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 e.g. 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 location as "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")
```

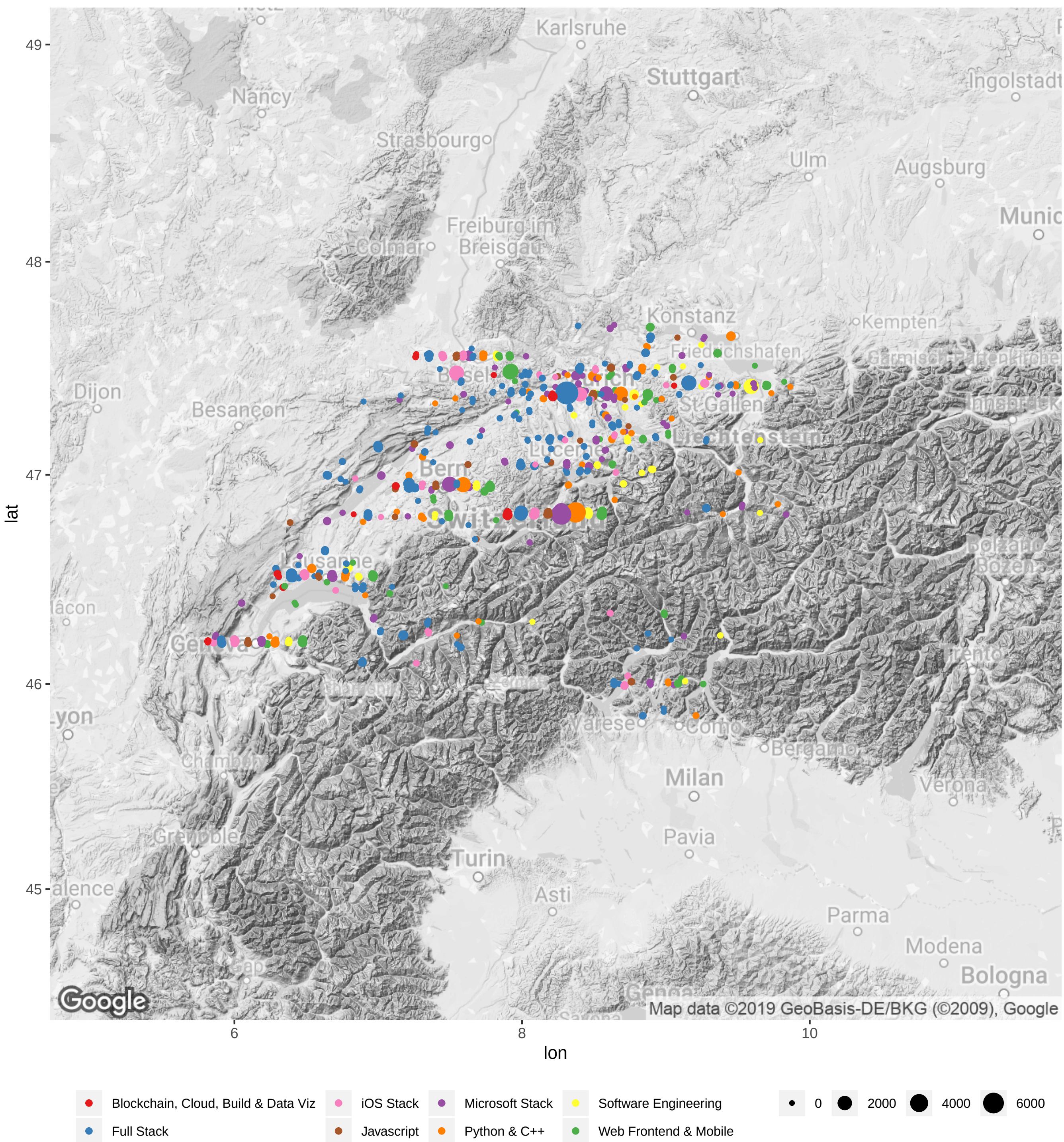


Figure 1.5: Users located in Switzerland and their matching technology trends, weighted by score

The top most prominent post data points in Switzerland are revealed in Table 1.8.

```
prettyPrint(
  usersChTop %>%
    top_n(10, score) %>%
    arrange(desc(score))
, caption = "Top most prominent posts from users located in Switzerland")
```

Table 1.8: Top most prominent posts from users located in Switzerland

| 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 sound 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. 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, will the order suggested by the badges system ensure significant differences among users? This is what we're about to find out in the following statistical inference analysis.

We'd like to validate the hypothesis of whether the user groups would be significantly different 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 average reputation of users awarded with gold answer badges in average significantly better than that of users awarded with gold question badges?

Table 1.9 depicts 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. Recall that we've observed in Figure 1.1 the users reputation to be highly skewed so we choose to work with 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

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

Table 1.9: Average user reputation per class & badge

| 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 | 3146.0 |
| silver | Good Question | 3031.0 |
| bronze | Nice Answer | 2580.0 |
| bronze | Nice Question | 2544.0 |

note that users granted gold badges depict higher average reputation than that of users granted silver badges, similarly users granted silver badges tend to have higher reputations in average than users granted bronze badges.

```
# 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"), caption = "Average user reputation per class \\& badge")
```

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 to work (e.g. for plotting) we're going to do conduct the following inference analysis using a `log10` transformation of the users reputation.

Figure 1.6 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:

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

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

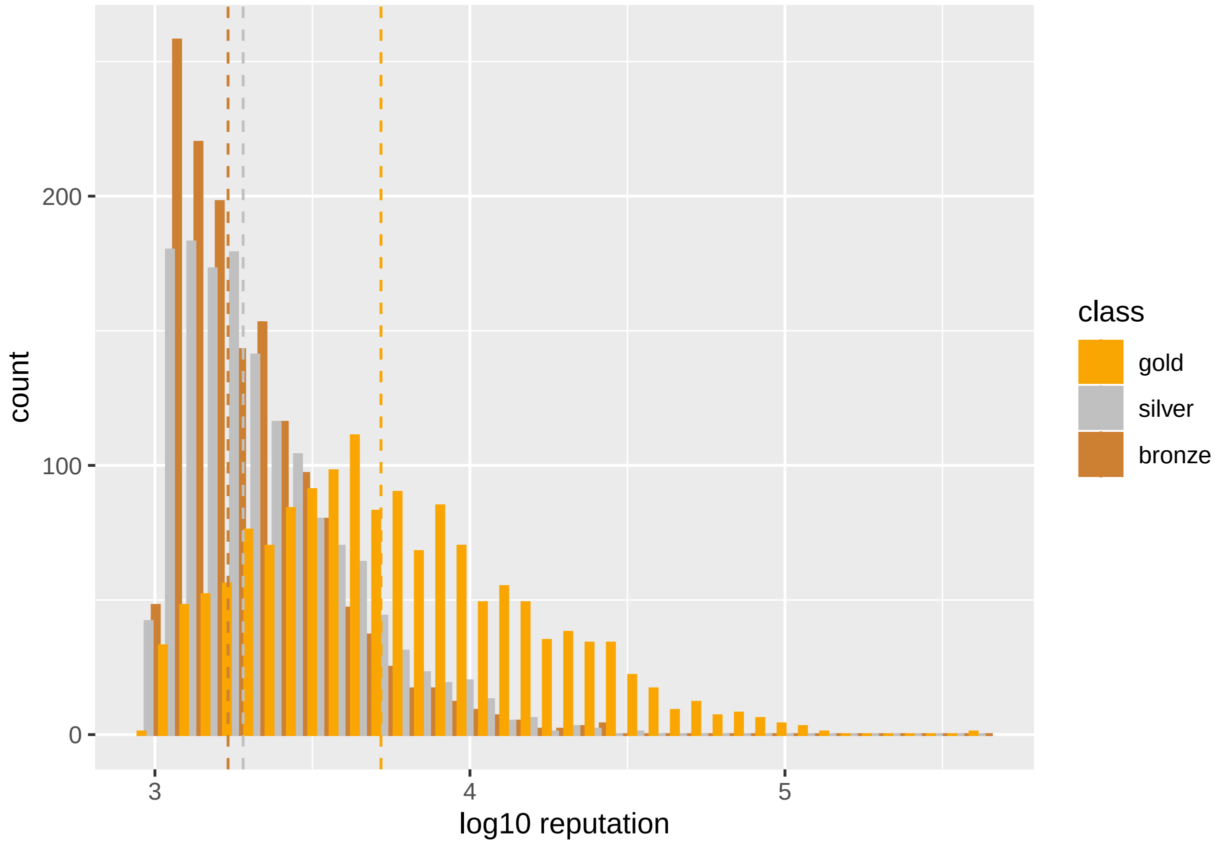


Figure 1.6: Histograms of users reputation per Answer badges

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 the difference in average reputation between the groups created using this exclusion process. To that end, we run the exclusion process starting with the average order suggested by the listing above. We then randomly sample N users per group, for instance, `N=1500` and compare their medians. Our final ordering is the one that depicts the reputation average and distribution of the random samples per group monotonically decreasing. We'd also like to use statistical inference to find whether there is significant difference in the user reputation population medians between these groups. However, we've observed before the in-group distributions' strong departure from normality in Figures 1.1 and 1.2. Therefore, we construct 95% confidence limits using the non-parametric bias-corrected and accelerated bootstrap interval $BC_a$, a statistically robust algorithm for producing highly accurate confidence limits from a boostrap distribution see (DiCiccio and Efron 1996) and (Davison and Hinkley 1997). Although our samples are chosen randomly and we could resort to the Central Limit

Theorem (CLT)[13] we believe that the non-parametric $BC_a$ approach is more adequate and precise since no assumptions are required regarding the underlying distribution of the data.

The following listing compares the median of the reputation for the independent groups 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 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 some questions:

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

[13]See the https://en.wikipedia.org/wiki/Central_limit_theorem
[14]See the question https://stackoverflow.com/questions/59504775/

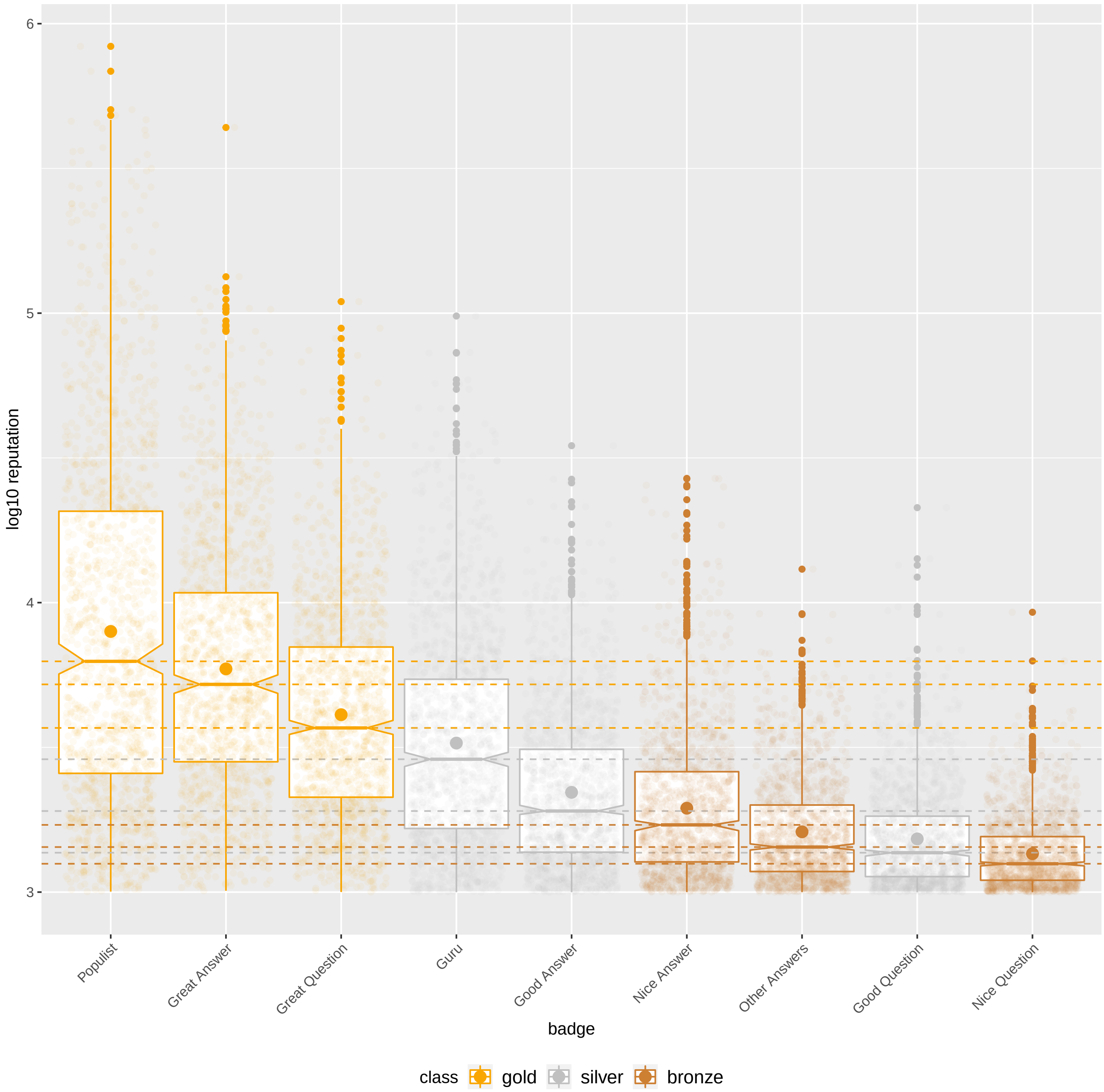


Figure 1.7: Reputation for random samples of 1500 users in each badge & class combination

No other ordering of badges and class levels produces the monotonically decreasing median and mean reputation distribution for the users depicted in Figure 1.7. However, 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[15] (see Bauer 1972) for independent samples to determine the significance of the difference in population median between unpaired groups. The Wilcoxon test is non-parametric, and thus doesn't require the normality assumption of the data which is exactly our scenario. The following listing executes the Wilcoxon test on all badges pair-wise. The listing produces no output which confirms what we observed in Figure 1.7, namely, that the averages between these groups are pair-wise significatively different:

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

Table 1.10: Rating assignments per badge and class

| Badge | Class | Rating | Conditions |
|---|---|---|---|
| Populist | gold | 5.0 | |
| Great Answer | gold | 5.0 | |
| Great Question | gold | 4.5 | |
| 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 | | 3.0 | Answers with score between [0, 5) |
| 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 |

```r
# generate all possible pair-wise badges combinations
c <- combn(badgesOrder, m=2)
# run Wilcoxon rank sum test for independent samples i.e. significance of
# the difference in population median between non-normally distributed
# unpaired groups. Print the pairs whose p-value is greater than 0.05 i.e.
# the mean difference between groups is not significative at
# 95% confidence
for (i in 1:ncol(c)) {
  res <- comp %>%
    filter(badge == c[1,i] | badge == c[2,i]) %>%
    wilcox.test(reputation~badge, data=., paired=F, conf.int=T)
  if (res$p.value > 0.05) {
    cat(sprintf(paste0("The median rep. for users with badge ",
                       "'%s' and '%s' is NOT significatively ",
                       "diff. with p-value=%.6f\n"),
                c[1,i], c[2,i], res$p.value))
  }
}
```

The final ordering is summarized in the following Table 1.10[16]. 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:

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[17] 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 our 32GB system RAM. Therefore we again leveraged on

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

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

Table 1.11: Ratings dataset structure

| 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 |

Table 1.12: Ratings dataset measures of central tendency and dispersion

| median | mean | sd |
|---|---|---|
| 3 | 3.266489 | 0.793884 |

the multi-core parallel architecture[18] 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 as depicted in Table 1.11 it includes the following main columns: `userId`, `skill` and `rating`; other columns were included for debugging, traceability[19] and to build model features e.g. `firstPostDate` which will be discussed later.

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

prettyPrint(head(glimpse(ratings)), caption = "Ratings dataset structure")
```

```
## Observations: 5,442,999
## Variables: 7
## $ userId        <int> 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...
```

We can see in Table 1.12 the measures of central tendency and dispersion 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"), caption = "Ratings dataset measures of central tendency and dispersi
```

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:

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

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

```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") +
  theme(legend.position="bottom", legend.title = element_blank())
```

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

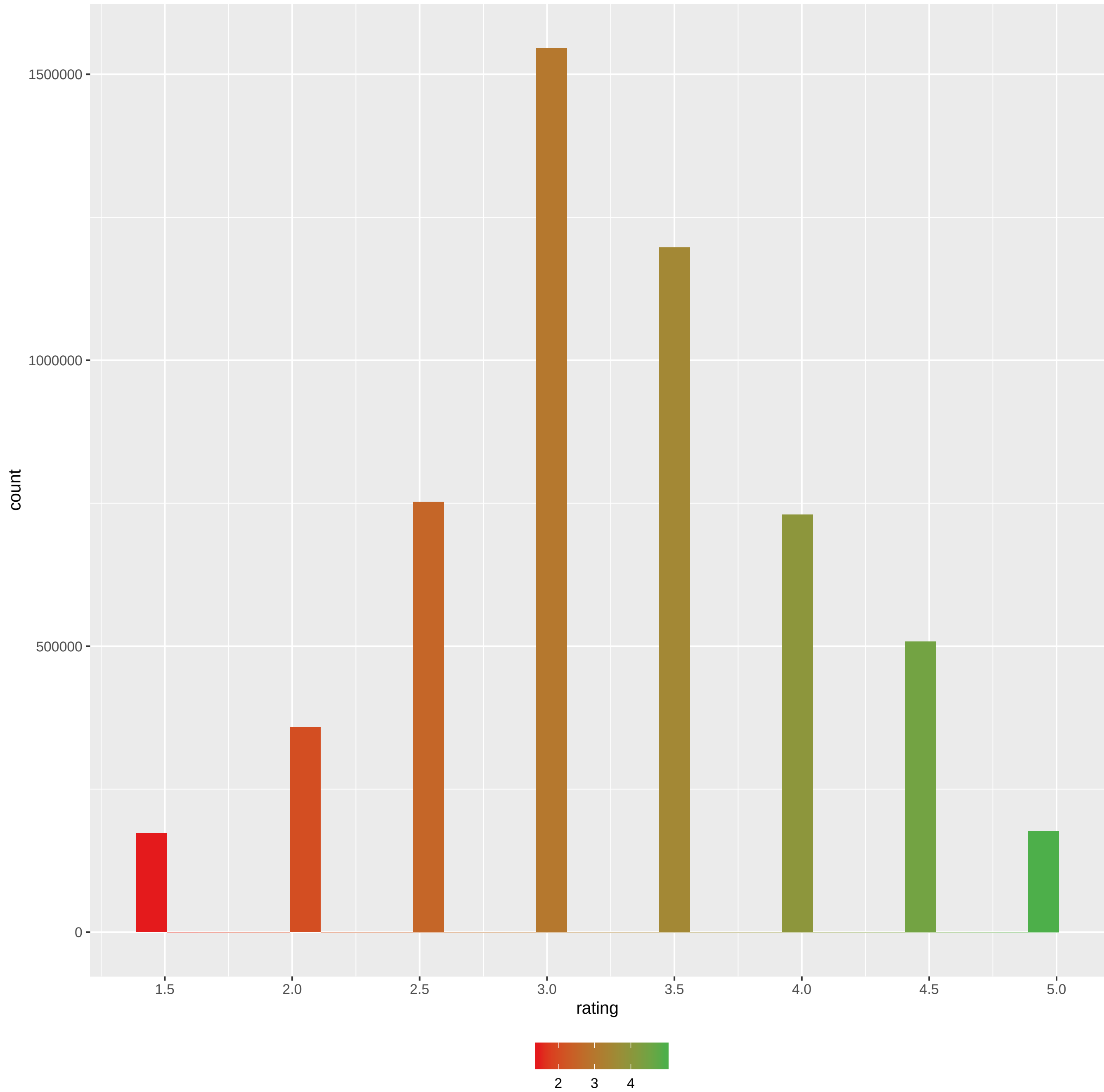


Figure 1.8: Histogram of user skill ratings dataset

At this point we're ready to assess whether the user reputation average differences are significant per rating group. We take N random samples from each rating group and again use the $BC_a$ method to construct 95% confidence limits. Figure 1.9 shows that several rating groups do not seem to be significatively different; namely, the $BC_a$ confidence limit notches clearly overlap for ratings 3, 3.5, 4, and 4.5:

```r
# 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)
```

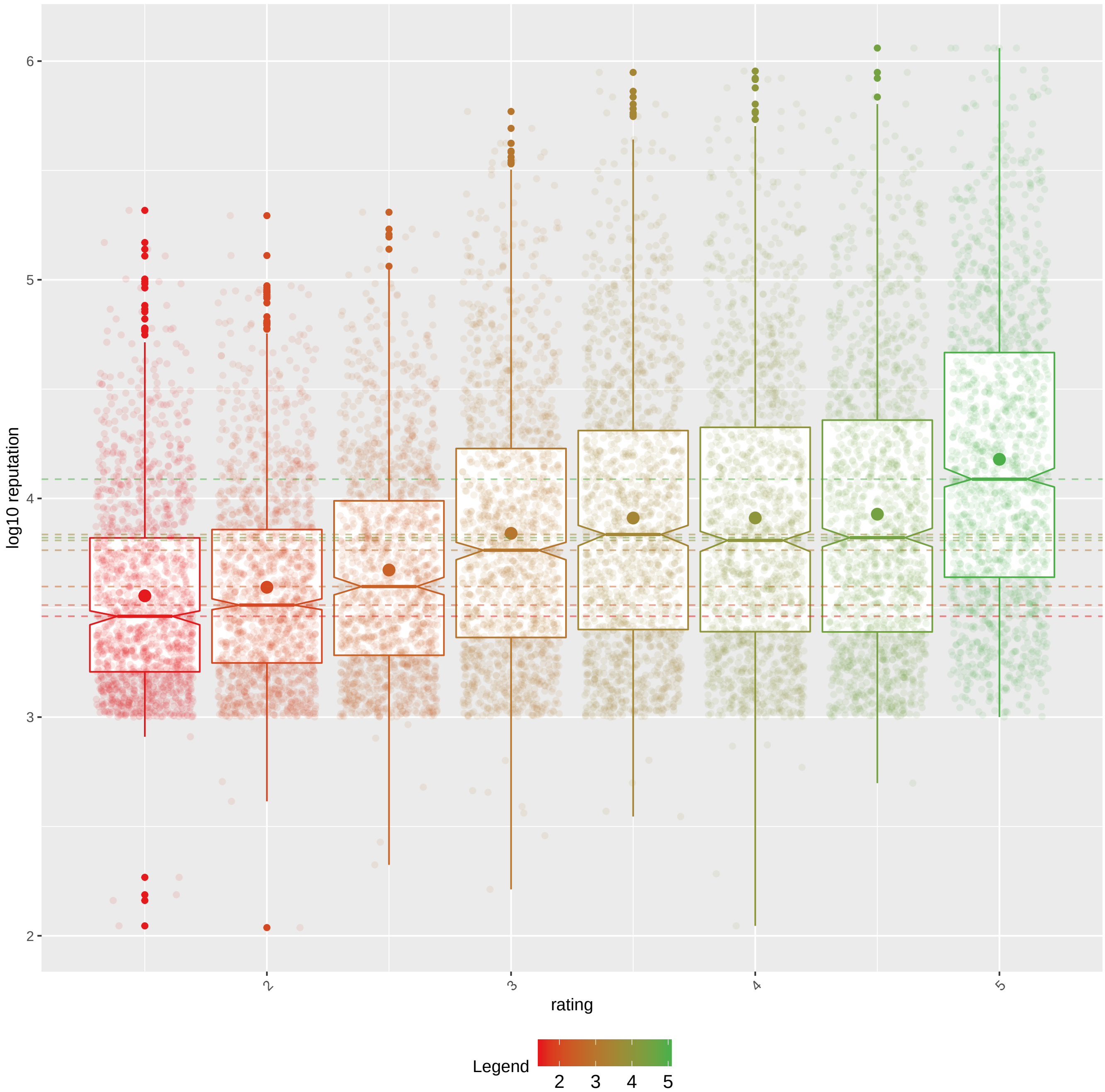


Figure 1.9: Boxplot of the users reputation per class & badge

We again use the Wilcoxon test to compare the average of the different groups. The results reveal that the reputation median differences for the rating group pairs `3.5-4.0`, `3.5-4.5` and `4.0-4.5` aren't significant at 95% confidence level, whereas all other rating pairs are:

```
# 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
# 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) {
```

```r
    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=0.69186
## Median rep. for users with rating 3.5 and 4.5 is NOT sig. diff. with p-value=0.71219
## Median rep. for users with rating 4.0 and 4.5 is NOT sig. diff. with p-value=0.47357
```

# 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 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 $\hat{r_{i,j}}$ is our predicted user skill rating for user $i$ and skill $j$, we'll reuse the `Metrics::rmse(...)` implementation:

$$\text{RMSE} = \sqrt{\frac{1}{N}\sum_{i,j}(\hat{r}_{i,j} - r_{i,j})^2}$$

Let's get some basic information about the `ratings` dataset e.g. sparsity. Table 2.1 shows there are 199.8k users and 2000 skills[1] and a highly sparse dataset i.e. only about 1.4% of all the possible user skill ratings:

```r
# 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"),
caption = "Number of unique users, skills and the sparsity level")
```

We start by separating the `ratings` dataset into training 90% and test 10% 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:

```r
# split the ratings dataset into separate train and test sets
portable.set.seed(1)
```

[1] These are the top 2000 skills we have used before that account for 82.3% of the total taggings.

Table 2.1: Number of unique users, skills and the sparsity level

| users | skills | sparsity |
|---|---|---|
| 199853 | 2000 | 1.4% |

```r
testIndex <- createDataPartition(y = ratings$rating, times = 1,
                                 p = 0.1, list = FALSE)
trainSet <- ratings[-testIndex,]
tmp <- ratings[testIndex,]

# 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] 4898943
```

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

```
## [1] 544056
```

## 2.1 Baseline

To get a benchmark and idea for what level of RMSE we can reach, we start by exploring the simpler but effective baseline model. In this model we'll account for the ratings' gobal mean $\mu$, user effects $b_i$, and skill effects $b_j$. We'll also employ regularization $\lambda$ (i.e. penalized least squares) which permit us to penalize large absolute value estimates that are formed from small sample sizes e.g. users with very few skill ratings. If these estimates were left untreated with regularization, they would increase uncertainty, and thus contribute to larger errors negatively impacting our RMSE. Our baseline model is then specified using the following cost function:

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

We start building the baseline model with just the average. Note that we're 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 | 0.7939102 |

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 | 0.7939102 |
| Regularized User Effects | 0.6636790 |

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 $\lambda$ were manually tried and $\lambda = 4$ was the best.

| method | RMSE |
|---|---|
| Just the average | 0.7939102 |
| Regularized User Effects | 0.6636790 |
| Regularized User + Skill Effects | 0.6489351 |

We find ourselves now in a place to account for more interesting effects. Let's consider adding an user-specific feature 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 experience effect $b_e$ on a skill. We included in the `ratings` dataset the column `firstPostDate` corresponding to the timestamp 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. 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]. Our initial baseline model is then extended as follows:

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

Figure 2.1 depicts a `loess` smoothing of the effect $b_e$ after discounting all others 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 Figure 2.1 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:

```r
# 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
```

```r
# 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) %>%
```

[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

```
  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'
```

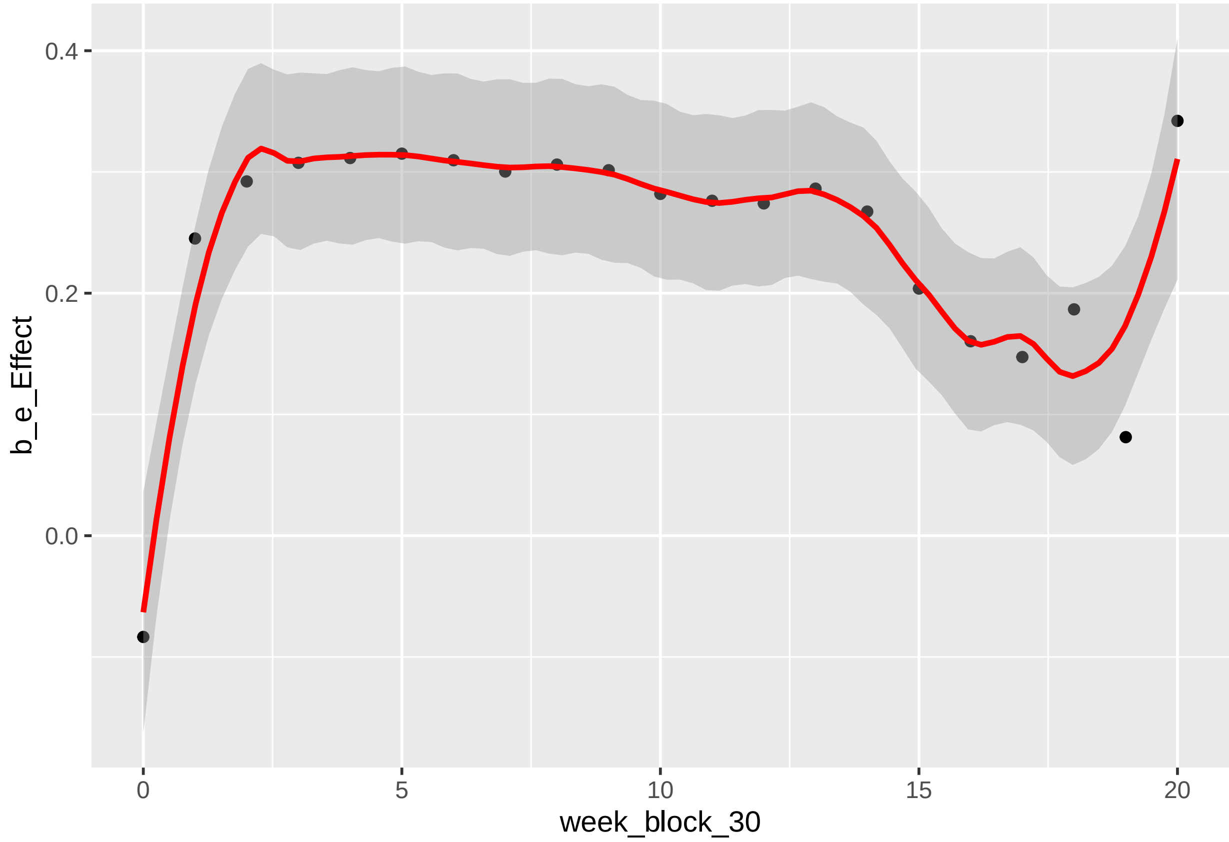


Figure 2.1: Smoothing function of the temporal effect gained experience

We then add the $b_e$ effect to our model as shown in Table 2.2:

```
# 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,
```

Table 2.2: Baseline simpler CF model in-sample RMSE

| method | RMSE |
|---|---|
| Just the average | 0.7939102 |
| Regularized User Effects | 0.6636790 |
| Regularized User + Skill Effects | 0.6489351 |
| Regularized User + Skill + Experience Effects | 0.6319561 |

```
                          tibble(method="Regularized User + Skill + Experience Effects",
                                 RMSE = Metrics::rmse(predictedRatings,
                                                      trainSet$rating)))
prettyPrint(
  rmseResults
, latex_options = c("striped"),
caption = "Baseline simpler CF model in-sample RMSE")
```

We reached a RMSE=`0.6319561` 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 0.653410415
```

```
# check that we get reproducible results
stopifnot(abs(rmseValue - 0.653410415) < 1e-9)
```

We reached an out-of-sample RMSE=`0.653410415` which is reasonable and ties in with the lack of significance between some ratings groups previously discussed i.e. we can't expect a much lower RMSE when the population average reputation differences between rating groups e.g. `3.5-4.0` aren't significative.

Note that we haven't calibrated this baseline model properly, we've simply manually experimented with the hyper-parameters: $\lambda$, `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`.

We could keep on exploring, adding features and accounting for more interesting effects that would lower our RMSE even further[6]; adding many features increases model complexity and

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

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 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 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-intrinsic dimension representation we obtain using LRMF can also be achieved by computing the singular value decomposition SVD[8] see (Golub and Loan 2013) 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 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 when possible 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:

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

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

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

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

# 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.
- $\lambda$: the regularizaton parameter.
- $\sigma$: 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 $\mu$ and $\sigma$.
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 $\gamma$ (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 and compute the gradient updates corresponding to those blocks 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 one core will work on users and skills whose `i %% ncores == 0 & j %% ncores == 0` respectively, another 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 any 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 on the matrix columns as discussed before. Lines `#20` through `#80` correspond to the parallel

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

implementation. There we pick a random block group and run `parIter` iterations on it. In lines #54 through #68 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 #79 “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
53            # 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: 0.016 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 873 unique skills and 1972 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 = 0.03, sigma = 0.05 on full training set
```

```
toc()
```

```
## calibrating the LRMF model: 63.834 sec elapsed
```

```
## the bestTune model found is:
stopifnot(cvFit$bestTune$K == 10)
stopifnot(cvFit$bestTune$gamma == 0.1)
stopifnot(cvFit$bestTune$lambda == 0.03)
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 27000 (i.e. `maxIter*batchIter`) to compare them. The parallel is trained with half the `maxIter` compensated with doubling the `batchIter`.

```
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: 218.388 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()
```

```
## training LRMF on the full training set - classic: 482.451 sec elapsed
```

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

```

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

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

Figure 4.1 reveals that the parallel implementation arrives to a comparable RMSE in about half the time of the classic. However, by halving the number of iterations, the parallel version has half the opportunities to correct the gradient direction. This we can see in the RMSE convergence for the parallel case being more bumpy compared to the classic:

```r
# 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) + 0.004,
           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))
```

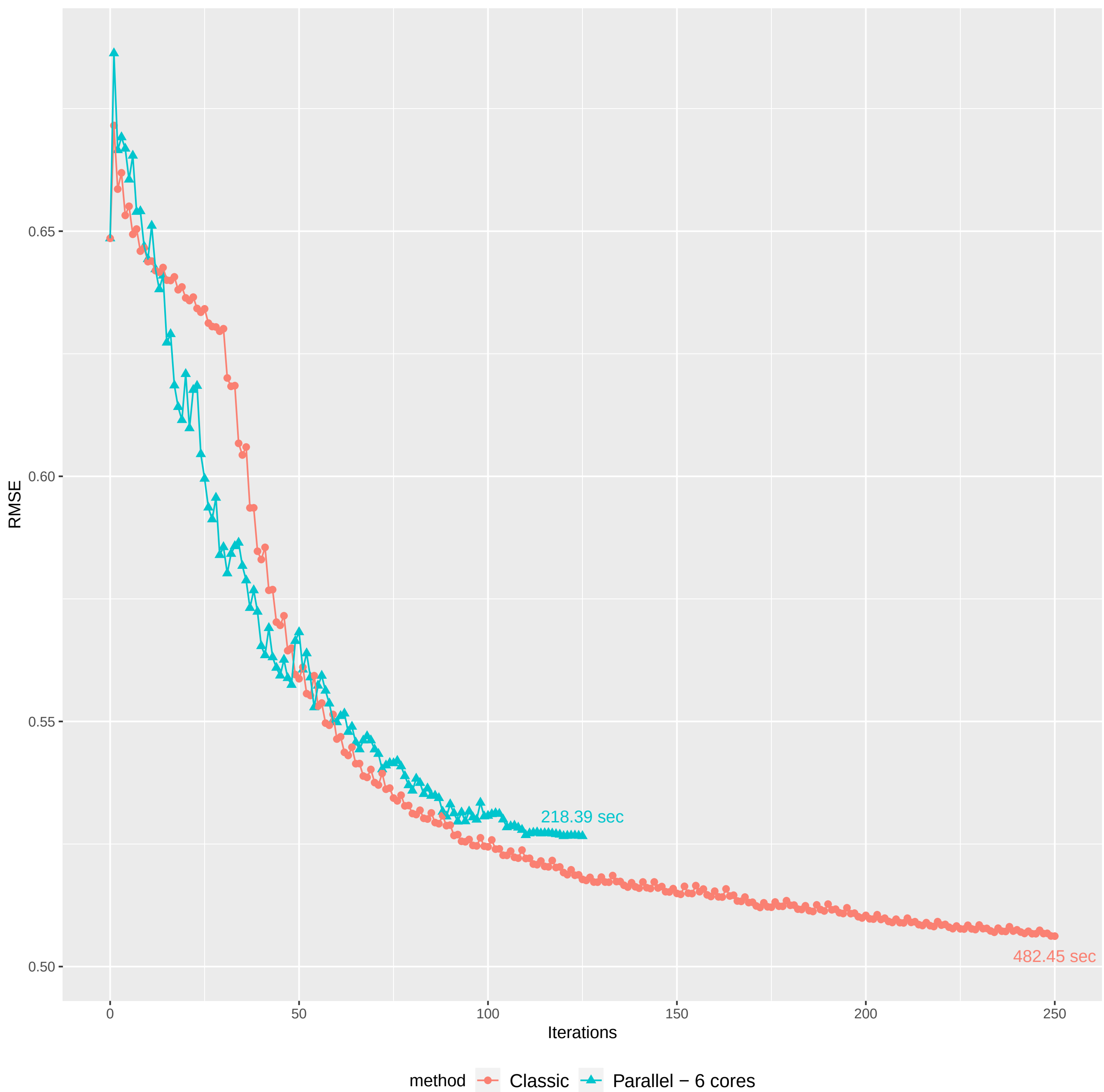


Figure 4.1: Convergence comparison between classic and parallel implementations

At this point, we're ready to test our trained model in the out-of-sample test set. In the out-of-sample we've reached a RMSE=`0.684472449` and RMSE=`0.667113795` 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 close:

```
## 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 0.686592201
```

```
# check that we get reproducible results
stopifnot(abs(rmseValue - 0.686592201) < 1e-9)
```

Table 4.1: Skills where the author is top rated according to his Stack Overflow activity

| userId | skill | firstPostDate | creationDate | postId | postType | rating |
|---|---|---|---|---|---|---|
| 1142881 | collections | 2012-12-10 08:57:14 | 2012-12-10 08:57:14 | 13797704 | answer | 4.5 |
| 1142881 | hashmap | 2012-12-10 08:57:14 | 2012-12-10 08:57:14 | 13797704 | answer | 4.5 |
| 1142881 | hashtable | 2012-12-10 08:57:14 | 2012-12-10 08:57:14 | 13797704 | answer | 4.5 |
| 1142881 | java | 2012-01-11 09:54:53 | 2012-12-10 08:57:14 | 13797704 | answer | 4.5 |
| 1142881 | cpu | 2012-08-19 16:04:40 | 2013-04-27 08:44:05 | 16250081 | answer | 4.0 |
| 1142881 | git | 2012-12-02 18:45:42 | 2014-10-07 13:57:42 | 26237795 | answer | 4.0 |
| 1142881 | maven | 2012-12-22 09:09:31 | 2014-10-07 13:57:42 | 26237795 | answer | 4.0 |
| 1142881 | boost | 2012-07-20 14:20:18 | 2012-07-20 15:03:14 | 11581933 | answer | 3.5 |
| 1142881 | c# | 2012-12-10 16:33:06 | 2012-12-10 16:33:06 | 13805069 | answer | 3.5 |
| 1142881 | c++ | 2012-04-05 19:31:10 | 2012-07-20 15:03:14 | 11581933 | answer | 3.5 |
| 1142881 | code-generation | 2015-09-24 08:20:53 | 2015-09-28 11:26:32 | 32821844 | answer | 3.5 |
| 1142881 | crash | 2019-06-05 09:23:35 | 2019-06-05 09:29:48 | 56457806 | answer | 3.5 |
| 1142881 | design-patterns | 2012-01-11 09:54:53 | 2012-12-10 16:33:06 | 13805069 | answer | 3.5 |
| 1142881 | domain-driven-design | 2012-12-10 16:33:06 | 2012-12-10 16:33:06 | 13805069 | answer | 3.5 |
| 1142881 | eclipselink | 2012-12-22 09:09:31 | 2012-12-22 10:03:13 | 14001887 | answer | 3.5 |
| 1142881 | generics | 2012-12-07 09:14:42 | 2012-12-07 09:14:42 | 13760045 | answer | 3.5 |
| 1142881 | image-processing | 2012-08-11 10:54:24 | 2012-08-11 10:54:24 | 11914054 | answer | 3.5 |
| 1142881 | linux | 2013-07-09 19:54:42 | 2019-06-05 09:29:48 | 56457806 | answer | 3.5 |
| 1142881 | sbt | 2015-09-24 08:20:53 | 2015-09-28 11:26:32 | 32821844 | answer | 3.5 |
| 1142881 | scala | 2013-07-01 09:51:13 | 2015-09-28 11:26:32 | 32821844 | answer | 3.5 |
| 1142881 | ubuntu | 2012-08-26 10:49:38 | 2019-06-05 09:29:48 | 56457806 | answer | 3.5 |
| 1142881 | vectorization | 2012-08-11 10:54:24 | 2012-08-11 10:54:24 | 11914054 | answer | 3.5 |

```

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

```
## RMSE on test data is 0.667113795
```

```
# check that we get reproducible results
stopifnot(abs(rmseValue - 0.667113795) < 1e-9)
```

Now we're ready to test drive our model on the author's data. A question posed in the introduction was to predict the author[2] ratings in skills for which there is no previous evidence. For example, how high would the author be rated on skill `tableau`? and whether it was a sound decision to reject the author as candidate applying for a job that had `tableau` as requirement simply because the candidate had not used that specific skill before? In order to do that let's first take a look at Table @ref(tag:results-top-skills) and notice the skills where the author is top rated:

```
# where is the author top rated?
prettyPrint(
  ratings %>%
    filter(userId == 1142881) %>%
    top_n(20, rating) %>%
    arrange(desc(rating), skill)
, caption = "Skills where the author is top rated according to his Stack Overflow activity")
```

Let's find the skills for which there is no previous evidence, that's it, those skills for which the author doesn't have any observed ratings. Then compute the average ratings for all users that have been rated on those skills and compute the predicted author's rating on those skills. We

[2] See https://stackoverflow.com/users/1142881

then output the author's predicted ratings for some interesting technologies in Table 4.2 and notice that he's predicted to be way above average for skill `tableau` confirming his initial hunch, this time using a machine learning model. Finally in Table 4.3 we output the top 20 skills where the author is predicted to rate above average and in descending predicted rating order. It could be worthwhile following the recommendation of our model and learn those skills:

```
# find the skills for which there are no ratings i.e. there is no evidence
noEvidenceSkills <- mainSkills %>%
  anti_join(ratings %>%
  filter(userId == 1142881) %>%
  select(skill) %>%
  unique(), by="skill") %>%
  arrange(desc(count))

# compute the ratings average for each of those skills
avgNoEvidenceSkills <- ratings %>%
  group_by(skill) %>%
  summarise(avg=mean(rating)) %>%
  semi_join(noEvidenceSkills, by="skill")

# create new data for prediction
newdata <- noEvidenceSkills %>%
  select(skill, count) %>%
  mutate(userId=1142881) %>%
  inner_join(avgNoEvidenceSkills, by="skill") %>%
  select(userId, skill, count, avg)

# compute skill rating predictions
newdata$predicted <- predict(fitSeq, newdata)

# show how would be rated for the following interesting technologies
prettyPrint(
  newdata %>%
    filter(skill %in% c("blockchain", "haskell", "klotin", "apache-kafka",
                        "tableau", "spring-boot", "google-maps", "c++11",
                        "c++17", "spring-mvc", "ejb", "game-physics",
                        "go", "java-stream", "teradata", "itext")) %>%
    arrange(desc(predicted))
, latex_options = c("striped"),
caption = "Author rating predictions for some interesting technologies")
```

```
# show the top 20 skills where the predicted rating is above average
prettyPrint(
  newdata %>%
    filter(predicted > avg) %>%
    top_n(20, predicted) %>%
    arrange(desc(predicted))
, latex_options = c("striped"),
caption = "Skill predictions where the author is rated above average")
```

Table 4.2: Author rating predictions for some interesting technologies

| userId | skill | count | avg | predicted |
|---|---|---|---|---|
| 1142881 | java-stream | 7222 | 3.369593 | 3.694358 |
| 1142881 | c++11 | 49481 | 3.383537 | 3.645484 |
| 1142881 | c++17 | 4729 | 3.379375 | 3.634192 |
| 1142881 | teradata | 3957 | 2.898148 | 3.301042 |
| 1142881 | game-physics | 2960 | 3.135027 | 3.143230 |
| 1142881 | itext | 8248 | 2.894942 | 3.113528 |
| 1142881 | ejb | 6420 | 3.001654 | 3.028257 |
| 1142881 | tableau | 3810 | 2.623077 | 3.025541 |
| 1142881 | spring-mvc | 52463 | 3.220928 | 3.024308 |
| 1142881 | go | 42071 | 3.372963 | 2.915764 |
| 1142881 | blockchain | 3081 | 3.247664 | 2.887793 |
| 1142881 | spring-boot | 70632 | 3.254878 | 2.803498 |
| 1142881 | google-maps | 61589 | 3.146213 | 2.701235 |
| 1142881 | haskell | 42535 | 3.437148 | 2.677507 |
| 1142881 | apache-kafka | 16288 | 3.172965 | 2.367614 |

Table 4.3: Skill predictions where the author is rated above average

| userId | skill | count | avg | predicted |
|---|---|---|---|---|
| 1142881 | angular2-routing | 4592 | 3.299296 | 4.078456 |
| 1142881 | pytest | 4099 | 3.347894 | 4.041698 |
| 1142881 | angularjs-scope | 8897 | 3.211193 | 4.029082 |
| 1142881 | elisp | 3682 | 3.440758 | 3.980609 |
| 1142881 | pom.xml | 3813 | 3.176471 | 3.977188 |
| 1142881 | d3.js | 34425 | 3.122737 | 3.932487 |
| 1142881 | glsl | 6203 | 3.116667 | 3.926849 |
| 1142881 | capistrano | 3734 | 3.292670 | 3.912535 |
| 1142881 | junit4 | 3435 | 3.273992 | 3.901931 |
| 1142881 | react-router | 11083 | 3.276860 | 3.901841 |
| 1142881 | ruby-on-rails | 312231 | 3.599279 | 3.878598 |
| 1142881 | maven-2 | 5568 | 3.326495 | 3.870781 |
| 1142881 | rake | 4220 | 3.375678 | 3.864666 |
| 1142881 | angularjs-directive | 17324 | 3.189521 | 3.860883 |
| 1142881 | language-lawyer | 4999 | 3.420351 | 3.859373 |
| 1142881 | observable | 6712 | 3.202961 | 3.858894 |
| 1142881 | kotlin | 31402 | 3.410618 | 3.845595 |
| 1142881 | actionscript | 9138 | 3.032289 | 3.841658 |
| 1142881 | conda | 3253 | 3.298755 | 3.818662 |
| 1142881 | eclipse | 118255 | 3.289904 | 3.818293 |

# Conclusion

In this work we have unveiled multiple data science analysis use-cases possible by studying the Stack Overflow data. There are many more interesting questions we could answer using this dataset, for example:

- how does the question sentiment affect the reputation of the users?
- how does the sentiment also affect the score of the posts?
- how is a post predicted to score?

However, in this study we have focussed on topics of more practical relevance and in the context of the recruitment industry i.e. clustering skills, identifying technology trends and putting them in geographical context. Furthermore, we have derived and designed a new user skill ratings dataset and presented a recommender system implementation to predict user skill ratings that offers many interesting uses in practice.

First we've built a fully automated R script to: download, extract, parse, clean and store the Stack Overflow data. It was already a challenge to start with since some of the extracted XML files expand to 75GB in size. We applied blocking and parallel processing in this and other areas of this work.

We successfully applied dimensionality reduction in two different areas, first we applied PCA to the co-occurrence matrix of skills in questions and in order to identify the main technology trends occurring in the last ten years which was show in Figure 1.4. As discussed, running this analysis in a rolling time window fashion we would discover the changes in technology trends over different periods of time. As result of this analysis we learned from the first principal component that the technologies explaining most of the variance in the data are related to blockchain, cloud computing, build, deployment tools and data visualization. Although not totally surprising it's interesting to learn that blockchain seems to be applied in many different contexts. Another take away result is that cloud computing is ubiquitous. It's definitely a strong asset for any technology professional to master.

Placing the main technology trends in geographical context as shown in Figure 1.5 for the Stack Overflow users located in Switzerland reveals that Zurich has become a technology center. We identified all the main trends there, and a high quantity and quality of Stack Overflow users posting from this location. Indeed most of the big technology players have been expanding to Zurich including: Google, Microsoft, Facebook, Oracle, and others. We noticed too that the south-east parts of Switzerland e.g. Tessin don't seem to be too active technology-wise at least this isn't obvious by looking at the number of Stack Overflow users posting those locations. We expected to see above average blockchain activity in Zug believed to be a cryptocurrency haven but this doesn't seem to be the case again judging by the top posts from users located in that area.

Finally, we shifted our focus to building a new dataset derived from Stack Overflow, the `ratings` dataset and we surprised ourselves facing an interesting statistical inference challenge. Namely to assess the significance of the user reputation average differences before and after building the `ratings` dataset. Before building the `ratings` dataset we assessed significance for the difference in user reputation averages between the class and badge groups, see Figure 1.7. After building the `ratings` dataset we assessed significance for the difference in user reputation averages between the rating groups, see Figure 1.9. Due to the users reputation strong departure from normality (see Figures 1.1 and 1.2) we tested significance of the difference in population medians and using the non-parametric $BC_a$ method see (DiCiccio and Efron 1996) and (Davison and Hinkley 1997) for constructing 95% boostrap confidence limits. We verified the overlapping visual inspection of the confidence interval results using the Wilcoxon rank sum test[3] (see Bauer 1972) for all groups pair-wise. Three ratings groups were not found significatively different pair-wise and we gauged the impact in the user skill rating predictions. The result of this challenge led to our user skill rating assignment strategy described in Table 1.10. Building the `ratings` dataset was also a performance challenge which we attacked, once more, using blocking and parallel processing. Last but not least we built a new recommender system using the collaborative filtering technique and featuring two SGD algorithm flavours: classic and parallel. We used our model for predicting user skill ratings which delivered interesting and useful results.

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

# Future Work

It was left out of scope to futher increase the search space used for the calibration of the LRMF model. We'd need better computer recources to achieve a more exhaustive search which would definitely lead to higher quality results.

We also found interesting in the parallel SGD implementation researching the possibility to have each core mildly tune the gradient descent direction, and thus have them each train longer independently on its own block.

# 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.

Golub, Gene H., and Charles F. van Loan. 2013. *Matrix Computations*. Fourth. JHU Press. http://www.cs.cornell.edu/cv/GVL4/golubandvanloan.htm.

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.