The Report committee for Stephan Garland
Certifies that this is the approved version of the following report:

Examining Gender Bias of GitHub Pull Request Comments with Machine Learning

SUPERVISING COMMITTEE:

Christine Julien, Supervisor

Milos Gligoric

Examining Gender Bias of GitHub Pull Request Comments with Machine Learning

by

Stephan Garland

REPORT

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN ENGINEERING

THE UNIVERSITY OF TEXAS AT AUSTIN

December 2020

Dedicated to my wife Jenn, whose support and encouragement made this endeavor possible.

Acknowledgments

I would like to thank my advisor, Dr. Christine Julien, and my other committee member, Dr. Milos Gligoric, for their efforts and advice. Additionally, Lauren Salinas, the Graduate Program Coordinator, fielded countless questions for me throughout my entire degree progress. Dr. Georgios Gousios, of the GHTorrent project, was instrumental in creating a dataset for large-scale data mining of public software repositories and contributions. Finally, I would be remiss to not thank Alexandra Elbakyan, whose generous contributions to the openness of science are gratefully received.

Examining Gender Bias of GitHub Pull Request Comments with Machine Learning

Stephan Garland, MSE
The University of Texas at Austin, 2020

Supervisor: Christine Julien

Bias is known to exist in many fields, software included. Specifically concerning gender bias, women are traditionally under-represented in technical fields, and previous research has shown a gatekeeping aspect to both the hiring practices and day-to-day work.

One way of investigating the existence and/or severity of bias is via sentiment analysis, using machine learning. This method could be utilized similarly to spellcheck, prompting the comment's author to rethink their comment's tone.

This paper examines that method, using approximately 154,000,000 unique comments from Pull Requests on GitHub sourced from the GHTorrent project, classified using the shallow neural network fastText, and GitHub avatars classified with Amazon Web Services Rekognition.

My results failed to reject the null hypothesis that gender does not have a measurable effect on the sentiment of comments received. However,

additional research could include a larger subset of comments and/or better tune fastText, as well as measuring other biases such as nationalism.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Bias in the technology and software industry is a well-documented issue. Michie and Nelson (2006) found that men had "less positive attitudes toward capabilities of women in IT" as compared to their attitudes towards other men.

One potential avenue for continued bias is in the code review process. Ideally an objective process, it tends to rely more on past participation (Dey and Mockus, 2020) than anything else. While this could be seen as a virtuous cycle — skill begets access — it can be a significant roadblock to people beginning their careers, or who are joining a new organization.

Automated code review tools are being explored, with offerings from both cloud providers and private Software-as-a-Service (SaaS) companies. While this may eliminate an explicit bias, algorithms themselves are subject to bias, either from supervised learning, or their original authors.

A recent example is from a recidivism risk algorithm that ProPublica (Angwin et al., 2016) investigated:

> The formula was particularly likely to falsely flag black defendants as future criminals, wrongly labeling them this way at almost twice the rate as white defendants.

While pull request acceptance is not nearly as problematic as criminal justice mishandling, some of the concepts in algorithm design and implementation crossover.

This report will examine the existence of gendered bias in Pull Request comments, and discuss the use of Natural Language Processing, specifically Sentiment Analysis, to determine an author's expressed sentiment during the code review process. This kind of automated check could be implemented much as spell check is ubiquitous in nearly every application, as a quick intent verification before committing the comment.

1.1 Hypothesis Statement

H_0: A code author's gender has no measurable effect on the sentiment of comments received.

H_1: A code author's gender has a measurable effect on the sentiment of comments received.

1.2 Introductory Notes

Note that throughout this report, the term "gender" should be taken to mean affirmed gender, i.e. the gender the person associates with, unless otherwise specified. Additionally, due to the limitations of image recognition software, only binary gender is examined in this paper.

Chapter 2

Background and Related Work

2.1 Git and Pull Requests

The process of software development has taken many turns since its infancy, but for the purpose of this paper, I will focus on Git and tools that leverage it.

Git has the concept of Pull Requests (PR), which are proposed changes to an underlying code repository, presented as a code commit. This commit contains a differential change to the files modified by the author[s] of the PR, and can stand on its own as an ongoing history of a repository. When a PR is made, the author requests that one or more reviewers consider their work to be integrated into a main code branch. The reviewers may approve this request, deny it, add comments, or some combination thereof. If approved, the author may then merge their code into the main branch. Of note, this is not new to version control systems (VCS); Mercurial has Merge Requests, which are similar. GitHub popularized PRs as they use Git as the underlying VCS, and the term is more commonly used in relation to GitHub than Git.

PRs range in size from minor fixes, such as typographical errors, to gargantuan in scope. While the onus is presumably on the code's author to

produce error-free and performant code, an expectation is also made of the reviewers to recognize potential issues prior to giving their approval. At some point on the scope scale, it becomes infeasible to expect humans to understand and give opinion on code that they had no hand in producing.

As an example, I undertook a project to upgrade an existing code base from Terraform (an Infrastructure-As-Code tool that instantiates cloud hardware via a declarative programming langauge) 0.11 to 0.12. This version bump had many breaking changes, and required the examination and modification of thousands of files. To make this possible in a reasonable timeline, the authors of Terraform produced a tool to automate much of this. Where their tool was unable to complete the conversion, I wrote Python scripts to fill in its gaps. While I had a good understanding of the resultant PR, I cannot expect the reviewers to look through thousands of nearly-identical files - they trust both the tools used, and more germane to this discussion, the humans involved.

2.2 Developer Trust

Trust, in that case, was developed from examining the algorithms in the tools, and from working together - trust that the quality of my work was adequate. How, though, is trust developed for strangers on the internet, contributing to public projects? Absent that trust, how is the decision made to approve or deny a code contribution? As I will show, human bias tends to have an outsized effect on this intrinsic trust.

Dey and Mockus (2020) found that, at least for a subset of repositories,

there is positive correlation between a developer's previous accepted work, and currently submitted work.

This makes sense from the standpoint of an author building trust. Lenarduzzi et al. (2019) found that " ... the quality of the code submitted in a pull request does not influence at all its acceptance."

This further introduces the idea that human biases may play a significant role in the acceptance of code, rather than its objective quality.

2.3 Reviewer Biases

Terrell et al. (2017) investigated gender bias in PRs, and found that while the acceptance rate for PRs was higher for people identifying as women, this only held true when their profiles did not identify them as women. This strongly suggests that reviewers may be relying on factors other than objective code quality — whether the reviewers were more critical when a PR was submitted by a woman, or whether they were more likely to deny it out of hand is not examined in the paper.

Rastogi et al. (2018) found a positive correlation to the geographical location of a PR's author and of the person reviewing the code, i.e. if both were co-located in the same country, the reviewer was more likely to accept the PR. This further cements the idea that factors unrelated to objective code quality. Interestingly, they further found that this correlation did not exist for India. An informal survey taken on Reddit's r/India community suggested that

India's heterogeneity may be a factor, as compared to other large countries with a diverse population, India's cultures tend to be much more distinct. Thus, it may be possible that if there is a tribal nature to reviewers tending to favor commits from their compatriots, this effect is diluted in India due to the unofficial internal borders. While this paper focuses on gender as the independent variable, the dataset used also includes location, and so further research could be done to explore this.

2.4 Similar Work

Imtiaz et al. (2019) found that women "were not generally measured at a stricter standard than men," and also "were less likely to express politeness and profanity than men, and were more restrictive in expressing their sentiments on [GitHub]." Additionally, they found that women have a higher number of review comments when the PR is ultimately not merged, indicating push-back.

2.5 Difficulty in Addressing Bias

Galhotra, Brun, and Meliou (2017) showed that even when bias is targeted during an algorithm's design, it's possible to reach an overall level of fairness by discriminating against certain groups of people. When examining a bank's decision whether or not to grant a loan:

> For example, an algorithm may give the same fraction of white

> and black individuals loans, but discriminate against black Canadian individuals as compared to white Canadian individuals. ... these findings suggest that the software designed to produce fair results sometimes achieved fairness at the global scale by creating severe discrimination for certain groups of inputs.

This must be considered when entrusting bias-checking to an algorithm, as the most efficient method found to reduce bias for one group may inadvertently harm others.

One potential counter for this is a so-called Seldonian algorithm. When using this framework to predict students' GPA based on their entrance exam scores, Thomas et al. (2019) found:

> In particular, our algorithms ensure that, with approximately 95% probability, the expected prediction errors for men and women will be within $e = 0.05$, and both effectively preclude the sexist behavior that was exhibited by the standard ML algorithms.

This algorithm framework could be adapted in sentiment analysis to set certain safeguards during model training, to avoid inadvertent discrimination.

Chapter 3

Methodology

3.1 Comment Pipeline

The following tables show the flow of a comment from its initial extraction from BigQuery, through final sentiment analysis. First, some comments are pulled from BigQuery, and the comment column is extracted.

comment
Fixed, but I cannot accept a suggestion - it's in a different file now.
We reduced visibility on this, but we still use this method if a couple [. . .]
Let's not check these as discussed
not having a return does impact the speed of the api - esp on contribution.get -

Table 3.1: Isolated Comment

I scored these comments as Neutral (2), Neutral (2), and Helpful (h), respectively. Note that the index skips 1 — if a comment is either from a bot or almost entirely code, it may be beneficial to not train the model on those. In this case, it is purely to show the process.

idx	score
0	2
2	2
3	h

Table 3.2: Isolated Comment

Several options are now available for how to label the text — this is an example of only mapping the index and score to the comment and text score. You can also see that two custom comments were added to the bottom, representing an example of positive and negative comments.

label	comment
__label__neutral	Fixed, but I cannot accept a suggestion - it's in a different file now.
__label__neutral	Let's not check these as discussed
__label__helpful	not having a return does impact the speed of the api - esp on contribution.get -
__label__positive	lgtm
__label__negative	this is horrible

Table 3.3: Labeled, No Upsampling, No Preprocessing

More commonly, the comments are upsampled to equalize the types of comments, and preprocessed with lemmatization and regexes to strip out extraneous information.

label	comment
__label__neutral	fixed cannot accept suggestion different file
__label__neutral	check these discussed
__label__helpful	having return impact speed contribution
__label__helpful	having return impact speed contribution
__label__positive	lgtm
__label__positive	lgtm
__label__negative	this is horrible
__label__negative	this is horrible

Table 3.4: Labeled, Upsampled, Preprocessed

The original comment file is then sampled from comments not trained on and saved as a separate CSV for validating the model.

A shell script then shuffles the training file with **shuf**, splits it 80:20 for training and testing, and runs fastText in supervised learning with autotuning

of its hyperparameters. The generated model then makes predictions from the previously exported unseen comments, where they can be compared with the comments using **paste**.

label	comment
__label__neutral	Agreed, its needs cleaning up across the code base.
__label__helpful	Ah, of course. Sorry, hadn't noticed that.

Table 3.5: Predictions

Depending on both the model's reported precision and recall, and human-verified results, more comments can be scored, or further tweaking of the hyperparameters may be done. Once satisfied with the model, its settings can be retrieved for future use and comparison, and it can also be quantized to reduce its size with little to no loss of accuracy. In my case, the model's size was reduced from 2.2 GB to 15 MB.

The entire pipeline is illustrated below — the above tables represent the left half of the figure, as avatar recognition is handled separately.

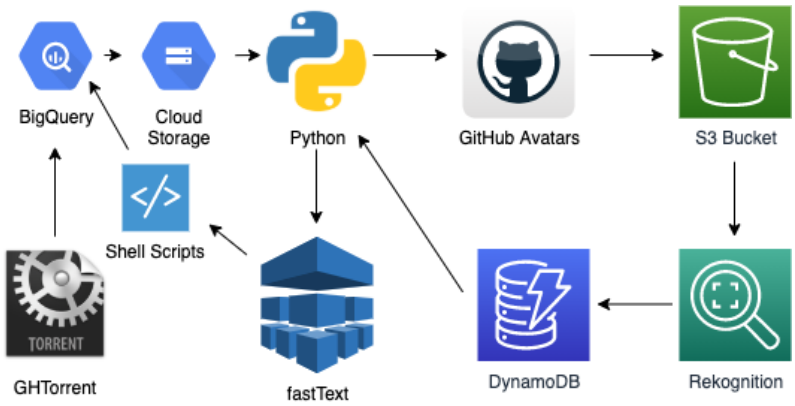


Figure 3.1: Workflow

3.2 Data Sourcing

Obtaining the raw data was courtesy of GHTorrent. I first downloaded the latest MySQL dump (2019-06-01) and attempted to load it into my home server; this failed approximately 24 hours into the load due to an error on my part. In retrospect, had I continued, I would have hit a disk and/or memory limit due to the enormity of some of the created tables. I found the project had been made publicly available on Google BigQuery (BigQuery), and I settled on that for all future work.

3.3 Initial Query

This query returns commits for a given repository, its language, the date, comment, and the login name of both the commit's author and the comment's author, as well as numeric IDs for many of those items. The resultant tables were then exported to Google Cloud Storage (GCS) as GZIP'd CSVs, and from there, downloaded. This intermediate step is a requirement of BigQuery for large datasets.

My first successful query containing all necessary information from 2019-01-01 to the end of the dataset (2019-06-01) resulted in a table approximately 14 TB in size. Running further queries on this would have created costs rapidly exceeding my budget, so I scaled down. I chose one Monday per month, as Monday has the highest number of commits (Georgios Gousios and Diomidis Spinellis 2012, 12-21). I avoided any holidays and varied the week throughout the months for more variability.

To further reduce the query size, I made intermediate tables of all large tables to be queried - for example, the **commits** table is approximately 104 GB in size, whereas my trimmed-down version is only 27 GB. This results in a total query processing of 28 GB, and a resultant table of 10-20 GB.

3.4 Initial Parse

My first attempt at parsing the CSVs was to import them as a Pandas DataFrame, dropping unneeded columns. As the files had been split into chunks, they first needed to be concatenated, with a **map** function being used to concatenate them for use by Pandas. This resulted in extremely high memory consumption and eventually caused system instability, so another method was needed.

I extracted the **author_login** column with **csvcut** and **parallel**:

```
parallel csvcut -c "author_login" {} '>' names_{} ::: $file_prefix*.csv
```

Duplicates were then dropped to reduce the size considerably.

I attempted to parse the names alone in order to determine the user's gender. As usernames frequently are not actual names or even coherent words, this ultimately was unsuccessful. Nevertheless, I have included it for posterity.

3.5 Avatar Scraping

I next opted to scrape user avatars and run them through an image recognition tool. For the purposes of speed and accuracy, rather than building

my own model using open-source machine learning tools, I chose Amazon Web Services (AWS) Rekognition. Google Cloud Vision would have also been an option, however, earlier this year they decided to remove gender recognition capabilities, citing problems with assumed gender. This is indeed a problem, but for the purposes of this research, was irrelevant as the hypothesis hinges on assumed gender, not affirmed.

A Python script was created to obtain avatars, scraping them from `https://avatars.githubusercontent.com` and writing them to a dictionary, with the key being the user's login. This worked well and could be sped up via parallelization if desired.

To save unnecessary calls to Rekognition, I empirically determined that anything less than 12 KB was either the default GitHub logo or a similar cartoon, useless for analysis. These were removed with the following command:

```
find . -size -12k -exec rm {} \;
```

3.5.1 Image Delivery Pipeline

A method to deliver these images to Rekognition was then needed. My first solution was to upload them to an S3 bucket, and from there, sequentially pass them to Rekognition. I thought this was too slow, and so created a pipeline utilizing AWS Lambda and Simple Queue Service (SQS). This, in theory, would have been tremendously faster, as the Lambda should split up the list, and rapidly drain the SQS queue. RabbitMQ was also briefly tested,

as it is significantly more performant and reliable for some situations, this included. A small test was successful, but due to easier integrations, SQS was chosen. Unfortunately, not being experienced with Lambdas, I did not set up concurrency and/or SQS wait and visibility times correctly, resulting in approximately 40 invocations of my Lambda, rather than the thousands expected, before timing out and failing. Approximately 70 images were analyzed and written to the DynamoDB table. I decided to simply let the original script run overnight to continue progress. The Lambda is included for posterity.

An example image (my own) with returned values from Rekognition is below:

Figure 3.2: Example Avatar

confidence	filename	gender
99.83228302001953	example_avatar.jpg	Male

Table 3.6: Avatar Prediction

Once done, the DynamoDB table was exported as JSON. A preliminary check showed that 75.2% of files were able to be classified as either male or female. Logic was included in the script such that if no facial information was found (i.e. the avatar was not human), a small JSON entry would be made, with the gender classified as unknown, and the confidence at 100%. This made for easy filtering in the next step. I set a cutoff at 99% confidence, which resulted in 17,007 avatars to pass to Rekognition. This cutoff and name selection can be done either from BigQuery, or with **jq**:

```
jq -r '.Items[] | select ((.Gender.S != "Unknown") and \
    (.Confidence.N | tonumber >= 99)) | .PK.S' > output
```

3.6 Sentiment Analyzer

I chose fastText as the library for performing sentiment analysis. My primary choice was its speed, as it would allow for a high level of iterative progress. fastText is a shallow neural network utilizing the Continuous Bag of Words model, which attempts to predict a word's sentiment based on its neighbors. It is nearly or as accurate as deep learning models, but trains on a CPU vs. GPU, and is orders of magnitude faster (Joulin et al., 2016).

I built a Python script for loading and working with the data, leaning heavily on the Pandas library for manipulation.

3.6.1 File Format

I was inspired by Stanford's Sentiment Treebank file format and used something similar. A given CSV dump from BigQuery has the **comment** column isolated. This is loaded into a Pandas DataFrame (DF) which contains two columns, idx, and score — see Table 3.2 for an example. The idx column corresponds to the line number of a loaded comment file, and score is a mapped value of a given comment's sentiment as decided by the human trainer.

3.6.2 Data Entry Overview

The DF's rows are iterated over, and the user is presented with a legend to select from to classify a given comment: Harsh, Negative, Neutral, Positive, Helpful, or Terse. Control commands Skip and Save/Quit are also options. Once a valid choice is made, the DF has a row written containing the comment's index, and the user's score. When the control command Save/Quit is given, the DF is appended to a CSV containing the same two columns — idx, and score. A text file also has the last index scored written. The idx column is crucial, as it allows for comments to be skipped. Without it, relying only on the DF's built-in index, the actual comments would gradually drift from those being scored. When resuming, the user is presented with the last index scored, and asked what index they'd like to resume at. This quit/resume cycle can occur any number of times for a given input file.

Additionally, it is possible to create a custom comment list, scored, for later inclusion. For example, "LGTM," short for "Looks Good To Me," is

commonly used both on its own and as part of a larger comment, and carries a large amount of trust and approval with it. Similarly, I struggled to find examples of comments that could be described as Harsh. I attempted to classify them by sourcing the comments of a well-known Linux contributor famous for their vitriol, but the comments were so remarkably offensive that the algorithm ignored lesser slights. Ultimately, I added some common vulgarities one might come across and left the category at that.

3.6.3 Data Labeling

Comments were then manually labeled. I labeled approximately 900 before noticing a modicum of performance from the model. I went as high as 1,600 but found that the model's accuracy had drifted, so I regressed to 1,318, where I had last saved progress without seeing issues. Next, processing began. First, the sentiments CSV were read into a DF, and then merged, using Pandas, with another DF containing all comments loaded, matching the latter on its index, and the right on its **idx** column. This is analogous to a SQL Inner Join. The distribution of my labeled comments is below:

label	count
__label__neutral	848
__label__positive	151
__label__terse	145
__label__helpful	139
__label__negative	35

Table 3.7: Labeled Distribution

3.6.4 Preprocessing

Text preprocessing is optional but highly recommended. I settled on a modified series of regular expressions (regexes) from "Python for NLP: Working with Facebook FastText Library" (Malik, 2020). My changes were to keep stop words, question marks, and emoticons (e.g. :-D). Stop words proved necessary, as the removal of "not", for example, has a dramatic effect on a comment such as "this line does not meet style guidelines." Question marks were kept as I found a large number of terse comments containing only question marks. Finally, emoticons can have outsized effects on the overall tone of a comment. Consider the difference between these two statements: "You forgot to call free()," and "You forgot to call free() ;)." A small inclusion takes what is a dispassionate if not accurate comment, and lightens it. Additionally, I found a large number of comments consisting purely of encoded emoji such as ":heart:" and ":+1:."

3.6.5 Score Mapping

Once preprocessing was complete, a slice of the input DF was iterated over and written to a list of lists containing a comment and its score. This list was then passed to a labeling function which translates its key into the format required by fastText, e.g. 3 $\rightarrow$ __label__positive. Additionally, optional logic was available to flatten scores into a simplified model, consisting only of Negative, Neutral, and Positive. Early in the model, this simplified model was more accurate, but as the corpus grew, sufficient accuracy existed with the full

classification schema.

3.6.6 Custom Comments

Custom comments to guide the training set were appended to the existing list, and it was written to a CSV. A slice from the original input, further down than what was trained on, was exported for testing the model.

3.6.7 Upsampling

Finally, upsampling was performed. This is a process in which the classes of comments are summed, and minority classes are replicated to bring the ratio of all comment classes to be equal. This can be helpful in presenting the model with an equal amount of each class. Given that the overwhelming majority classification of comments was Neutral, as expected, I found this to be enormously helpful, with a dramatic increase in accuracy.

3.7 Model Training

3.7.1 Training Set Preparation

In the command line, a small shell script performs some tasks. The training file exported earlier is shuffled with **shuf**, and split into an 80:20 train:test set. fastText is then run in supervised learning mode, with hyperparameter autotuning. Finally, fastText is run in predict mode on the test set, redirected to a new file. This file is interleaved with the test file with **paste**, and verified with **grep** — for instance, checking the **__label__positive**

comments for reasonableness. Objective accuracy was primarily done by measuring the average loss rate of the generated model. To achieve this, once other parameters had been tuned, I used a high-core count cloud instance and set the number of epochs to 100,000. My final average loss was 0.089752, and my precision and recall at 1 were 0.962

3.7.2 Data Export

The **comment_id** and **comment** columns were extracted from BigQuery and exported to a GCS bucket, where **gsutil** was used in an Elastic Compute Cloud (EC2) instance to sync them, prior to preparing for processing with fastText.

3.7.3 Postprocessing

The required header initially removed from the comment file is added to the beginning of the file via **echo** before the content is appended to it:

```
echo $hdr > $month-comments.csv && cat comments-month*.csv | \
 grep -v 'comment_id,comment' >> $month-comments.csv && \
 xsv select comment $month-comments.csv | tail
```

In previous usage, I found that **csvcut** was fairly slow, as it's a Python program. I found **xsv**, a CSV manipulation program written in Rust, which was far faster.

I encountered some issues with newlines not being respected, so this

writes the necessary header (**comment_id,comment**) to the file before appending each separate extracted CSV from BigQuery. The xsv select command at the end verifies that the new CSV is intact, as it will throw an error when parsing the file if there is a problem. Once concatenated, the comment field was isolated for use by fastText.

3.7.4 Training Execution

For performing the predictions, I used **tmux** to have separate sessions running for each. fastText's predict mode is not multi-threaded, so each invocation will use a single thread. Alternately, **parallel** could be used for this purpose, as well as splitting the files up and invoking more instances.

3.7.5 Prediction Merging

To merge the predictions back into the CSV file, a header must first be prepended to the prediction file. This can be done via **ex**, which on files this large proved to be quite a bit faster than the usual approach of **sed.**

```
ex -snc '1i|HEADER_TO_INSERT' -cx file
```

Then, the files can be merged with:

```
paste -d, file.csv predictions.txt > merged_file.csv
```

3.7.6 Post-Postprocessing

I noted some extraneous spare lines at the end of the model's predicted file. I tracked down one error to the concatenation causing a loss of newline characters but was unable to find all causes. I found some other users on the project's GitHub page discussing similar issues. I checked for lines matching up between the original and commented files and found them to be the same, so I felt confident in deleting the extra lines. Again, this can be done more quickly via ex:

```
ex -snc '$-NUM_LINES_EXCLUSIVE_FROM_END_TO_DELETE,$d|x' file
```

Of note, this will consume an amount of memory equal to the file's size, so ensure you have adequate memory if you run this in parallel.

From there, the labeled files can either be directly analyzed with Python, R, etc. or loaded back into BigQuery. I loaded them into BigQuery, joined them with the original content, and then extracted gender and sentiment as rows for analysis with Python, the code for which is in the Appendix.

Chapter 4

Results

4.1 Report Limitations

This report has multiple limitations, the largest of which is the model. Supervised machine learning is entirely dependent upon the accuracy of its model, and has several sub-factors that can skew the apparent accuracy, such as over-fitting to training data. With a total population size of 154,270,017, my training dataset of 1,318 labels has a margin of error of 3% as given by Cochran's formula, after removing 20% of the training set for validation.

4.1.1 Margin of Error

$$e = \frac{\sqrt{p}\times\sqrt{q}\times Z}{\sqrt{n}} \therefore e = \frac{\sqrt{0.5}\times\sqrt{0.5}\times 1.96}{\sqrt{1318\times 0.8}} = 0.0302$$

4.1.2 Ground Truth

Its second-largest limitation is the method of ground truth, viz. image recognition. Image recognition software is notoriously easy to fool while giving little to no indication to human viewers that any manipulation has occurred.

> ... by adding such a quasi-imperceptible perturbation to natural images, the label estimated by the deep neural network is changed with high probability. (Moosavi-Dezfooli et al., 2017)

Tangentially, some users choose to have pictures of (presumably their own) children as their avatar. A reasonably good counter for this would be to filter returned values from Rekognition for age.

4.1.3 Industry Specificity

Thirdly, developers do not work in the same fields. StackOverflow conducts a yearly survey—2020's had nearly 65,000 respondents— and shows that the ratio of men:women across various computer science fields ranges from roughly 10:1-30:1, with women being far more likely to work in Data Science, QA, and Frontend than Site Reliability Engineering, Database Administration, or DevOps. Adjusting the analysis to reflect those values with respect to the category of the repository under analysis may show different results.

4.1.4 Human Bias

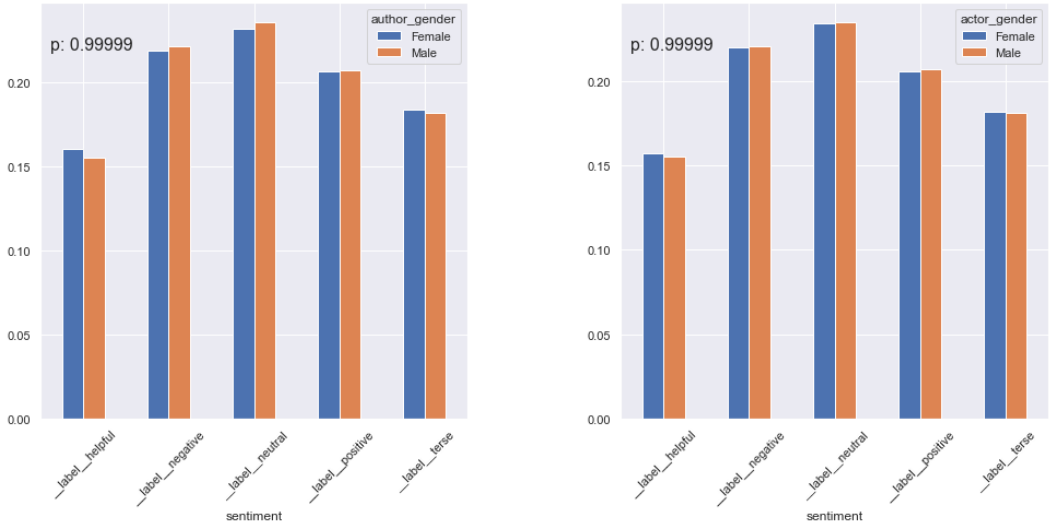
Finally, my personal bias in labeling text must be recognized. What I may think of as negative may be seen as helpful criticism by another. Moreover, the difference between constructive criticism and nitpicking can be razor-thin, and difficult to objectively measure.

4.2 Hypothesis Result

The null hypothesis for this report is that gender has no effect on a PR's comment sentiment. I analyzed a meaningful subset of available data, and was unable to disprove the null hypothesis; namely, the p-value approached 1.

The p-value was calculated using a chi-squared test, from the Python library **scipy**. I passed into it a cross-tabulation of the comments, generated with Pandas, and normalized.

Figure 4.1(a) shows the relative probability of a comment's sentiment when received by a commit author, grouped by gender. Figure 4.1(b) shows the relative probability of a comment's sentiment when made by a commenter, grouped by gender.



(a) Commit Authors (b) Commit Actors

Figure 4.1: Comment Sentiment by Gender

In both cases, it can be seen that while there is a small difference, notably in the **helpful**, **negative**, and **neutral** sentiments, the p-value is such that this distribution is expected, and the null hypothesis is not disproven.

Chapter 5

Conclusions and Future Research

5.1 Conclusions

I investigated the hypothesis that a code author's gender has a measurable effect on the sentiment of comments received. This was done via a subset of a larger dataset, with my subset containing roughly 154,000,000 examples of comments. I used image recognition to determine an author's gender, and a shallow neural network with supervised training to determine comment sentiment. I was unable to disprove the null hypothesis, thus the question remains outstanding.

5.2 Future Research

- The model's accuracy could be improved, ideally with additional training data.
- Other machine learning algorithms such as Gensim could be used for sentiment analysis.
- Other days of the week could be checked to see if, for instance, Friday is friendlier than Monday.

- Commit author and commenter location (country or region) could be checked in the same manner as gender. This has the added benefit of not requiring image recognition and its limitations.
- Commit author and commenter age could be sourced either by image recognition estimates or cross-referencing social media to investigate age bias.
- Seniority of an individual could be obtained from LinkedIn or similar professional social media platforms to determine if years of experience play a role in sentiment.

Appendix

Appendix 1

Code Listing

1.1 Shell Script

1.1.1 prepare.sh

```
#!/usr/bin/env bash

cd ../BQ\ Results/comments

mv training.csv ../../ml/training.txt
mv test.csv ../../ml/test.txt

cd ../../ml

for i in training.txt test.txt; do
    sed -i 's/^"//' $i
    sed -i 's/"$//' $i
done

rm model*
shuf training.txt > temp.txt
head -$(echo "$(echo $(wc -l training.txt) | cut -d" " -f1) / 5" | bc) \
    temp.txt > validate.txt
tail -$(echo "$(echo $(wc -l training.txt) | cut -d" " -f1) / 1.25" | bc) \
    temp.txt > training.txt

rm temp.txt

./fasttext supervised -input training.txt -output model \
    -autotune-validation validate.txt -autotune-duration 300
./fasttext predict model.bin test.txt > predictions.txt
```

1.2 SQL

1.2.1 Combined Table

```
-- This presupposes that you have created a schema as shown.
-- Replace schema and the ending date range as desired.
-- Also note that two databases are being queried here,
-- as I saw no reason to replicate the smaller tables.
SELECT
  u1.login AS actor_login,
  prc.user_id AS actor_id,
  prc.comment_id AS comment_id,
  prc.body AS comment,
  p.name AS repo,
  p.language AS language,
  u2.login AS author_login,
  c.author_id AS author_id,
  prc.pull_request_id AS pr_id,
  prc.commit_id AS c_id,
  c.created_at AS commit_date
FROM
  `ghtorrent-290615.project_commits.pr_comments_small` prc
JOIN
  `ghtorrent-290615.project_commits.project_commits_small` pc
ON
  pc.commit_id = prc.commit_id
JOIN
  `ghtorrent-290615.project_commits.commits_small` c
ON
  pc.idproject_id = c.project_id
JOIN
  `ghtorrentmysql1906.MySQL1906.projects` p
ON
  c.project_id = p.id
JOIN
  `ghtorrentmysql1906.MySQL1906.pull_request_history` prh
ON
  prc.pull_request_id = prh.pull_request_id
JOIN
  `ghtorrentmysql1906.MySQL1906.users` u1
ON
```

```
  prc.user_id = u1.id
JOIN
  'ghtorrentmysql1906.MySQL1906.users' u2
ON
  c.author_id = u2.id
WHERE
  p.deleted = 0
AND
  p.forked_from IS NULL
AND
  prh.action = "opened"
AND
  c.created_at BETWEEN "$DATE_START" AND "$DATE_END"
;
```

1.2.2 Assign Labels

```
  -- This presupposes that you've loaded the labeled comment results.
SELECT DISTINCT
    cm.actor_login,
    ug1.gender AS actor_gender,
    cm.actor_id,
    cm.repo,
    cm.language,
    cm.comment,
    ml.label AS sentiment,
    cm.author_login,
    ug2.gender AS author_gender,
    cm.author_id,
    cm.comment_id,
    cm.pr_id,
    cm.c_id AS commit_id,
    cm.commit_date
FROM 'ghtorrent-290615.project_commits.combined_$month' cm
JOIN
    'ghtorrent -290615.project_commits.$month_load' ml
ON
  cm.comment_id = ml.comment_id
JOIN
```

```
    'ghtorrent -290615.project_commits.users_gendered_high_prob' ug1
ON
  cm.actor_login = ug1.login
JOIN
  'ghtorrent-290615.project_commits.users_gendered_high_prob' ug2
ON
  cm.author_login = ug2.login
;
```

1.3 Python

1.3.1 Avatar Scraper

```
#!/usr/bin/env python

import glob
from io import BytesIO
import os
from pathlib import Path
import requests

import numpy as np
import pandas as pd
from PIL import Image

ava_dict = {}
save_path = "$YOUR_PATH"

def get_names():
    """
    This presupposes that you have extracted the author_login column and are
    globbing only those files. It will work on the entire files, but unless
    you have >= 32 GB of RAM and a fast SSD, you're unlikely to be pleased
    with the results.

    Args:
        None.

    Returns:
        A Pandas DataFrame.
    """
    df_names = pd.concat(
        map(
            pd.read_csv, glob.glob(
                os.path.join(r"$YOUR_CSV_PATH", "$YOUR_PREFIX*.csv")
            )
        )
    )
    df_names = df_names.drop_duplicates()
    df_names.to_csv(r"$YOUR_CSV_PATH", index=False)
```

```
    return df_names

def resume(df_names):
    """
    In the event of an interruption, the following code can be used to quickly
    determine what avatars exist vs. what remain to be scraped. Note that
    np.setdiff1d() returns values (ar1 not in ar2), and is not symmetric,
    so comment out whichever line you don't need.

    Args:
        df_names: A DataFrame returned from get_names().

    Returns:
        A list containing files that have not yet been downloaded.
    """
    avatars = glob.glob(save_path + "*.png")
    # This will return a list of all filenames you have saved.
    existing_names = [Path(x).name.split(".")[0] for x in avatars]
    # This will return a list of names that exist in df_names but are not
    # yet saved to disk.
    existing_diff = np.setdiff1d(df_names["author_login"].tolist(), existing_names)

    return existing_diff.tolist()

def scrape(ava_dict, df_names):
    """
    This scrapes avatars from githubusercontent.

    Args:
        ava_dict: A dictionary consisting of filename: file_content.
        df_names: A DataFrame with filenames to be downloaded.

    Returns:
        Nothing.

    Raises:
        TypeError: Ignores.
    """
    # Replace the iterable with one generated above if needing to resume scraping.
```

```
    for i in df_names:
        if not ava_dict.get(i):
            try:
                r = requests.get("https://avatars.githubusercontent.com/" + i)
                ava_dict[i] = Image.open(BytesIO(r.content))
                # Optional, but provides some measure of progress.
                if not len(ava_dict) % 500:
                    print("Current length: {} out of {}".format(len(ava_dict),
                        len(df_names)))
            # Some avatars resulted in Image.open() parsing errors; they were
                dropped.
            except TypeError:
                pass
    print("All done")

def save_and_dedupe(ava_dict):
    """
    Saves the dictionary's files to disk, ignoring duplicates for faster I/O.

    Args:
        ava_dict: A dictionary consisting of filename: file_content.

    Returns:
        Nothing.
    """
    dupes = []
    for i in ava_dict:
            file_exists = Path(save_path + i + ".png")
            if not file_exists.is_file():
                ava_dict[i].save(save_path + i + ".png")
            else:
                dupes.append(i)
    print("Saved {} files, ignored {} duplicates".format(
        len(ava_dict) - len(dupes), len(dupes)))

def main():
    df_names = get_names()
    scrape(ava_dict, df_names)
    save_and_dedupe(ava_dict)
```

1.3.2 Levenshtein Finder

```
#!/usr/bin/env python

import fuzzy_pandas as fpd

def levenshtein(df_input, df_names):
    """
    Attempts to determine a user's gender by comparing their username to
    tradtionally gendered names. Utilizes Levenshtein distance. Assumes
    two Pandas Dataframes with columns "author_login" and "name". The
    threshold can be varied as needed; I empirically determined 0.72
    to be the most accurate.

    Args:
        df_input: A DataFrame with unknown usernames to check.
        df_names: A DataFrame with traditionally gendered names,
                  i.e. only names traditionally associated with
                  men or women.

    Returns:
        A Fuzzy Pandas DataFrame consisting of names which match up.
    """
    results = fpd.fuzzy_merge(df_input, df_names,
                  left_on="author_login",
                  right_on="name",
                  method="levenshtein",
                  threshold=0.72,
                  keep="match")
    return results

results = levenshtein(df_input, df_names)
```

1.3.3 SQS Publisher

```
#!/usr/bin/env python

import hashlib

import boto3

def setup_session(profile_name, region):
    """
    Sets up boto3 for instantiating other resources.
    Assumes that you have named credentials saved in ~/.aws/credentials.
    Note that all resources should exist in the same region.
    Args:
        profile_name: The profile name to be used from ~/.aws/credentials.
        region: The AWS region your resources exist in.


    Returns:
        A boto3 Session.
    """
    session = boto3.session.Session(
        region_name=region,
        profile_name=profile_name
    )

    return session

def setup_s3(session, bucket_name):
    """
    Instantiates an S3 resource.

    Args:
        session: A boto3 Session.
        bucket_name: Your S3 bucket's name.

    Returns:
        An S3 bucket.
    """
    s3_resource = session.resource("s3")
```

```
    bucket = s3_resource.Bucket(bucket_name)

    return bucket

def setup_sqs(session):
    """
    Instantiates a SQS resource.

    Args:
        session: A boto3 Session.

    Returns:
        An SQS client.
    """
    sqs_client = session.client("sqs")

    return sqs_client

def send(files, queue_url, sqs_client, n=10, send_all=False):
    """
    This sends n, or optionally all, filenames into an SQS queue.
    This presupposes that you have set up a standard SQS queue.

    Args:
        files: A list of filenames to send.
        queue_url: The URL of your SQS queue.
        sqs_client: The SQS client instantiated by setup_sqs().
        [n]: The number of filenames to send. Defaults to 10.
        [send_all]: If True, ignores n and sends all filenames. Defaults to False.

    Raises:
        KeyError: Ignores exception; indicates either that there were successful
                  entries, or failures.

    Returns:
        A response message indicating the number of files sent, and a list of all
        failed files (if any).
    """
    cnt = 0
```

```
    success = []
    failed = []
    for f in files:
        if cnt >= n and not send_all:
            break
        response = sqs_client.send_message_batch(
            QueueUrl=queue_url,
            # The ID is the filename hashed, which can be used later for
            # deleting the message.
            Entries=[
                {
                    "Id": hashlib.md5(f.key.encode("utf-8")).hexdigest(),
                    "MessageBody": f.key
                }
            ]
        )
        cnt = cnt + 1
        try:
            success.append((f.key, response["Successful"][0]["Id"]))
        except KeyError:
            pass
        try:
            failed.append((f.key, response["Failed"][0]["Id"]))
        except KeyError:
            pass
    ret = "Sent {} messages, {} failed: {}".format(
        len(success), len(failed), failed)
    return ret

def main():
    queue_url = "$QUEUE_URL"
    region = "$REGION"
    profile_name = "$PROFILE_NAME"
    session = setup_session(profile_name, region)
    bucket = setup_s3(session)
    sqs_client = setup_sqs(session)
    files = list(bucket.objects.all())
    resp = send(files, queue_url, sqs_client)
    print(resp)
```

1.3.4 Lambda Handler

```
#!/usr/bin/env python

from decimal import Decimal
import json
import logging
import sys
import traceback

import boto3
from botocore.config import Config

logger = logging.getLogger()
logger.setLevel(logging.INFO)
region = "$REGION"
bucket_name = "$BUCKET_NAME"
queue_url = "$QUEUE_URL"

reg_config = Config(region_name=region)
s3_resource = boto3.resource("s3", config=reg_config)
sqs_client = boto3.client("sqs", config=reg_config)
bucket = s3_resource.Bucket(bucket_name)
rekognition = boto3.client("rekognition", config=reg_config)

def detect_gender(photo, bucket):
    """
    Passes a photo into Rekognition from S3.

    Args:
        photo: A filename to retrieve from the bucket.
        bucket: The bucket's name.

    Returns:
        A tuple containing the filename and the assumed gender, with confidence.

    Raises:
        IndexError: If the photo is not human, no details will be returned.
                    An easily identifiable marker is made to filter out later.
    """
```

```
    response = rekognition.detect_faces(
        Image={
            'S3Object':
            {
                'Bucket': bucket, 'Name': photo
            }
        },
        Attributes=["ALL"]
    )
    try:
        ret = response["FaceDetails"][0]["Gender"]
    except IndexError:
        ret = {"Value": "Unknown", "Confidence": 100}
    logger.info("Response: ".format(response))
    return photo, ret

def write_table(rek_ret):
    """
    Writes results returned from detect_gender() to a DynamoDB table.
    Presupposes that you have created the table.

    Args:
        rek_ret: The return value from detect_gender()

    Returns:
        Nothing.
    """
    table = boto3.resource("dynamodb").Table("ghtorrent-faces")
    table.put_item(Item={
        "PK": rek_ret[0],
        "Gender": rek_ret[1]["Value"],
        "Confidence": Decimal(rek_ret[1]["Confidence"])
    })

def delete_msg(receipt_handle):
    """
    Deletes a message from the SQS queue given a receipt handle.

    Args:
```

```
        receipt_handle: A receipt handle.

    Returns:
        Nothing.
    """
    sqs_client.delete_message(
        QueueUrl=queue_url,
        ReceiptHandle=receipt_handle
    )

def recv():
    """
    Retrieves a message from an SQS queue.
    Presupposes that you've created the queue.

    Args:
        None.

    Returns:
        The body of a message, containing a filename.

    Raises:
        KeyError: If a message can't be retrieved, usually due to IAM errors.
        Exception: Catch-all, raises exception.

    """
    recvd = []
    export = []
    while True:
        try:
            response = sqs_client.receive_message(
                QueueUrl=queue_url,
                MaxNumberOfMessages=10,
                MessageAttributeNames=[
                    'All'
                ],
                VisibilityTimeout=30,
                WaitTimeSeconds=15
            )
```

```
            logger.info("Received {}".format(response))
            for x in response["Messages"]:
                if x["MD5OfBody"] not in recvd:
                    recvd.append(x["MD5OfBody"])
                    export.append(x["Body"])
                    delete_msg(x["ReceiptHandle"])
        except KeyError:
            exception_type, exception_value, exception_traceback = sys.exc_info()
            traceback_string = traceback.format_exception(
                exception_type, exception_value, exception_traceback)
            err_msg = json.dumps({
                "errorType": exception_type.__name__,
                "errorMessage": str(exception_value),
                "stackTrace": traceback_string
            })
            logger.error(err_msg)
            break
        except Exception as exp:
            exception_type, exception_value, exception_traceback = sys.exc_info()
            traceback_string = traceback.format_exception(
            exception_type, exception_value, exception_traceback)
            err_msg = json.dumps({
            "errorType": exception_type.__name__,
            "errorMessage": str(exception_value),
            "stackTrace": traceback_string
            })
            logger.error(err_msg)

    return export

def lambda_handler(event, context):
    """
    The Lambda handler. Runs other functions, logs results for troubleshooting.

    Args:
        All handled by AWS Lambda.

    Returns:
        Nothing.
```

```
    """
    file_chunks = recv()
    try:
        for f in file_chunks:
            logger.info("Working on file {}".format(f))
            gender_resp = detect_gender(f, bucket_name)
            write_table(gender_resp)
    except Exception as exp:
        exception_type, exception_value, exception_traceback = sys.exc_info()
        traceback_string = traceback.format_exception(
            exception_type, exception_value, exception_traceback)
        err_msg = json.dumps({
            "errorType": exception_type.__name__,
            "errorMessage": str(exception_value),
            "stackTrace": traceback_string
        })
        logger.error(err_msg)
```

1.3.5 Sentiment Classifier

```
#!/usr/bin/env python

import csv
import os
import re

import nltk
from nltk.stem import WordNetLemmatizer
import pandas as pd

nltk.download('punkt')
nltk.download('wordnet')
stemmer = WordNetLemmatizer()

output_headers = ["idx", "score"]

def reset_df():
    """
    A small helper function to reset the DataFrame during testing.

    Args:
        None.

    Returns:
        Nothing.
    """
    global sentiment_df
    sentiment_df = pd.DataFrame(columns=output_headers)

def reset_csv():
    """
    A small helper function to reset the outputted CSV and current
    index during testing.

    Args:
        None.

    Returns:
```

```
        Nothing, but writes to a file.
    """
    global sentiment_path
    with open(sentiment_path, "w") as f:
        f.write("idx, score\n")
    with open(csv_path + "last_idx.txt", "w") as f:
        f.write("0")

def check_sentiment(df,
                    output_file,
                    output_headers=output_headers,
                    start_iloc=0
                    ):
    """
    Writes sentiment values for use by supervised learning.

    Args:
        df: A Pandas DataFrame consisting of comments to be printed.
        output_file: A CSV file path to be appended to.
        output_headers: A list containing headers for the DataFrame.
                        Defined above.
        start_iloc: The iloc in the DF to start at. Defaults to 0.

    Returns:
        Nothing, but writes to a file.
    """
    output_df = pd.DataFrame(columns=["idx", "score"])
    print("\n0:\tharsh")
    print("1:\tnegative")
    print("2:\tneutral")
    print("3:\tpositive")
    print("h:\thelpful")
    print("q:\tsave and quit")
    print("s:\tskip")
    print("t:\tterse")
    for idx, row in df.iloc[start_iloc:].iterrows():
        print("")
        score = input(row[0] + " ")
        while score not in ["0", "1", "2", "3", "h", "q", "s", "t"]:
```

```
            print("Error, please select from [0, 1, 2, 3, h, q, s, t]")
            score = input(row[0] + " ")
        if score == "s":
            continue
        if score == "q":
            output_df.to_csv(sentiment_path,
                             mode="a",
                             header=False,
                             index=False
            )
            with open(csv_path + "last_idx.txt", "w") as last_idx:
                last_idx.write(str(idx - 1))
                print("Index: {}".format(str(idx - 1)))
                break
        else:
            output_df = output_df.append({"idx": idx, "score": score},
                ignore_index=True)
    return output_df

def run(sentiment_df):
    """
    Runs the analysis by calling check_sentiment().

    Args:
        sentiment_df: A Pandas DataFrame, empty or containing existing
            sentiments.

    Returns:
        A Pandas DataFrame containing the sentiments.
    """
    if sentiment_df.empty:
        ignore_empty = input("The dataframe is empty. Would you like to \
                              continue anyway [yn]? ")
        if ignore_empty in "nN":
            print("Exiting - recover the dataframe with pd.read_csv().")
            return
    try:
        with open(csv_path + "last_idx.txt", "r") as f:
            cur_idx = f.read()
```

```
    except OSError:
        reset_csv()
        with open(csv_path + "last_idx.txt", "r") as f:
            cur_idx = f.read()

    print("Last index saved: {}".format(cur_idx))
    start = int(input("Start index: "))

    return sentiment_df.append(check_sentiment(input_df,
                                               output_headers,
                                               sentiment_path,
                                               start),
                               ignore_index=True)

def preprocess_text(document):
    """
    This was taken from
    https://stackabuse.com/python-for-nlp-working-with-facebook-fasttext-
        library/
    and modified for use.
    The emoticon regex was taken from
    https://stackoverflow.com/a/59890719/4221094

    Specifically, I want to retain emoticons at the expense of stripping out
    special characters, as they often carry enormous weight in the tone of a PR.
    Likewise, I am ignoring /?+/ as it is often a comment on its own,
    albeit a terse one.

    Args:
        A string to process.

    Returns:
        A processed string.
    """
    # Check for emoticons
    emoticons = re.search(r"(>?[\:\;X][\-=]*[3\)D\(>sp}])", document, re.I)

    # Remove all the special characters, excepting "?", if no emoticons present
    if not emoticons:
```

```
        document = re.sub(r"(?!\?)\W", " ", str(document))

    # remove all single characters
    document = re.sub(r"\s+[a-zA-Z]\s+", " ", document)

    # Remove single characters from the start
    document = re.sub(r"\^[a-zA-Z]\s+", " ", document)

    # Substituting multiple spaces with single space
    document = re.sub(r"\s+", " ", document, flags=re.I)

    # Removing prefixed "b"
    document = re.sub(r"^b\s+", "", document)

    # Converting to Lowercase
    document = document.lower()

    # Lemmatization
    tokens = document.split()
    tokens = [stemmer.lemmatize(word) for word in tokens]
    #tokens = [word for word in tokens if word not in en_stop]
    tokens = [word for word in tokens if len(word) > 3]

    preprocessed_text = " ".join(tokens)

    return preprocessed_text

def upsampling(input_file, output_file, ratio_upsampling=1):
    """
    This was taken from
    https://towardsdatascience.com/fasttext-sentiment-analysis-for-tweets-a-
        straightforward-guide-9a8c070449a2
    without further modification, save for this docstring.

    Args:
        input_file: an input CSV containing sentiments.
        output_file: an output CSV containing sentiments, upsampled -
                    can be the same as input_file.
        ratio_upsampling: the ratio of each minority class:majority class.
```

```
    Returns:
        None, but writes to a file.
    """
    i=0
    counts = {}
    dict_data_by_label = {}
    i=0
    counts = {}
    dict_data_by_label = {}
# GET LABEL LIST AND GET DATA PER LABEL
    with open(input_file, 'r', newline='') as csvinfile:
        csv_reader = csv.reader(csvinfile, delimiter=',', quotechar='"')
        for row in csv_reader:
            counts[row[0].split()[0]] = counts.get(row[0].split()[0], 0) + 1
            if not row[0].split()[0] in dict_data_by_label:
                dict_data_by_label[row[0].split()[0]]=[row[0]]
            else:
                dict_data_by_label[row[0].split()[0]].append(row[0])
            i=i+1
            if i%10000 ==0:
                print("read" + str(i))
# FIND MAJORITY CLASS
    majority_class=""
    count_majority_class=0
    for item in dict_data_by_label:
        if len(dict_data_by_label[item])>count_majority_class:
            majority_class= item
            count_majority_class=len(dict_data_by_label[item])

    # UPSAMPLE MINORITY CLASS
    data_upsampled=[]
    for item in dict_data_by_label:
        data_upsampled.extend(dict_data_by_label[item])
        if item != majority_class:
            items_added=0
            items_to_add = count_majority_class - len(dict_data_by_label[item])
            while items_added<items_to_add:
                data_upsampled.extend(
```

```
                    dict_data_by_label[item][:max(
                        0,min(items_to_add-items_added,
                            len(dict_data_by_label[item])))
                        ]
                )
                items_added = items_added + max(0,min(
                    items_to_add-items_added,len(dict_data_by_label[item]))
                    )
# WRITE ALL
    i=0
    with open(output_file, 'w') as txtoutfile:
            for row in data_upsampled:
                txtoutfile.write(row+ '\n' )
                i=i+1
                if i%10000 ==0:
                    print("writer" + str(i))

def post_process(sentiment_path,
                 df_comments,
                 upsample=False,
                 preprocess=True,
                 custom=False,
                 simple=False,
                 concat=False
                ):
    """
    Runs post-processing (and calls preprocess()) to generate labeled comments,
    and optionally upsamples the data to have more equal sentiments.

    This should ideally be split into separate functions.

    Args:
        sentiment_path: The path to the sentiment file.
        df_comments: The main Pandas DataFrame containing comments.
        upsample: If True, upsamples the data. Defaults to False.
        preprocess: If True, preprocesses the data. Defaults to True.
        custom: If True, appends custom comments for processing.
                Expects a list of lists, defaults to False.
        simple: If True, uses a simplified classifier. Defaults to False.
```

```
    concat: If True, appends to a shared file. Defaults to False.

Returns:
    Nothing, but writes to files.
"""
append_or_overwrite = "a" if concat else "w"
sentiments = pd.read_csv(sentiment_path)
training_path = csv_path + "training.csv"
test_path = csv_path + "test.csv"
df_merged = df_comments.merge(sentiments,
                              left_index=True,
                              right_on="idx"
                              )
df_merged = df_merged.rename(columns=
                             {
                                 0: "comment",
                                 "idx": "idx",
                                 "score": "score"
                             }
                            )

def pre_process(input_df, preprocess):
    """
    Appends necessary columns to a list, optionally running pre-processing.

    Args:
        input_df: An input Pandas DataFrame, or a list of custom comments.
        preprocess: If True, preprocesses the data.
                    Called from parent function post_process().

    Returns:
        A list.
    """
    processed_list = []
    if type(input_df) == pd.core.frame.DataFrame:
        for x in input_df.iloc[:,[0,2]].values.tolist():
            if preprocess:
                processed_list.append([preprocess_text(x[0]), str(x[1])])
            else:
```

```
                processed_list.append([x[0], x[1]])
    else:
        for x in input_df:
            if preprocess:
                processed_list.append([preprocess_text(x[0]), str(x[1])])
            else:
                processed_list.append([x[0], x[1]])
    return processed_list


def list_labeler(input_list, simple):
    """
    Converts added labels to fastText format.

    Args:
        input_list: The input list containing coded labels.
        simple: Boolean that determines the complexity of the classier.
                Obtained from parent function.

    Returns:
        A list containing fastText labels and comments.

    """
    labeled_list = []
    for x in range(len(input_list)):
        if not simple:
            if input_list[x][1] == "0":
                labeled_list.append("__label__harsh " + input_list[x][0])
            elif input_list[x][1] == "1":
                labeled_list.append("__label__negative " + input_list[x][0])
            elif input_list[x][1] == "2":
                labeled_list.append("__label__neutral " + input_list[x][0])
            elif input_list[x][1] == "3":
                labeled_list.append("__label__positive " + input_list[x][0])
            elif input_list[x][1] == "h":
                labeled_list.append("__label__helpful " + input_list[x][0])
            elif input_list[x][1] == "t":
                labeled_list.append("__label__terse " + input_list[x][0])
        else:
```

```
                if input_list[x][1] in "01t":
                    labeled_list.append("__label__negative " + input_list[x][0])
                elif input_list[x][1] == "2":
                    labeled_list.append("__label__neutral " + input_list[x][0])
                elif input_list[x][1] in "3h":
                    labeled_list.append("__label__positive " + input_list[x][0])

        return labeled_list

    if preprocess:
        processed_list = pre_process(df_merged, True)
    else:
        processed_list = pre_process(df_merged, False)
    labeled_list = list_labeler(processed_list, simple)

    if custom:
        if preprocess:
            custom_list = pre_process(custom, True)
        else:
            custom_list = pre_process(custom, False)
        custom_list = list_labeler(custom, simple)
        for x in custom_list:
            labeled_list.append(x)

    pd.DataFrame(labeled_list).to_csv(training_path,
                                      header=False,
                                      index=False,
                                      mode=append_or_overwrite
                                      )

    # You should change this to a different range if you go this far into the
        dataset.
    df_comments[10000:12000].to_csv(test_path, header=False, index=False)
    if upsample:
        upsampling(training_path, training_path)

def make_custom():
    """
    Makes a custom set of comments to be trained on.
```

```
    Args:
        None.

    Returns:
        A formatted list for use by post_process()
    """

    custom_list = []
    print("\n0:\tharsh")
    print("1:\tnegative")
    print("2:\tneutral")
    print("3:\tpositive")
    print("h:\thelpful")
    print("q:\tsave and quit")
    print("t:\tterse")

    while True:
        comment = input("\nPlease enter a comment; case is insensitive: ")
        if comment == "q":
            break
        score = input("Please score that comment per the above key: ")
        while score not in ["0", "1", "2", "3", "h", "q", "t"]:
            print("Error, please select from [0, 1, 2, 3, h, q, t]")
            score = input("Please score that comment per the above key: ")
        custom_list.append([comment, score])
    # Save it as a CSV as a backup
    pd.DataFrame(custom_list).to_csv(csv_path + "custom_list.csv",
                                     header=False,
                                     index=False
                                     )

    return custom_list

# Anything you put in here will be evaluated literally, so be careful.
csv_path = input("Path to CSVs (hint: can type os.getcwd(): ")
csv_path = eval(csv_path) if csv_path == "os.getcwd()" else csv_path
csv_path = csv_path + "/"
input_csv_file = input("CSV filename without extension: ")
```

```
input_csv_file_path = csv_path + input_csv_file + ".csv"
sentiment_path = csv_path + input_csv_file + "_sentiments.csv"

try:
    input_df = pd.read_csv(input_csv_file_path, header=None)
except FileNotFoundError:
    reset_csv()
    input_df = pd.read_csv(input_csv_file_path, header=None)

custom_list = make_custom()

try:
    sentiment_df = run(sentiment_df)
except NameError:
    reset_df()
    sentiment_df = run(sentiment_df)

# This is used to load in previously scored sentiments to process them and merge.
# It will first overwrite any existing files –
# use append in the second run of the current data.
# It is far more useful if you're running this in a Jupyter Notebook.

#month_to_load = "jan"
#past_sentiment_path = csv_path + "sentiments_" + month_to_load + ".csv"
#past_month_df = pd.read_csv(csv_path + month_to_load + "_comments.csv.gz",
# header=None
# )
#post_process(past_sentiment_path,
# past_month_df,
# upsample=True,
# preprocess=True,
# custom=custom_list,
# simple=False,
# concat=False)

# As above, these are more useful if running this in a Jupyter Notebook.

#reset_csv()
#reset_df()
```

Index

References

[1] Julia Angwin, Jeff Larson, Surya Mattu, and Lauren Kirchner. Machine bias. `https://www.propublica.org/article/machine-bias-risk-assessments-in-criminal-sentencing`, May 2016. Accessed 2020-09-12.

[2] Tapajit Dey and Audris Mockus. Which pull requests get accepted and why? a study of popular npm packages. In *Proceedings of ACM Conference*, New York, NY, 2017. ACM.

[3] Sainyam Galhotra, Yuriy Brun, and Alexandra Meliou. Fairness testing: Testing software for discrimination. pages 498–510, 08 2017.

[4] Georgios Gousios. The ghtorrent dataset and tool suite. pages 233–236, 05 2013.

[5] Georgios Gousios and Diomidis Spinellis. Ghtorrent: Github's data from a firehose. *IEEE International Working Conference on Mining Software Repositories*, pages 12–21, 06 2012.

[6] Nasif Imtiaz, Justin Middleton, Joymallya Chakraborty, Neill Robson, Gina Bai, and Emerson Murphy-Hill. Investigating the effects of gender bias on github. 05 2019.

[7] Armand Joulin, Edouard Grave, Piotr Bojanowski, and Tomas Mikolov. Bag of tricks for efficient text classification. In *Proceedings of the 15th*

Conference of the European Chapter of the Association for Computational Linguistics: Volume 2, Short Papers, pages 427–431. Association for Computational Linguistics, April 2017.

[8] Charles Malafosse. Fasttext sentiment analysis for tweets: A straightforward guide. `https://towardsdatascience.com/fasttext-sentiment-analysis-for-tweets-a-straightforward-guide-9a8c070449a2`, 10 2019. Accessed 2020-11-19.

[9] Usman Malik. Python for nlp: Working with facebook fasttext library. `https://stackabuse.com/python-for-nlp-working-with-facebook-fasttext-library/`. Accessed 2020-11-15.

[10] Susan Michie and Debra L. Nelson. Barriers women face in information technology careers. pages 10–27, 2006.

[11] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, Omar Fawzi, and Pascal Frossard. Universal adversarial perturbations. pages 86–94, 07 2017.

[12] Stack Overflow. Stack overflow developer survey 2020. `https://insights.stackoverflow.com/survey/2020`. Accessed 2020-11-19.

[13] Ayushi Rastogi, Nachiappan Nagappan, Georgios Gousios, and André van der Hoek. Relationship between geographical location and evaluation of developer contributions in github. In *Proceedings of ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, New York, NY, 2018. ACM.

[14] Josh Terrell, Andrew Kofink, Justin Middleton, Clarissa Rainear, Emerson Murphy-Hill, Chris Parnin, and Jon Stallings. Gender differences and bias in open source: pull request acceptance of women versus men. *PeerJ Computer Science*, 2017.

[15] Philip Thomas, Bruno da Silva, Andrew Barto, Stephen Giguere, Yuriy Brun, and Emma Brunskill. Preventing undesirable behavior of intelligent machines. *Science (New York, N.Y.)*, 366:999–1004, 11 2019.