Coding like a girl: Detecting gender based on code stylometrics

Ian Drosos (izdrosos@ncsu.edu)

Introduction:

Differences in code have been studied for the sake of deanonymizing the programmers behind the code [1]. However their focus was on identifying individual programmers, what if code stylometry can identify the gender of the programmer? Do females programmers actually "code like a girl"? This project represents a first pass at using features extracted from code and using those features to predict the gender of the programmer who wrote it.

The issue at hand is how to predict gender based solely on the code they have written. Features of the code have to be discovered through NLP, parsing the code to create abstract syntax trees and use regex to find various features hypothesized to be relevant to gender prediction. This feature data is then applied with machine learning to predict the author's gender. Do features of the code like cyclomatic complexity, parameter amounts of functions, or even whitespace allow identification of gender? These features are found through code stylometry, or the study of linguistic style applied to code. Stylometry is usually used to analyze written works for variations between author and code stylometry extends this analysis to identifying authors of code. This project seeks to address using code stylometry to identify the gender of authors of code.

The motivation of this project is to research code stylometry as features for prediction. While the data and tests performed in this project were focused on gender, code stylometry could possibly be used in other types of prediction (see Future Work). For this project, the motivation was to create a code stylometric feature extractor that could be later extended (more languages parsed, more attributes predicted). The data obtained from this feature extractor could then be run through machine learning algorithms to test their relevance to gender classification tasks. If this feature extractor produces accurate predictions of gender, it could perhaps be used to find projects that pass the technical version of the "Bechdel test".

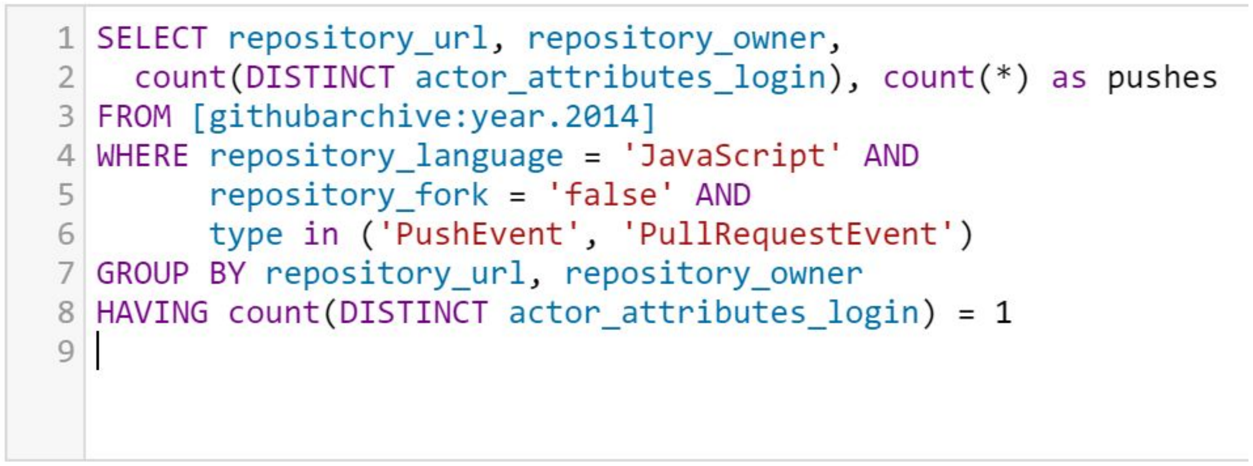
The original Bechdel test looks at a work of fiction (originally a comic strip) and checks whether or not it contains two or more women speaking to each other about something other than a man. This test has been extended to the tech world to see if a piece of code (pulls, commits) or other technical communication (Stack Overflow Q&A, GitHub Issues) features two or more women interacting with each other. For instance a project that passes the technical Bechdel test would have a woman developer pulling original code of another woman and then committing changes to the project. This could be used as a measure of diversity for tech companies and projects, which is an issue frequently talked about recently. The difficulty is in discovering the gender of the author. If the programmer has not identified their gender, how can we tell if the system passes the Bechdel test? If code stylometry features can be used for gender prediction, it could also be used in seeing if the project passes Bechdel test without requiring self identification of the programmers.

The goal of this project is to produce a system that can extract stylometry features from code and use those features to predict the gender of the author who wrote the code. To do this, first a system for extracting features from code must be created. While machine learning algorithms are important to predict classification of the code, the algorithms need features extracted from the parsed code. Success of the algorithms depends on the features presented to them, which is why the parsing system in this project is crucial. After the features are extracted they can then be passed to the algorithms for prediction.

This project can currently only handle feature extraction from JavaScript (.js) files. To increase the amount of code languages covered, the system will need to be extended. The prediction engine could be used for any extension as long as the data conforms to the same schema. The system could also be used to make other interesting predictions if re-configured (see Future work). The system architecture is described below.

System description:

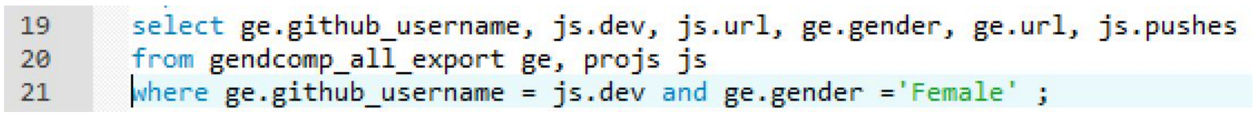
The system implemented had several pieces that needed to be created. First the data needing to be parsed would have to be extracted. The decision was to use JavaScript files since JS is a very popular language that I am familiar with. The code would be taken from GitHub so the first step in data collection was to find an archive of JavaScript projects. Luckily GitHub stores their archives and allows Google BigQuery searches on them. BigQuery allows users to query large datasets through a web interface, processing large amounts of data for the user.



```sql
SELECT repository_url, repository_owner,
  count(DISTINCT actor_attributes_login), count(*) as pushes
FROM [githubarchive:year.2014]
WHERE repository_language = 'JavaScript' AND
      repository_fork = 'false' AND
      type in ('PushEvent', 'PullRequestEvent')
GROUP BY repository_url, repository_owner
HAVING count(DISTINCT actor_attributes_login) = 1
```

Query ran to extract relevant projects for data.

The dataset used was [githubarchive:year.2014], which is all the stored data for the year 2014 on GitHub (pushes, pull request, fork actions, etc). The year 2014 was the last time GitHub stored an entire year in one dataset (they have now moved to temporally shorter datasets, month and day), but the reason I had to use the older 'year' data is because GitHub has also stopped storing language of the project in the newer datasets. The language of the projects is crucial as the parser built and features extracted are relevant only to the language chosen (whitespace in Python is more functional than whitespace in Java which is more about style and readability). Once the parser is extended to be able to handle many languages, the newer datasets can be used without care in what language is chosen. The query above provided over 595,000 repositories that: were JavaScript projects, were not forks, had push events that only had 1 contributor. This data was exported to CSV and added to a database for later joining with a gender table.



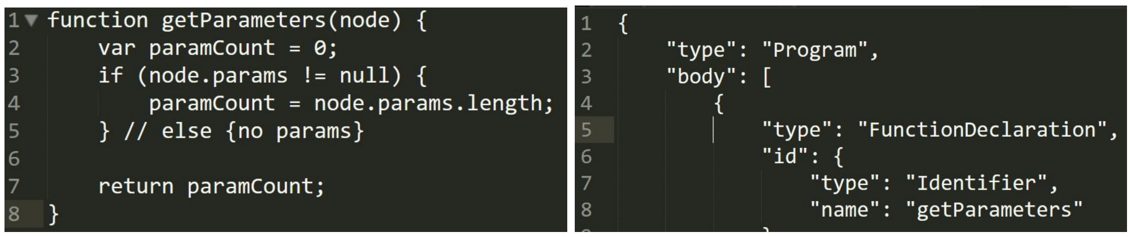
```sql
select ge.github_username, js.dev, js.url, ge.gender, ge.url, js.pushes
from gendcomp_all_export ge, projs js
where ge.github_username = js.dev and ge.gender ='Female' ;
```

Query ran to join tables and find suitable projects for parsing.

The gender table was loaded from data collected by Dr. Parnin's lab that matched GitHub users with their gender by using G+ profiles. G+ profiles require a gender selection of Male, Female, Decline to state, or a custom gender. Only confirmed genders were used to collect files from GitHub users for parsing. This data was added to a database for joining with the GitHub archive. There is a ground truth problem here in that the authors can lie about their gender or identify as a different gender. This is ignored as there is nothing that can be done to prove the author's claims are truthful and the self-reported gender of the author is used for obtaining training data.

The issue of the GitHub data being from 2014 meant that some profiles no longer existed and those that did still exist could have had changes to the project since 2014. Because of these issues the files for parsing had to be downloaded manually from GitHub after checking that the files to be parsed still only had one contributor, the contributor that had their gender confirmed. Another issue that had to be dealt with is that not all JavaScript files are usable or written by the author themselves. Node.js modules are generated by a machine and should be ignored. This is not necessarily the same as code snippets adopted from other sources as when outside code, for example a Stack Overflow snippet solving an issue or algorithm, is brought into a project the programmer usually adapts the snippet to their style. Generated code like node modules are usually never touched by a programmer and hold no stylometric data of use. The use of certain modules and libraries could be used as a feature in prediction in a future extension of the system. Questions like do females use the esprima library more and do males use the fs library more could be answered. With these issues in mind 212 files were pulled for parsing, 106 files from males and 106 files from females. These files were JavaScript code of different programs: websites, servers, games, etc. The files were separated by gender and placed in directories for parsing.

The first step in parsing the files was *analysis.js*. This program was ran twice, once for male files and once for female files. The files were read one by one for parsing. The library esprima was used to create an AST for feature extraction for each file. Esprima takes in the source code of a JS files and creates an AST, for an example see Image 1.



```
function getParameters(node) {
    var paramCount = 0;
    if (node.params != null) {
        paramCount = node.params.length;
    } // else {no params}

    return paramCount;
}
```

```
{
    "type": "Program",
    "body": [
        {
            "type": "FunctionDeclaration",
            "id": {
                "type": "Identifier",
                "name": "getParameters"
```

Image 1: The code on the left produces the tree on the right (partial tree shown).

The 17 features collected by this program are shown and explained in Table 1 and Table 2.These features were taken for each individual JS file by creating an AST and traversing through the tree, collecting data on the existence of various features. Table 1 shows program wide features, these are features taken in the scope of the entire program file. Table 2 shows function wide features. These are features extracted per function inside the program. Both the maximum results from the function and the cumulative results from adding all the features from all the functions in the program are collected and added to the result file. Some features were easily extracted by detecting certain nodes (FunctionDeclaration) or the count of certain nodes (parameters), but some features required algorithms to extract. For instance Max Nesting Depth required traversing deep into the tree until the last decision node and then seeing how many parent decision nodes it had. Another difficult parse was finding the amount of conditions a decision node had, as nested decisions changed the shape of the tree and had to be dealt with. This data is then output to an individual result file for each JS file parsed and stored for another parser that obtains other relevant features (main.py).

Program wide features	
Imports	Number of imports in the program
Functions	Number of functions in the program
Literals	Number of literals in the program
Identifiers	Number of identifiers in the program
Vars	Number of vars in the program
Decisions	Number of “decision” nodes (if statements, for loops, etc) in the program
Size	Size of program in line numbers

Table 1

Features:	Maximum (per function) and Cumulative (all functions) calculated
Cyclomatic Complexity	A measure of software complexity using decision statements
Max Nesting Depth	Maximum depth of scopes (nested decision statements)
Max Conditions	Maximum number of conditions of a single decision statement
Parameters	Number of parameters in a function
Size	Line number size

Table 2

The second step in parsing the files was main.py. This program is ran after analysis.js and appends the feature data it extracts to the result files analysis.js made. The main.py parser uses regex patterns to parse the JS files and find other features (listed in Table 3) to add to the data. These features are generally self explanatory. Lines beginning with '{' means when a line of code starts with an opening curly bracket. This is a difference in style between programmers as some prefer to put the opening curly bracket on the same line of the code that requires it. Other programmers prefer to move the opening curly bracket to the next line to create a noticeable code block under the assumption that it makes the code more readable. Rather than traversing an AST like in analysis.js, this parser reads in the files line by line and extracts feature data. The features are extracted through use of regex. These 57 new feature data points are appended to the individual result files previously created. These files are then to be given to machine learning algorithms for prediction.

Features of the program	
Standard deviation of line length	33 JavaScript keywords count (ECMAScript 6)
Variance of line length	7 Lexical keywords count
Line count	Unique lexical keywords count
Character count	Tab character count
Comment count	Space character count
Ternary symbol count	Empty Line count
Lines beginning with '}'	Whitespace count
(cont next page)	

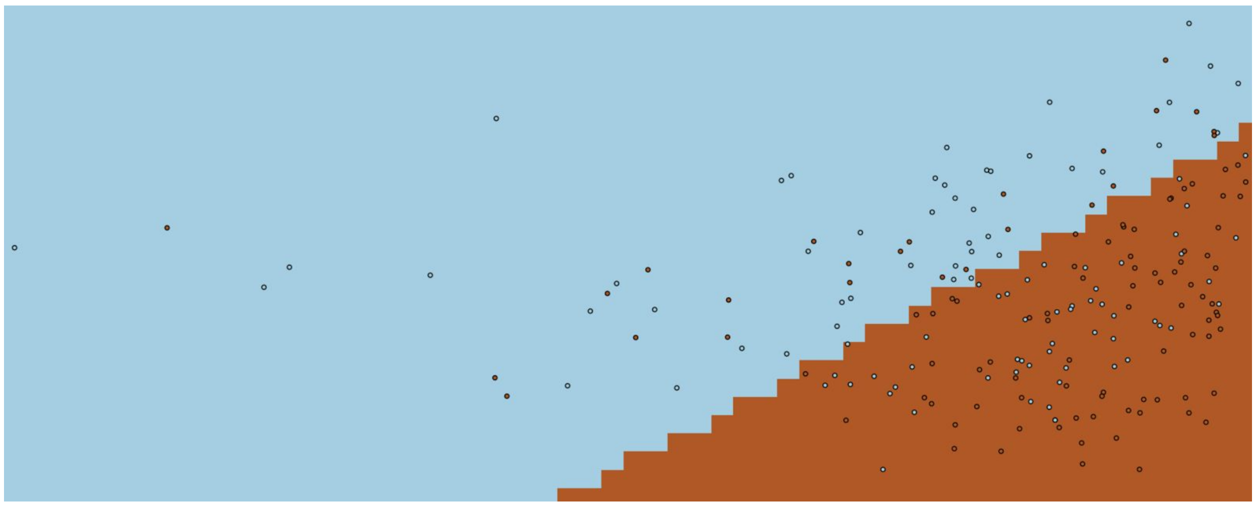
Lines beginning with ‘{‘	Lines beginning with tab character
“Balance of brackets” (difference between the above 2 features)	Underscore ‘_’ count

Table 3

The data was processed and predictions made on that data by science.py. The library scikit learn was used to make predictions and generate graphs. The first thing this program does is merge all of the result files generated from the 2 parse programs. It takes the female result files and prepends a ‘-1’ to the data and then takes the male result files and prepends a ‘1’ to the data. The program writes the data to 1 final result file, science.txt. A NumPy multidimensional array is created using the result file and is used as the dataset being used for prediction, with the first element (-1 or 1) being used as the target and the next 74 elements (parsed features) being used as the data. The data is then split into training and test data for cross validation. The dataset is then ran through several machine learning algorithms to test prediction accuracy (see Results and issues).

Results and issues:

The first thing the prediction program does is produce a graph (Graph 1) using Principal Component Analysis (PCA) and producing a Logistic Regression classifier graph with data points colored according to their labels. PCA does data composition and keeps only the most significant vectors of the data. As seen from the graph, the “most significant vectors” are mostly in their right classification produced by the Logistic Regression classifier but obviously more than the 2 most significant features are needed for accurate classification. Feature selection and engineering is something that can improve prediction accuracy and would be a good next step for extending this system.



Graph 1

The first algorithm ran in science.py is Logistic Regression, a linear model for classification. In this algorithm probabilities describing possible outcomes of a trial are modeled using a logistic function. The first run using Logistic Regression has a cross validation test split on the data of 40% test data 60% training data. The model is fitted to the training data and then a score is received on the test data. The results are: average precision 78%, average recall 76%, and average f1-score of 76%. When this model was fitted and ran 10 consecutive times I received an accuracy of 67% +/- 17% with a maximum accuracy of 80%. This is fairly good accuracy given a first pass at feature extraction. I believe with use of the best features (already discovered or still waiting to be discovered) and more data this accuracy will increase.

Next the dataset is ran through a Gaussian Naive Bayes algorithm with the same test/training split in Logistic Regression. Naive Bayes methods are a set of supervised learning algorithms that apply Bayes' theorem while assuming each pair of features is independent of each other. The model is fitted to the training data and then a score is received on the test data. The results are: average precision 68%, average recall 53%, average f1-score of 43%. Naive Bayes did not perform as well as logistic regression but this could change with more data and better feature sets.

Next the dataset is ran through a few support vector machine algorithms (SVC with linear kernel and a polynomial kernel). The models were fitted to training data and tested on test data. The linear kernel SVC results are when ran 10 consecutive times: accuracy 63% +/- 17% with a maximum accuracy of 76%. The polynomial kernel SVC results are: average accuracy score of 51% +/- 13% with a maximum accuracy of 57%. Polynomial is seemingly no better than just guessing the classification, while the linear kernel performs slightly better than polynomial. So SVMs also did not perform as well as logistic regression but this could change with more data and better feature sets.

Next the dataset is ran through a Nearest Neighbors classifier. Using the same test split as logistic regression it obtains the results: average precision 55%, average recall 55%, average f1-score of 55%. Not a successful algorithm for this data and feature set but this could change with more data and better feature sets.

Finally the dataset is ran through a Decision Tree Classifier using the same test split as before. The results are: average precision 59%, average recall 59%, average f1-score of 59%. Slightly better than nearest neighbors but not as successful as logistic regression. The output of the runs is included with the attached project files for more detailed (and arbitrary) data as sciencepyoutput.txt.

Future work (and Issues):

There are several issues at play in obtaining the best accuracy for gender prediction that require future work to solve. The first is obtaining data. With the necessity of manual file obtainment due to several attributes that disqualify a potential file for inclusion for the parsing, it is currently difficult to obtain the tens of thousands to hundreds of thousands of JavaScript files that could be parsed. Automated scripts could be used to obtain this data if there were ways to deal with disqualifying attributes. More specific data archive columns that describe the language of each file inside the project with the amount of contributors would be a good first step, but this needs GitHub to expose this data to users on BigQuery. Research into common libraries could be used to automatically disqualify code that is generated by libraries like node.js, that way a human does not need to look at the code to make sure files are developed by the human author. Once a better, automatic way to find valid code is found there will be an abundance of data to train algorithms on. Another good source of files are code competitions.

The Caliskan-Islam paper used C++ files from a Google code jam. The benefit of this is all programmers are solving the same problems by implementing similar functionality, leaving style as the major differentiator. This is a great dataset that could be used now that code jams are becoming a popular event for both males and females, as long as organizers of the events take gender information from the contestants. This also prevents style being resultant from problem being solved. A JS file creating a server may stylistically look different than a file creating a game, even with the same author of both. Using a codebase made up of code that does the same thing could remove that variation. Though potential differences in style because of problem difference does not necessarily prevent gender prediction. Perhaps an analysis of programs will show that males have usually code servers while it is rare for a female to do so. The goal of a file could also be a good feature to extract in this case.

Beyond obtaining JavaScript files, the system can be extended to handle many programming languages. The issue is that each individual language will need a separate feature set. Whitespace in Python is functional while whitespace in Java is stylistic, so I believe certain features will not be good data to predict on. Languages differ in keywords, decision statement structures, and some even use different symbols for comments. Different parsers are needed for different languages, each language will need it's own version of analysis.js that creates ASTs to traverse and obtain feature data. Once these new parsers and feature sets are created, programming language no longer limits the amount of files that can be parsed. More data means better predictions, as long as stylometric features continue to be used for accurate classification of gender.

Another issue is with the data itself. How do we know if the data is actually written by the author? Or that the author is actually the gender they state? These are a ground truth problem that is difficult to solve. A future project in analyzing the stylometry inside an individual file could help find code that does not match the code of the author. Each function can be analyzed and compared to other functions inside the code, identifying functions that are different than the rest. Perhaps this could be used as a plagiarism detector or, more positively, a contributor and contribution detector. Using actual push data from GitHub (individual additions to code files) could also give individual contributions of code but does not detect code snippets obtained from other sources (Stack Overflow copy and paste, etc) as stylometric analysis of an individual file and comparison of that data would.

Another issue is obtaining more features. 74 sounds like a lot of features but it is possible that many are not good for prediction (at least, for JavaScript files). More research into what features are relevant and what features can still be extracted from code. Features like use of block comments vs inline comments, acronym usage, and even library usage could be used as more features for the dataset. This will require adding more functionality to the parsers analysis.js and main.py. Beyond obtaining more features, current feature extraction can be refined. It is possible certain features are not perfectly extracted and it limits the features usage in the prediction algorithms. Analysis on features expected to be extracted vs features actually extracted could be useful in rating the successfulness of the parsers.

After the above extensions are made, adding more ways to analyze data and make prediction could be the next step. Neural networks (or now "deep learning") can be used once larger datasets (in features and is samples) are obtained. This could increase training performance and lead to better predictions while showing which features are most important for predicting gender. The addition of larger datasets will allow for algorithms that require more sizable datasets that currently provided by this projects to have more success in prediction.

Beyond gender, there are projects that can be pivoted to with this system. The first was mentioned earlier in this paper, a multiple author detector. This could be used for detecting multiple authors of code and what code each author wrote. This would be useful for automatically obtaining data for this project as GitHub does not currently provide a way to know if there are multiple contributors with current data archives other than visual inspection of the code on GitHub.

Another use for this work would be extending the system to detect code meaning. I am currently working on collecting frustration data from programmers learning python and one of the pieces of data obtained is the source code they are working on that is frustrating them. It would be very useful to be able to detect what the code is trying to do without resorting to visual inspection. Is the code trying to print a fibonacci sequence? Is the code trying to simulate a machine interference problem? Understanding what the code means would allow me to find what tasks cause the most frustration for new programmers. A further extension would be to provide interventions (ways to solve the frustration) to the programmer, but in order to do that I must understand what the problem is that needs to be solved. If the system can parse the code and then predict what the problem is, or at least the category of the problem is, the interventions will be more successful.

Conclusion:

The system described in this paper is a first pass at a stylometric feature extractor of JavaScript files using JavaScript and Python. These features were classified and compiled into a dataset and which was used to produce training and test data for learning algorithms to predict gender. The logistic regression algorithm performed the best with around 78% accuracy. I believe this accuracy can be improved with more samples and more features. Some algorithms perform best when they are provided a large amount of samples, something lacking here due to manual inspection and obtaining of JavaScript files.

In order to provide a larger amount of samples for the dataset, extensions to the system have been proposed in the Future work section. Besides better performance of existing algorithms used, unused approaches could be used once the dataset grows large enough. These new approaches (deep learning) could provide new insight on using code stylometry to predict gender. This why the main focus of the Future work section is obtaining more data samples for the data set and obtaining more features to extract. In order to find more features, each language must be studied further to understand the features it provides the user. Some features may also be newer (Java 8 may include new features not in Java 7) and so new feature adoption may be a feature to look at for extraction. The more data and more instances of data you have the better results your prediction engines will have. That is why I believe the first step must be increasing the breadth of data (more languages parsed), the depth of data (more features extracted), and the amount of data (more files parsed).

Even before the proposed extensions are implemented, the current first pass of using code stylometry has provided good accuracy. The Logistic Regression algorithm provided 78% accuracy in gender prediction and future extensions will improve this accuracy. This result shows that there could be something to the statement “code like a girl” and that code stylometry can be applied to gender detection. Whether or not this is due to gender differences or some other unknown reason is yet to be answered. But the results of this project do give me several paths to take in discovering new methods of predicting gender, extracting stylometric features in code, and parsing code to derive meaning. The next step is to increase dataset size and work on increasing classification accuracy.

Reference:

[1] De-anonymizing Programmers via Code Stylometry: Aylin Caliskan-Islam, Richard Harang, Andrew Liu, Arvind Narayanan, Clare Voss, Fabian Yamaguchi, Rachel Greenstadt.

Source files of the system(see zip file attached in submission):

Parsers: analysis.js, main.py

Dataset analysis with machine learning: science.py

Final dataset: science.txt, a merger and classification of the result files created by the 2 parsers.

Sample output: sciencepyoutput.txt, output from science.py with different results and confusion matrices.

Data files (code files being parsed): not included as the folders are quite sizable with over 200 JS files.