Mining Co-Change Information to Understand when Feature Changes are Necessary in Behavior Driven Development

Aidan Yang
Queen's University
Kingston, Ontario, Canada
a.yang@queensu.ca

Daniel Alencar Da Costa
Queen's University
Kingston, Ontario, Canada
daniel.alencar@queensu.ca

Ying Zou
Queen's University
Kingston, Ontario, Canada
ying.zou@queensu.ca

ABSTRACT

In using Behavior Driven Development (BDD) as a code testing strategy, scenarios that describe the source code's behavior are written in feature files. These feature files change alongside source and test files in a BDD repository. Since BDD is a relatively new testing strategy, developers are often unable to identify code characteristics that need to be modified to accompany added scenarios. This can cause unused scenarios and test failures that slow development progress and reduces the advantages of using BDD. In this paper, we first mine repositories with evidence of using BDD and corresponding non-BDD repositories to explore differences and potential advantages of using BDD. Using fuzzy time clustering, we find that source and test lines of code (LOC) grow much closer to each other, signifying an easier transition between test code and source code writing. We also find that BDD repositories have higher close rate for issue reports and pull requests, but close times are longer. To better understand those differences, we then analyze BDD repositories deeper, with an emphasis on feature file co-changes. Natural language processing is used for identifying feature co-changes, with an accuracy of 79% after reading a sample size of commit messages. We build random forest classifiers using language-agnostic and language-specific code change characteristics to explain when feature co-changes are necessary, achieving an AUC of 0.85. We determine that the most significant characteristics in explaining feature co-change are added test files, modified source files, and modified step definition files. Finally, we employ a frequent item set mining algorithm to give project specific recommendations.

CCS CONCEPTS

• **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability;

KEYWORDS

Behavior Driven Development, Mining Software Repositories

ACM Reference Format:
Aidan Yang, Daniel Alencar Da Costa, and Ying Zou. 2018. Mining Co-Change Information to Understand when Feature Changes are Necessary in Behavior Driven Development. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, Article 4, 7 pages. https://doi.org/10.475/123_4

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 123-4567-24-567/08/06.
https://doi.org/10.475/123_4

1 INTRODUCTION

Engineering software systems is a multidisciplinary process in industries, often involving marketing, accounting and financial teams on top of engineering teams to provide feedback with projects. With industries' growing interest in using ubiquitous language for development to reduce the cost of translation between teams, a new form of testing strategy, behavior driven development (BDD), became popular in recent years. BDD is a software development process emerged from test driven development (TDD). BDD aims to combine business and technical interests by using domain-specific language (DSL) that is similar to English and easy to understand. The principal of BDD is writing scenarios to describe the features and functionalities of development code. These scenarios can be written by all stakeholders of a project due to its simplistic English-like language format.

Projects using BDD as their main form of testing strategy have feature files containing all the project's scenarios. Since feature files are modified along side source files throughout a project's life span, the continual build maintenance generate considerable overhead on development activities. This overhead is worsened as BDD is a relatively new testing strategy and developers have difficulty identifying source code characteristics that need to co-change when scenarios are added, modified or deleted in feature files. To address this difficulty, we compare BDD with non-BDD repositories so as to understand BDD characteristics, and then construct random forest classifiers to evaluate Java language specific characteristics of source and test code changes to understand when feature co-changes are required. After mining open-sourced repositories on Github, and analyzing 133 BDD repositories with 133 similar non-BDD repositories, we address the following four research questions:

(RQ1) What are the differences in how developers create and maintain BDD and non-BDD projects?

We analyze lines of code (LOC) growth evolution for source and test files using fuzzy clustering on the LOC differences between source and test files, and observed that BDD repositories have smaller differences over all. We then mine pull requests and issue reports information from Github's API, and observed that BDD repositories have a higher close and merge rate than non-BDD repositories, but takes longer doing so.

(RQ2) Can we accurately detect co-changes between feature files and production files in a repository?

We link feature files and production files committed within one work week of one another using natural language processing tools and the cosine similarity algorithm. After performing manual analysis, we conclude that we can automatically detect feature co-changes with an accuracy of 79%.

(RQ3) Can we accurately predict when feature co-changes are necessary using code change characteristics?

Yes, our classifiers can explain source and code changes that require accompanying feature changes with an AUC of 0.80.

(RQ4) What are the most significant code change characteristics for explaining feature co-changes?

Using a mean decrease accuracy of AUC approach, we determined that step definition files and LOC deleted and source files deleted are the most influential characteristics for explaining feature co-changes.

1.1 Limitations of exiting work

Prior work has been done on code co-change with regards to build code, or between source and test code based on Test Driven Development, but none on BDD. There are also no work that directly compares BDD repositories with non-BDD repositories. Prior works focusing on BDD only explore evidences of BDD, as in what characteristics clearly constitutes BDD. Other works discussing BDD discuss how BDD can be used more efficiently in specific areas, such as circuit design and verification. Tools to enable this include machine learning and natural language processing. However, no BDD related works use Mining Software Repository (MSR) tools to analyze large amounts of repositories and explore how BDD affects other files, programs, and code lines in a repository.

This work compares BDD with non-BDD from open source repositories and identifies advantages and disadvantages of using BDD. We find and evaluate a method to link feature files to production files in a BDD repository, as well evaluate characteristics that explain feature co-changes, so as to give recommendations for BDD developers when writing and maintaining both feature and production code.

1.2 Related work

1.
   2.
   3.
   4.

1.3 Paper organization

The remainder of the paper is organized as follows. Section 2 outlines the experiment setup, Section 3 present the results with respect to our four research questions, Section 4 discusses threats to the validity of our results, and Section 5 draws conclusions to the paper.

2 EXPERIMENT SETUP

All data are collected from Github open source repositories. After BDD repositories are collected, characteristics of those repositories are recorded: language, size of repository, popularity (amount of stars on Github), and number of authors. Using those characteristics, non-BDD repositories with similar characteristics are collected for comparison.

2.1 Methodology for Data Collection

We use Jersey's json parser (a Java extension) to crawl through Github's search API to find repositories that mainly uses java. This is done by Github's search API and calculating the majority language. We start with one million repositories in total, and are left with 59933 repositories that mainly uses java. We then use Github's tree API to find repositories that include at least one .feature file, which we deem as a BDD repository. We find that 927 out of 59933 uses BDD. BDD repositories that include only one commit that uses BDD are eliminated, as that could be a developer simply trying BDD for a short time and then stopping, and the utilization of BDD in that one commit does not propagate to the entire repository. Furthermore, BDD repositories that do not have both *log* and *name-stat* data in *UTF-8* format are eliminated.

In the end, 133 BDD repositories remained. 133 non-BDD repositories with similar characteristics were then cloned.

2.2 Methodology for Data Analysis

To compare BDD with non-BDD repositories (RQ1), we analyze LOC growth, pull requests, and issue reports. For LOC growth, we use a time clustering algorithm to identify trends. To compare pull requests and issue report stabilization times, we use wilcox rank sum test and cliff delta test. To establish feature co-changes (RQ2), we use Stanford NLP to parse feature LOC and Cosine Similarity for production file linking. To find characteristics that explain feature co-changes, we construct classifiers using random forest technique, and evaluate those classifiers by calculating the Area Under the Curve (AUC). Finally, we find the mean decrease in accuracy with respect to AUC for each characteristic to determine the most significant ones in predicting feature co-change.

3 RESULTS

3.1 RQ1

What are the differences in how developers create and maintain BDD and non-BDD projects? We want to find and compare differences, if any, between BDD and non-BDD repositories. If there are significant differences, we are more motivated to find characteristics that explain feature co-changes.

3.1.1 Approach. We first explore LOC growth and analyze how LOC from source and test files interact with one another. We then mine pull requests and issue reports from Github's API. The collected information is used to analyze percentages of merges and fixes, and time to stabilize requests.

3.1.2 LOC initial observation. Figure 1 shows two similar repositories, with repository *apodhrad* using BDD and *lukasz* not using BDD. We see that the difference between source and test LOC on the BDD repository is much lower than its non-BDD counterpart. To better understand this difference, we move onto to another LOC analysis involving all studied repositories.

3.1.3 Source LOC and test LOC difference growth. Figure 1 motivates a closer look at differences between source LOC and test LOC. The differences within all 133 repositories are recorded using simple subtraction: $Difference = SourceLOC - TestLOC$. We then use time series clustering to identify difference trends. Time

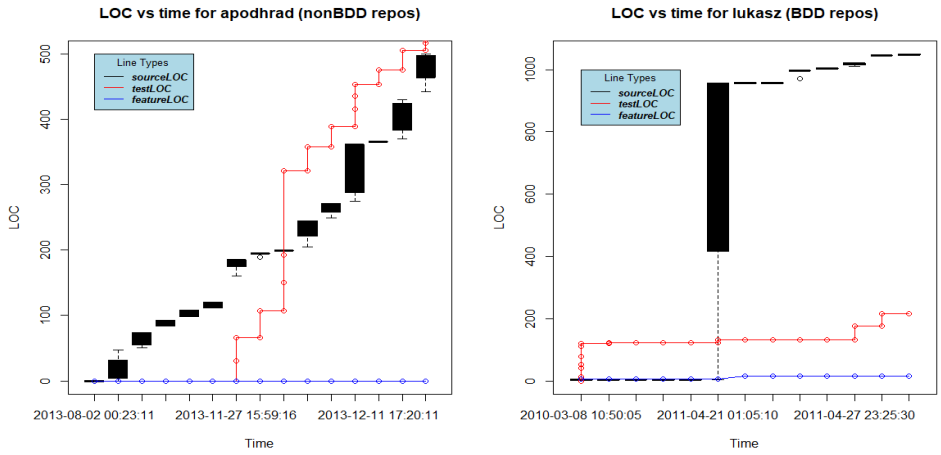

Figure 1: BDD vs non-BDD

Table 1: Pull requests data

Attribute	BDD	Non-BDD
% Repositories with pull requests	12.8%	13.5%
Total pull requests	132	106
Merged	117	88
Closed	4	15
Open	11	3
Merge rate	88.6%	83%

Table 2: Issue reports data

Attribute	BDD	Non-BDD
% Repositories with issue reports	20.3%	21.8%
Total pull requests	252	195
Closed	208	136
Open	44	59
Close rate	82.5%	69.7%

series clustering requires 1) a clustering algorithm, 2) a distance measurement method, and 3) number of clusters. We apply fuzzy clustering algorithm with dynamic time warping (DTW), and the gap statistic approach to estimate the optimum number of clusters.

By using gap statistics approach, we find that 18 is the optimum number of clusters to use.

As seen in figure 2, BDD repositories have 5 clusters in which the difference is mostly stable, and is not significantly increasing: 2, 12, 14, 17, 18. The LOC differences in the other 13 clusters are predominantly growing. In clusters 2, 17 and 18, the difference are at times below zero, meaning that the test LOC are higher than the source LOC.

For non-BDD repositories, all 18 clusters are predominantly growing, with higher slope than BDD clusters as well, and at no point does the difference reach below zero.

We can see that in BDD repositories, developers tend to match the amount of test code closer with the amount of source code, and sometimes write test code before source code.

3.1.4 Pull requests and issue reports. We can see from Table 1 that BDD repositories have a higher merge rate for pull requests, and a higher fix rate for issue reports. However, from observing Figure 3, both pull requests and issue reports take longer times to merge pull requests and close issue reports in BDD repositories. From the previous analysis on LOC, a reason for that maybe the additional amount of test LOC to stabilize in BDD repositories.

To find the statistical differences between BDD and non-BDD pull requests merge times, we use Wilxocon Rank Sum Test to find a Continuity Correction p value of 3.25e-06. Furthermore, we find a Cliff Delta Estimate of 0.3884.

Similarly, we perform the same analysis on issue report close times and find a Wilcox p value of 2.26e-07 and a Cliff Delta Estimate of 0.3954.

Due to low p value from wilcox test (<0.05), and medium delta estimate (>0.35) for both pull requests and issue report times, there is a difference between how developers maintain BDD repositories and non-BDD repositories time wise.

3.2 RQ2

Can we accurately predict co-changes between .feature files and other files in a repository?

We want to explore how feature files and source/test files are written and maintained together, and the first step in that process is identifying co-changes.

3.2.1 Approach. We use Stanford's NLP to categorize all words in .feature files into word type brackets. Words that do not fall into the desired brackets (nouns and verbs) are eliminated and not used for comparison. To parse development files, we use a java keyword parser to identify and eliminate java language specific words and phrases. The remaining words and phrases are user specific and kept for comparison. We use cosine similarity algorithm to compare LOC changed in feature files and development files. Finally, we manually

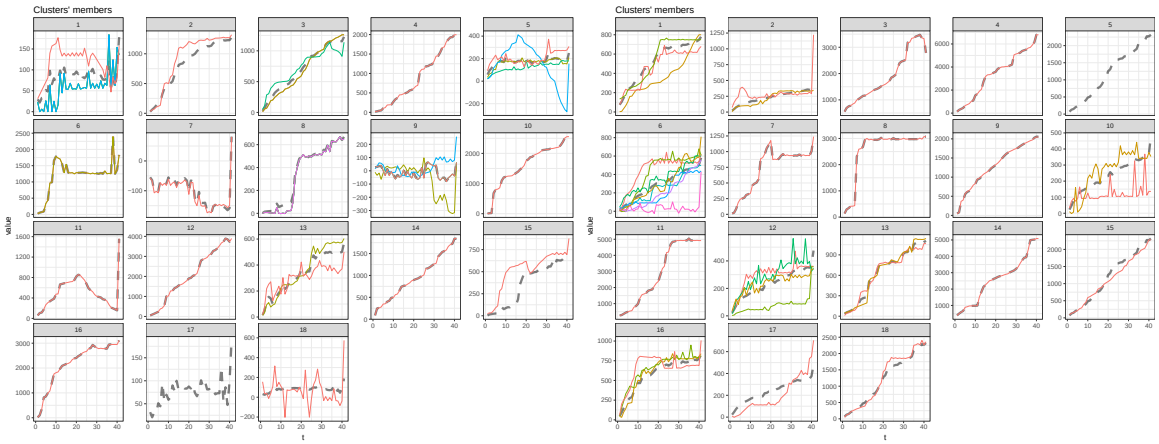

Figure 2: LOC difference growth clusters for 133 BDD repositories (left) and 133 non-BDD repositories (right)

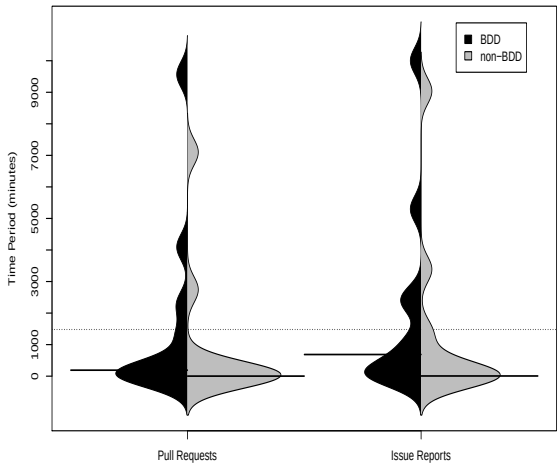

Figure 3: Merge and close times comparison

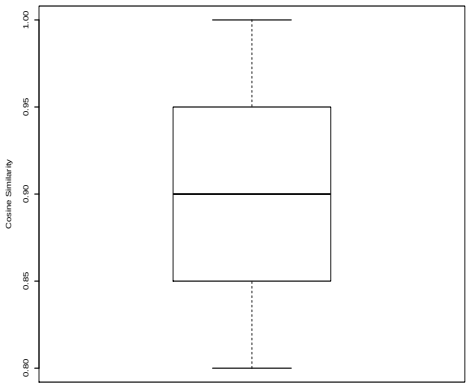

Figure 4: Cosine similarity distribution for all identified links

examine git logs and commit messages to determine if cross commit links are actual links.

3.2.2 Results. All feature files within a commit are linked to at least one production or test file in the same commit. For files that are linked across different commits, we took the upper 75% quantile of all link similarity outputs (between 0 and 1) as a minimum threshold for cross commit links. As seen by Figure 4, this threshold was 0.95. Time difference between commits was determined to be one work week as a maximum threshold to accept cross commit links. This is 2400 minutes.

To evaluate the detected links, we read developer's commit messages to understand the functionality of each commit.

Example of two commits logs in repository *Chorus-bdd-Chorus*:

Commit id 812df log: Start to add extra remoting tests. Commit id 063ca log: Add a test for remoting to locally launched process.

063ca is committed 79 minutes after 812df, but we can see the two commits are related.

Of 451 cross commit work items across 133 BDD repositories, we conclude that 360 work items are actually linked together, and the rest were unable to be fully identified. This is an overall 79.8% accuracy.

3.3 RQ3

Can we accurately explain when feature co-changes are necessary using code change characteristics?

If we can find code characteristics that accurately predict co-changes, we can narrow down the most important characteristics that cause those differences.

Table 3: Example confusion matrix

	Classified as	
Actual category	Change	No change
Change	a	b
No change	c	d

3.3.1 Approach. Both language agnostic and java specific characteristics were used as independent variables in predicting whether a work item involves a feature co-change (signifying the use of BDD). We use random forest technique to construct classifiers that explain when feature co-changes are necessary. The random forest technique constructs a large number of decision trees at training time. Each node in a decision tree is split using a random subset of all of the attributes. Performing this random split ensure that all of the trees have a low correlation between them [L. Breiman Random Forest]. To further ensure low correlation, we performed a correlation redundancy test on our variables. After a large number of trees is generated, they vote for the most popular class. To evaluate the performance of classifiers, we construct classifiers using a testing corpus and compare its deduction against known result. We use tenfold cross-validation to obtain the testing corpus, which is splitting the data into ten equal parts, and taking one part at random as the testing corpus while the other nine as the training corpus. This process is repeated ten times, using a different part as the training corpus each time. We use Area Under the Curve (AUC) to evaluate the performance of the decision tree. AUC is the area under the plot of true positive rate ($\frac{a}{a+b}$). against false positive rate ($\frac{c}{c+d}$).

3.3.2 Results. Using all variables, we achieve an AUC of 0.8630. However, some variables might not be independent of the others. Since all variables are numeric values, we use Pearson correlation test, with a threshold of $r^2 = 0.7$. In Figure 5, the horizontal line shows the threshold from which the variables are highly correlated with each other. We can see that files deleted and other LOC deleted are correlated with each other, and source files modified and test files modified are correlated with each other. We eliminated other LOC deleted and test files modified for the random forest classification process.

AUC after eliminating two variables was 0.849.We conclude that both language agnostic and language specific characteristics can predict feature co-change, with minimal redundancies.

3.4 RQ4

What are the most significant code change characteristics for explaining feature co-changes?

We address RQ4 to answer the paper's central question on if we can fully explain feature co-changes. We do this by predicting the most important code changes developers need to make when changing feature files.

3.4.1 Approach. To study the most influential code change characteristics in the random forest classifiers, we calculate the mean decrease accuracy with respect to AUC of each characteristic.

3.4.2 Result. The four most significant predictors for BDD work items are as follows: addition of step definition files, modification of

Table 4: Taxonomy

Attribute Definition	Type Rationale
Name Number of source files added in a commit	Numeric Changing a source file modify ne
Source file added Number of test files added in a commit	Numeric functionalities that need to be tes
Test file added Number of other files added in a commit	Numeric and test the new scenarios
Other file added Number of step definition files added in	Numeric Newly defined feature scenarios
Step definition file added a commit	Numeric test file to initialize
Source file modified Number of source files modified in a	Numeric All other files changed could also
Test file modified commit	Numeric the structure of the repository, w
Other file modified Number of test files modified in a commit	Numeric changes for testing
Step definition file modified Number of other files modified in a	Numeric Step defintion files are required f
Source file deleted commit	Numeric tests to run, so all changes to step
Test file deleted Number of step definition files modified	Numeric accompanying feature changes
Other file deleted in a commit	Numeric Same as source file added
Step definition file deleted Number of source files deleted in a	Numeric Same as test file added
Source file renamed commit	Numeric Same as other file added
Test file renamed Number of test files deleted in a commit	Numeric Same as step definition file added
Other file renamed Number of other files deleted in a commit	Numeric Same as source file added
Step definition file renamed Number of step definition files deleted	Numeric Same as test file added
Source LOC added in a commit	Numeric Same as other file added
Test LOC added Number of source files renamed in a	Numeric Same as step definition file added
Other LOC added commit	Numeric Same as source file added

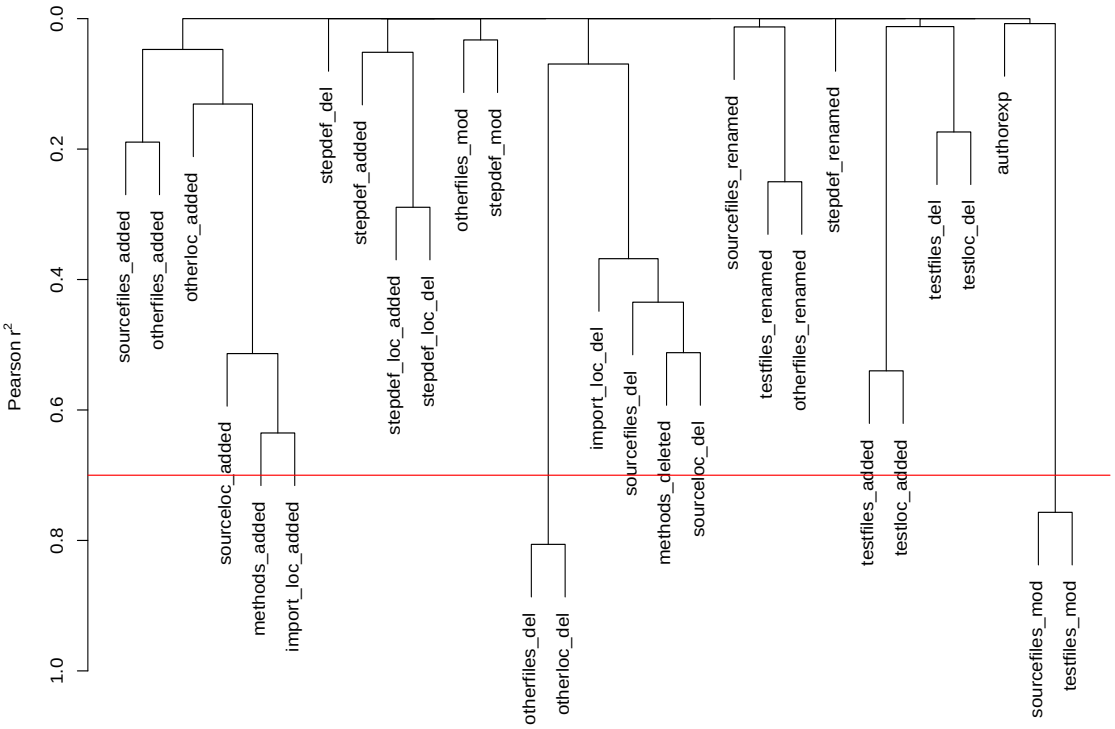

Figure 5: Hierarchical clustering of code characteristics

source files, modification of step definition files, and modification of all other files.

4 THREATS TO VALIDITY

4.1 Construct validity

All repositories collected and analyzed are open source Github repositories, which may cause threats to validity. Some projects have few commits, some projects are already inactive, some could be personal and not for software development, and not all repositories use pull requests and issue reports, and the ones who do could have states declared as "open" or "non-merged" but might actually be closed already. Furthermore, when a pull-request is merged Github records commits that are the result of the peer-review, and not the original commits. This causes errors in RQ2's manual analysis as commit logs sometimes misrepresent the actual function of the commit

4.2 Internal validity

In parsing java production LOC in RQ2, Java keywords used were keywords for the most popular java development toolkits. Some toolkits that are not widely used could employ keywords that were not filtered out, and feature co-changes were identified based on those keywords and not user specific code.

Moreover, many commits have a low amount of commit messages, and some do not explain the functionality of the commit precisely. This makes manual analysis in RQ2 sometimes arbitrary.

In RQ3, characteristics used for random forest classification were not selected with concrete, quantitative analysis. Some characteristics that we have overlooked may have improved the performance of classifiers.

4.3 External validity

We only use 133 repositories for both BDD and non-BDD in our analysis. further studies should investigate more repositories to avoid any sample bias, preferably large industry repositories.

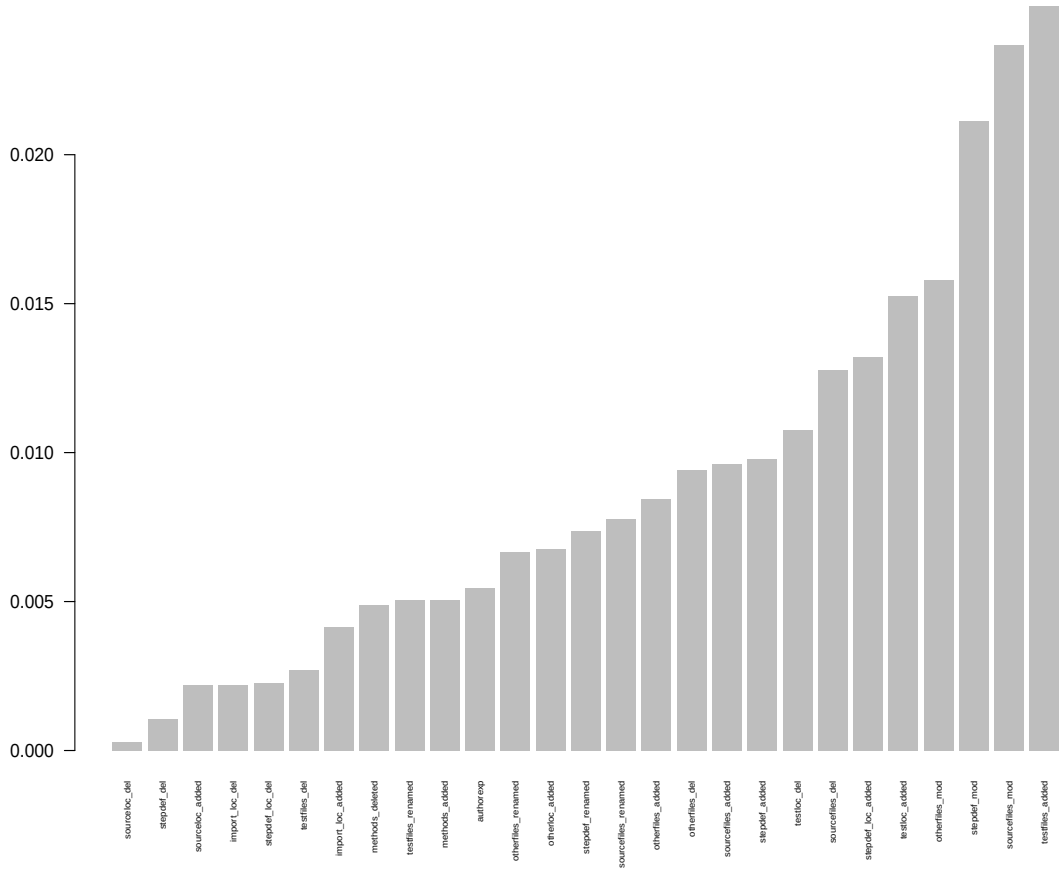

Figure 6: Most influential code change characteristics by mean decrease AUC

5 CONCLUSIONS

BDD is a relatively new testing strategy that uses English-like syntax in describing code functionalities. BDD makes it easier for all stakeholders involved in a project to understand the features of a repository. With the use of feature files in describing the scenarios of development files, the co-evolution of feature files with source and test files can cause code breakage or slow development process if developers are not certain of code characteristics that must accompany feature file changes. We set out to compare BDD and non-BDD repositories, and to find code characteristics that can accurately predict feature co-changes.

We identified multiple differences between BDD and non-BDD repositories. We first observed that BDD repositories have a lower source and test code growth difference, signifying a smoother transition between writing source code and test code. We also observe that BDD repositories have higher merge and close rate for pull requests and issue reports, but takes longer doing so.

We then use natural language processing to link feature files with development files, with an accuracy of 79%. Using those links, we found twenty seven independent code characteristics that can predict feature co-changes with an AUC of 0.85. By observing the mean decrease AUC of each characteristic, we found that added test files, modified source files, and modified step definition files are the best predictors for feature file co-change. In conclusion, we can predict that a feature file must be modified when step definition files within test folders are added, and step definition files within source folders are modified.

5.1 Future work

Our conclusions show that there are differences between BDD and non-BDD repositories. We plan on exploring the reasons behind those differences in more detail. We also plan on building project specific recommendation systems to give developers a better understanding of feature co-changes.