Optimizing Code Smell Detection Techniques

Jaymin Desai, Timothy Dement, Dyuti De, Matt Leader
North Carolina State University
{jddesai2,tmdement,dde,mfleader}@ncsu.edu

ABSTRACT

Detecting code smells using machine learning techniques is gaining popularity as it does not require the cognitive skills of developers nor the subjective bias of the code smell detectors. We analyzed the current research in this domain and replicated realistic code smells datasets with a low density of code smell instances [15]. To optimize classifier performance we used feature selection methods to reduce the feature space, we used SMOTE to oversample the minority of true code smell instances, and we applied K-Means clustering to the datasets. Finally, we targeted the "Simple," "Useful," and "Available" baselines for adequate AI with our research. Our research shows that further gains in classifier performance on small code smell datasets can be achieved with SMOTE (Synthetic Minority Oversampling Technique) and feature selection.

KEYWORDS

Code Smells, Machine Learning Techniques, Feature Selection, SMOTE, Scott-Knott, Classifiers, Clustering, Small Data, Oversampling

1 INTRODUCTION

A code smell is a surface indication that usually corresponds to a deeper problem in the system [7]. It is a very common occurrence for software engineers to be under constant pressure to roll out working deliverables within a short period of time [11], as is, especially, the case with software development teams that employ the Agile paradigm. Understanding and detecting code smells makes the work of re-factoring code much faster and easier for the software developer. The type of code smell also helps developers ascertain the type of re-factoring required. Thus detection of code smells has become a crucial business goal and research topic in the modern era.

Detection of code smells has been approached in several ways. Various solutions have been employed such as code inspection [18], metric or rule based strategies [10], static code analysis [19], etc. However, most of these detection techniques are subjective to the developers' bias [5] and do not provide consistent results nor agree with one another [6].

2 RELATED WORK

In 2016, Arcelli Fontana, et al. [1] sought to address the aforementioned problems with code smell detection techniques by employing machine learning algorithms. The researches observed a lack of formal definitions for code smells, and thus identified and defined four code smells that seemed to be the most prevalent and have the highest negative impact on the observed projects:

- **Data Class:** Refers to classes that store data without providing complex functionality and having other classes strongly relying on them.
- **God Class:** Refers to classes that tend to centralize the intelligence of the system.
- **Feature Envy:** Refers to methods that use much more data from other classes than from their own class.
- **Long Method:** Refers to methods that tend to centralize the functionality of a class.

The researchers then carried out a large-scale case study to determine the effectiveness of different machine learning algorithms in detecting the four identified code smell types. They reported high and stable scores across the Accuracy, F-Score, and AUC-ROC (Area Under Curve - Receiver Operator Curve: true positive versus false positive rate) for the majority of the 32 algorithms they applied, with the Accuracy and F-Score frequently exceeding 95%, and their results suggest that classifiers from the Decision Tree and Random Forest families generally produced the best Accuracy across the different code smell datasets [1].

In 2018, however, Nucci, et al. [15] sought to replicate the results of the previous study, and in the process discovered shortcomings in the construction of the datasets and experimental process that lead to the unrealistically high performance results obtained:

- **Selection of the instances in the dataset:** Within each code smell set, the authors found that the distribution of the independent variables of the smelly and non-smelly instances were effectively disjoint, allowing any machine learning technique to easily classify new instances.
- **Unrealistically balanced dataset:** The original datasets contained a relatively balanced frequency of smelly and non-smelly instances, which is not representative of software engineering code bases in practice.
- **Biased validation methods:** Similar to the previous point, the original study applied ten-fold cross validation on artificially balanced datasets, potentially introducing bias to the training and testing sets.
- **Missing analysis of relevant features:** The authors note the inclusion of highly-correlated independent variables in the dataset can lead to over-fitting of the machine learning models and biased results.

To amend these shortcomings, the authors created four new datasets based on the original datasets of the reference study. They merged the two class-level datasets, Data Class and God Class, and made two copies of the merged dataset. In one, only the smelly instances for the Data Class smell were set to true; in the other, only the smelly instances for the God Class smell were set to true. This merging process was repeated for the two method-level datasets, Feature Envy and Long Method. The result was four datasets that 1) had a smaller difference in the distribution of the independent variables, 2) were less balanced between smelly and non-smelly

Foundations of Software Science, Fall 2018, Computer Science

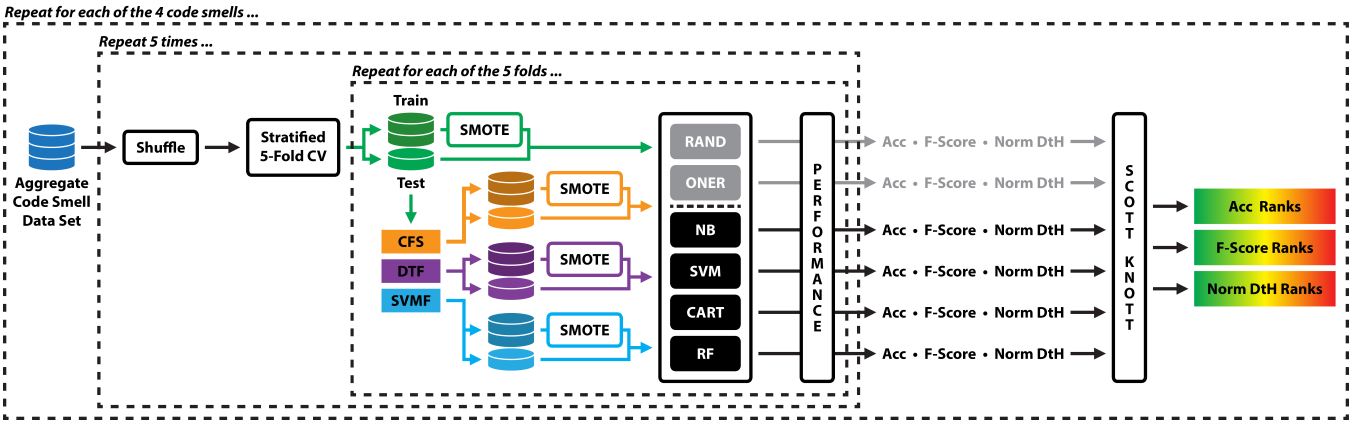

Figure 1: Empirical Study Process Overview

instances, and 3) were a more realistic representation of software engineering code bases in practice [15].

When the same 32 machine learning algorithms were applied to the new datasets, Nucci, et al. observed that overall the Accuracy scores did drop, but were still relatively high (on average, 76% versus the 96% reported in Arcelli Fontana, et al). The F-Scores observed, however, were 90% lower than those of the reference study, "indicating that the models were not actually able to properly classify the smelliness of the analyzed code elements" [15]. Similarly, the AUC-ROC (true positive rate versus false positive rate) scores showed a high variance between different classifiers. Although their results seemed to suggest that classifiers from the Decision Tree and Random Forest families generally performed better on the code smell datasets, the authors concluded that the appropriateness of different classifiers for code smell detection is still an open question.

3 RESEARCH QUESTIONS

In this paper, we expanded upon similar machine learning approaches applied in the two aforementioned studies, and pose the following four research questions:

- **RQ1:** Do the families of classifiers available in Python's Scikit-Learn library produce comparable results to those given by Nucci, et al.?
- **RQ2:** Does oversampling using SMOTE improve the performance of the studied classifiers for code smell detection?
- **RQ3:** Does applying feature selection improve the performance of the studied classifiers for code smell detection?
- **RQ4:** Does clustering the data help reduce the computation and/or improve the performance of the studied classifiers for code smell detection?

4 EXPERIMENTAL APPROACH

4.1 Dataset Overview

Our empirical study was performed on the two class-level code smell datasets, Data Class and God Class, and the two method-level code smell datasets, Feature Envy and Long Method, as constructed by Nucci, et al. [15]:

Dataset	Features	Rows	Smell Frequency
Data Class	61	289	18%
God Class	61	289	26%
Feature Envy	82	234	24%
Long Method	82	234	20%

The class label (smellly or non-smelly) was originally determined by Arcelli, et al. [1] using a combination of automatic code smell detection tools and manual validation.

4.2 Process Overview for RQ1 - RQ3

Having constructed the four code smell datasets according to the method proposed by Nucci et al. [15], we performed our empirical study as outlined in **Figure 1**. For each code smell dataset, we first shuffled the given dataset five times. Since the frequency of smelly instances in our datasets decreased from 38-48% to 18-26%, we applied stratified cross-validation to determine five training and testing folds. The five shuffles and the five folds gave us a total of twenty-five unique runs and result sets for the experimental pipeline.

Within a given fold, we applied three feature selection algorithms to the training data and created three new training datasets based on the features selected. We used the Correlation-Based Feature Selection (CFS), Decision Tree Forward-Selection Wrapper (DTF), and Support Vector Machine Forward-Selection Wrapper (SVMF) implemented by Cheng, et al. in the Scikit-Feature Python library [13] to evaluate **RQ3**. The CFS algorithm automatically selected the number of features to select, and on average selected approximately 9 features out of the 61-82 available. For the DTF and SVMF algorithms, we set the number of features to select at 7 for the class-level datasets and 9 for the method-level datasets (approximately the square root of the number of features available).

After having obtained four training datasets (the original training dataset, along with the three training datasets produced by CFS, DTF, and SVMF), we also applied SMOTE (Synthetic Minority Oversampling Technique) using the Imbalanced-Learn Python library [12] to each training dataset, giving us a total of eight training variations with which to evaluate **RQ2**:

- The original training data, with and without SMOTE
- The CFS training data, with and without SMOTE
- The DTF training data, with and without SMOTE
- The SVMF training data, with and without SMOTE

As noted in Swagatam, et al. [3], when a dataset exhibits a high Imbalance Ratio (majority class to minority class), overlap between the classes, and has a small quantity of examples, then model training, and subsequent prediction, can be impacted by class imbalance. For each code smell dataset, there is a small minority of smelly instances. Preliminary results showed the majority of our classifiers misclassified the smelly instances in the minority class. Oversampling, specifically the SMOTE algorithm as implemented in Guillaume, et al. [12], is used to decrease the effect of class imbalance on classifier learning and prediction in our experiment.

We then trained two sanity-check classifiers, a Random Guess classifier (RAND) and One-Rule classifier (ONER) on the original training data (with and without SMOTE), and trained four additional classifiers from different families — Naive Bayes (NB), Support Vector Macine (SVM), Classification and Regression Tree (CART), and Random Forest (RF) — on each of the eight training variations listed above in order to evaluate **RQ1** and **RQ3**. We implemented these using the Scikit-Learn Python library [16]. To limit the scope of the study, we did not apply any hyperparameter optimization techniques to these classifiers, other than to increase the number of trees used in RF from the default of 10 to 100.

We tested each of the classifiers on the previously set-aside testing data for the fold, and recorded the Accuracy (ACC), F-Score (F1), and Normalized Distance to Heaven (ND2H) performance metrics for each classifier across all of the twenty-five unique runs. ACC and F1 were chosen from their inclusion in the Arcelli Fontana, et al. [1] and Nucci, et al. [15] studies, and ND2H is proposed as a more intuitive alternative to the use of AUC-ROC in both papers. Note that for reporting purposes, 1 - ND2H is often given in order to facilitate trend comparison with ACC and F1.

$$ACC = \frac{TP + TN}{TP + TN + FP + FN} \tag{1}$$

$$F1 = \frac{2TP}{2TP + FP + FN} \tag{2}$$

$$ND2H = \frac{\sqrt{FPR^2 + FNR^2}}{\sqrt{2}} \tag{3}$$

Finally, we applied the Scott-Knott ranking algorithm recommended by Mittas, et al. [14] for each of the reported performance metrics, ranking all combinations of classifiers and feature selection techniques. This was done separately on the results using SMOTE and without using SMOTE.

4.3 Process Overview for RQ4

After analyzing the performance of the model with and without SMOTE, we found that SMOTE improved the performance (discussed in Section 5.2) and decided to experiment with Unsupervised Clustering on over-sampled data. The idea was to cluster the data and classify within the cluster bounds. Note: We excluded Feature selection for this experiment.

We started with K-means clustering - one of the most simple clustering algorithms. To determine the number of clusters we used the elbow method. We plotted values of K as the horizontal axis and the sum of squared distances of samples to their closest cluster center as the vertical axis. From this plot, we chose the optimal number of clusters to be the value of K that forms the elbow arc. To determine the initial centroids, we used the K-means++ Initial Seeding [2] method.

The approach that we employed is as follows:

- Separate the independent variables and train the K-means classifier to determine the clusters for training set instances. Determine the clusters labels for testing set instances using the trained K-means classifier.
- Segregate the training and testing instances cluster-wise.
- Within each cluster, predict the class label of test instances using a machine learning classifier trained from the training instances belonging to the same cluster.
- Aggregate the ground truth and predictions for every cluster and compute the performance metrics.

The other setup remains the same as described in Section 4.2 where we perform stratified 5-fold cross-validation, 5 times and predict the class labels using ONER, Naive Bayes, Support Vector Machine, CART and Random Forest.

The motivation was to improve the overall turn-around time for computation-heavy machine learning classifiers by clustering the data first. If the distribution of class labels within a particular cluster is 90:10 or better (majority class label is at least 90% and in best case, 100% which implies all the instances in the cluster have same class label), we predict the class of all the test instances belonging to that cluster to be the majority class label of that cluster. Hence, no further classification is required which eventually saves computation.

5 RESULTS

5.1 RQ1 Results

> ***RQ1:*** *Do the families of classifiers available in Python's Scikit-Learn library produce comparable results to those given by Nucci, et al.?*

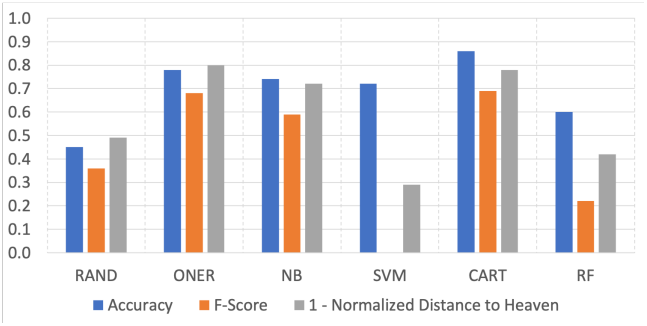

Figure 2: RQ1 - Comparison of classifier performance metrics for the God Class set.

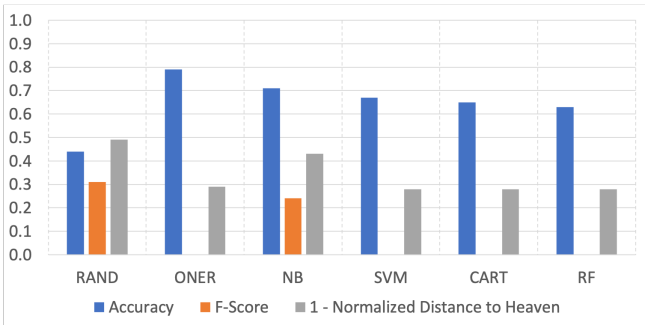

Figure 3: RQ1 - Comparison of classifier performance metrics for the Long Method set.

The median classifier scores across the 25 runs for the God Class and Long Method datasets are given for Accuracy, F-Score, and 1 - Normalized Distance to Heaven in **Figures 2 and 3** (the figures for the Data Class and Feature Envy sets follow similar patterns, and are included in the appendix). These charts include the sanity-check classifiers (RAND and ONER), as well as the default classifiers (NB, SVM, CART, RF).

The variation between the different metrics is immediately apparent from the figures, and clearly demonstrates that the relatively high ACC values are generally poor indicators of the performance of the classifiers. This is particularly evident for the SVM classifier across the four code smell datasets; while its median ACC scores range from 0.59 (Feature Envy) to 0.81 (Data Class), its median F1 scores remain at 0.00, and its median 1-ND2H scores range from 0.27 (Feature Envy) to 0.29 (Data Class and God Class). This reinforces the argument from Nucci, et al. [15] that despite the relatively high ACC scores reported from a variety of machine learning algorithms, they actually have little predictive value.

In addition, even when discounting the rankings based on ACC performance, we observe a significant variation in the best-performing classifiers from dataset to dataset. In the Data Class set, CART is the best-ranked F-Score classifier with a median score of 0.36, and CART and NB are the best-ranked 1-ND2H classifiers with median scores of 0.56 and 0.58. In the God Class set, CART is the best-ranked classifier for F1 and 1-ND2H, with median scores of 0.69 and 0.78 respectively. In both the Feature Envy and Long Method sets, however, NB is the best-ranked classifier for both F1 and 1-ND2H, with median F1 scores of 0.48 and 0.24, and median 1-ND2H scores of 0.61 and 0.43.

The above discussion of best-ranked classifiers should, however, be strongly informed by the performance of the sanity-check classifiers RAND and ONER. As can be observed in **Figures 2 and 3**, the default classifiers frequently do not perform any better than random guessing or one-rule classification, regardless of the specific performance metric or dataset. These last two points reinforce the argument from Nucci, et al. [15] that, counter to the claims made by Arcelli Fontana, et al. [1], the appropriate application of machine learning techniques to code smell detection is still very much an open question. This is further discussed in the Threats to Validity section.

5.2 RQ2 Results

> ***RQ2:*** *Does oversampling using SMOTE improve the performance of the studied classifiers for code smell detection?*

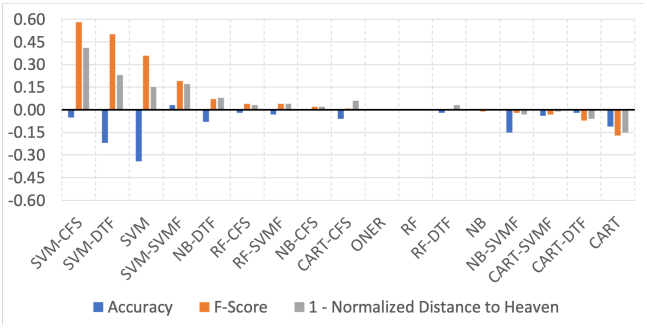

Figure 4: RQ2 - Change in classifier performance metrics when using SMOTE for the God Class set.

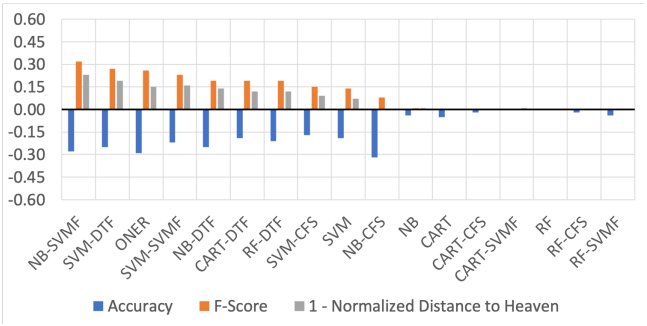

Figure 5: RQ2 - Change in classifier performance metrics when using SMOTE for the Long Method set.

The change in the median classifier scores across the 25 runs for the God Class and Long Method datasets are given for ACC, F1, and 1-ND2H in **Figures 4 and 5** (the figures for the Data Class and Feature Envy sets follow similar patterns, and are included in the appendix). These charts also include the sanity-check classifiers RAND and ONER, as well as the default classifiers (NB, SVM, CART, RF) and their feature-selection variations (CFS, DTF, SVMF). For a detailed view of the Scott-Knott rankings, please see the appendix.

Across the 68 classifier instances (the 17 classifier variations run on each of the four code smell datasets), the largest F1 improvement was gained by the SVM-CFS classifier in the Data Class set, changing from an F1 score of 0.00 (Scott-Knott Rank 6 of 6) to 0.58 (Scott-Knott Rank 2 of 5). SMOTE improved the F1 scores of 72% of the classifier instances, had no effect on 18%, and decreased the F1 scores for 10% of the classifiers. These decreases were relatively minimal, however. In the worst case, the CART classifier in the God Class dataset saw an F1 decrease from 0.69 (Scott-Knott Rank 1 of 6) to 0.52 (Scott-Knott Rank 3 of 5). The average and median improvements to F1 scores across all classifier instances were by 0.16 and 0.12, respectively, whereas the average and median decreases to F1 scores were by -0.05 and -0.02, respectively.

The improvements for 1-ND2H were similarly promising. Across the 68 classifier instances, the largest 1-ND2H improvement was

again gained by the SVM-CFS classifier in the Data Class set, changing from a 1-ND2H score of 0.29 (Scott-Knott Rank 7 of 7) to 0.70 (Scott-Knott Rank 2 of 5). SMOTE improved the 1-ND2H scores of 75% of the classifier instances, had no effect on 15%, and decreased the 1-ND2H scores for 10% of the classifiers. Again, these decreases were relatively minimal, with the worst case of CART in the God Class seeing a 1-ND2H decrease from 0.78 (Rank 1 of 7) to 0.64 (Rank 3 of 5). The average and median improvements to 1-ND2H scores across all classifier instances were by 0.12 and 0.10, respectively, whereas the average and median decreases were by -0.05 and -0.04, respectively.

As can be seen in **Figures 4 and 5**, the gains in F1 and 1-ND2H often come at the cost of decreased ACC, but we argue that this is due to the nature of the ACC metric, and its inappropriateness when applied to highly-imbalanced data. The overall increase in F1 and 1-ND2H alongside the decrease of ACC when applying SMOTE suggests that the overall predictive values of our classifiers are improving. We seem to be increasing our True Positive Rate (TPR) to a higher degree than we are increasing our False Positive Rate (FPR); both increases can be expected from introducing mutations of the minority class into our datasets as SMOTE does. Indeed, we observed that our TPR increased as much as or more than our FPR in 63% of the cases where SMOTE was applied. Due to these increases in F1 and 1-ND2H, applying SMOTE to the code smell datasets does seem to greatly increase the performance and predictive value of our classifiers in code smell detection.

5.3 RQ3 Results

> ***RQ3:*** *Does applying feature selection improve the performance of the studied classifiers for code smell detection?*

The effects of applying the three feature selection algorithms on the Scott-Knott rankings of individual classifiers for each code smell dataset (with SMOTE applied) are given in **Figure 6**. For a more detailed view, please see the full Scott-Knott ranking in the appendix.

As shown in the included figure, the application of feature selection to the default classifiers has a strong effect on the placement of a classifier in the Scott-Knott rankings, both in terms of F1 and ND2H. Note that we do not focus here on the ACC performance metric, for the reasons discussed in the previous section. Our results show that for all of the default classifiers, there is a feature selection technique that will provide equivalent or better performance in terms of F1 and ND2H. Even in cases of no rank improvement, the value of reducing the number of features may be significant in terms of domain understanding and simplicity — we are reducing the feature space from 61-82 features down to 6-10 features.

From the perspective of the effect of individual feature selection algorithms on the performance of a classifier, we see that across all datasets and classifiers the F1 rank is only decreased in 6.25% of the cases, remains the same in 33.33% of the cases, and is increased in 60.42% of the cases. Likewise for ND2H, we see that across all datasets and classifiers, the rank is only decreased in 8.33% of the cases, remains the same in 31.25% of the cases, and is increased in 60.42% of the cases.

Of particular note are cases such as that of SVM, which was often placed near the bottom of the Scott-Knott ranking in terms of both F1 and ND2H when no feature selection was applied. With feature selection applied, however, it was frequently placed near the top of both rankings. Take for example SVM's performance in the God Class set, where previous to feature selection it had a median F1 score of 0.36 (Rank 4 of 5) and a median 1-ND2H score of 0.44 (Rank 5 of 5). When the SVMF feature selection algorithm was applied, the median F1 score rose to 0.64 (Rank 1 of 5) and the median 1-ND2H score rose to 0.71 (Rank 2 of 5).

In addition, if we focus only on the top-ranking F1 classifiers across all datasets, we see that 66.67% employ a feature selection algorithm. Likewise, from among the top-ranking ND2H classifiers we see that 73.3% employ a feature selection algorithm.

Despite some variety in the success of individual feature selection algorithms for specific classifiers in certain code smell datasets, we overall see great promise in applying feature selection techniques to improve not only the performance, but also the comprehensibility and simplicity of code smell detection methods.

Data Class

	F-Score Rank Change	Norm DtH Rank Change
NB	**Rank 3 of 4**	**Rank 2 of 3**
NB-CFS	0	0
NB-DTF	0	-1
NB-SVMF	0	0
SVM	**Rank 4 of 4**	**Rank 3 of 3**
SVM-CFS	+1	0
SVM-DTF	+2	+2
SVM-SVMF	+3	+2
CART	**Rank 3 of 4**	**Rank 2 of 3**
CART-CFS	+2	+1
CART-DTF	+1	+1
CART-SVMF	+2	+1
RF	**Rank 4 of 4**	**Rank 3 of 3**
RF-CFS	+2	+1
RF-CTF	+2	+2
RF-SVMF	+3	+2

God Class

	F-Score Rank Change	Norm DtH Rank Change
NB	**Rank 2 of 5**	**Rank 2 of 5**
NB-CFS	0	0
NB-DTF	-1	-1
NB-SVMF	-1	-1
SVM	**Rank 4 of 5**	**Rank 5 of 5**
SVM-CFS	+2	+3
SVM-DTF	+1	+2
SVM-SVMF	+3	+3
CART	**Rank 3 of 5**	**Rank 3 of 5**
CART-CFS	0	0
CART-DTF	+2	+2
CART-SVMF	+1	+1
RF	**Rank 5 of 5**	**Rank 5 of 5**
RF-CFS	+1	+1
RF-CTF	+3	+3
RF-SVMF	+2	+2

Feature Envy

	F-Score Rank Change	Norm DtH Rank Change
NB	**Rank 3 of 6**	**Rank 3 of 6**
NB-CFS	+1	+1
NB-DTF	-1	-2
NB-SVMF	0	+1
SVM	**Rank 5 of 6**	**Rank 4 of 6**
SVM-CFS	+2	+1
SVM-DTF	+1	0
SVM-SVMF	+2	+1
CART	**Rank 6 of 6**	**Rank 6 of 6**
CART-CFS	0	0
CART-DTF	+1	+2
CART-SVMF	0	0
RF	**Rank 5 of 6**	**Rank 5 of 6**
RF-CFS	0	+1
RF-CTF	+1	+1
RF-SVMF	0	0

Long Method

	F-Score Rank Change	Norm DtH Rank Change
NB	**Rank 2 of 5**	**Rank 2 of 5**
NB-CFS	0	0
NB-DTF	+1	+2
NB-SVMF	+1	+2
SVM	**Rank 4 of 5**	**Rank 4 of 5**
SVM-CFS	0	0
SVM-DTF	+2	+3
SVM-SVMF	+1	+2
CART	**Rank 5 of 5**	**Rank 5 of 5**
CART-CFS	0	0
CART-DTF	+2	+3
CART-SVMF	0	0
RF	**Rank 5 of 5**	**Rank 5 of 5**
RF-CFS	0	0
RF-CTF	+2	+3
RF-SVMF	0	0

Figure 6: RQ3 - Scott Knott rank changes (F-Score, 1 - Normalized Distance to Heaven) resulting from feature selection algorithms.

5.4 RQ4 Results

> ***RQ4:*** *Does clustering the data help reduce the computation and/or improve the performance of the studied classifiers for code smell detection?*

We performed stratified 5-fold cross-validation, 5 times for each over-sampled dataset. For four different datasets, we have a total of 100 runs, 25 per dataset where we compare the median values of ACC, F1 and 1 - ND2H with and without clustering.

- **Computation**: Out of 100 runs in total, we found that there were only 6 runs with 1 or 2 clusters such that the distribution of class labels within the cluster was at least 90:10 or better as described in Section 4.3. This even increased the turn-around/computation time of the ML classifiers, which contradicted our hypothesis, as the model had to go through the K-means overhead in every run.
- **Performance**: We observed a performance improvement with Data Class (**Figure 7**), but it was not significant as the overall Scott-Knott rankings suggest. For Data Class, Naive Bayes and ONER do claim a significant improvement in ACC, for ONER - ACC increased from 0.57 without clustering to 0.67 with clustering which is 17.5% improvement and for Naive Bayes - it increased from 0.46 to 0.68, an improvement of 47.8%. But for SVM, CART and RF, the change is insignificant or negligible across all the three performance metrics. For the God Class code smell, there is a significant decrease in F1 and 1 - ND2H, the maximum decrease was observed in case of Naive Bayes where the F1 scores dropped from 0.58 without clustering to 0.36 with clustering, which is a 38% decrease (**Figure 8**). We find the exact same trends with Feature Envy and Long Method datasets (included in the appendix), with a significant decrease in F1 and 1 - ND2H. Moreover, across all four code smell datasets, the noticeable changes can only be seen with Naive Bayes and ONER while the other three classifiers perform almost the same with or without clustering.

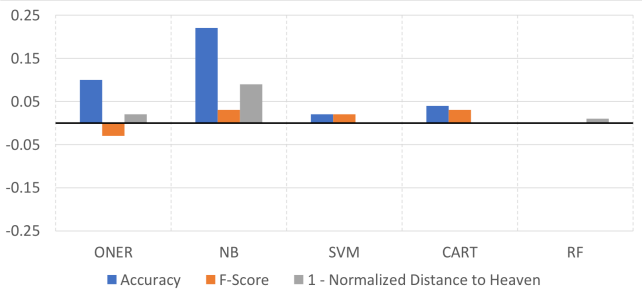

Figure 7: Clustering - Data Class Performance Changes

We speculate the following to be the primary reasons for the contradicting results.

- The distribution of data points across the feature space in the oversampled training set must be such that there is an excessive amount of overlap amongst the smelly and non-smelly data points which makes it difficult for K-means to determine coherent cluster boundaries. The smelly and non-smelly instances if visualized, would tend to occur in random uneven shapes other than the spherical blobs that K-means tends to work best with.
- We used Elbow Method to determine the number of clusters for the Data Class and Long Method datasets. The God Class and Feature Envy plots showed a more uniform curve that gave an ambiguous elbow arc. Better alternatives are discussed later in Future Work (Section 7).

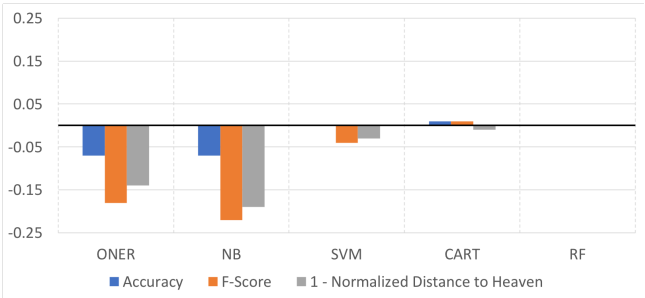

Figure 8: Clustering - God Class Performance Changes

6 CONCLUSIONS

In these experiments, we work to further the findings in Nucci et. al 2018 [15], by replicating the methods they used to reconstruct the code smell datasets. Our novel additions to code smell detection include clustering, oversampling, and feature selection to improve classifier performance. Clustering had a low effect while oversampling and feature selection tended to improve the performance of most classifiers. In performance on each dataset, many of the top ranked classification algorithms received a feature selection treatment. Consequently, we believe these techniques are an important step in optimizing and improving classifier performance on small code smell datasets.

7 FUTURE WORK

There is plenty to explore and experiment with that could extend our study on 'Optimizing Code Smells Detection Techniques'.

- **Hyperparameter Optimization**: To limit the scope of our study, we do not perform any hyperparameter optimization but changed the number of trees from 10 to 100 in case of Random Forest. The principal aim of our study was to experiment with **Oversampling**, **Feature Selection** and **Clustering** which was excluded in the research conducted by Nucci, et al. [15]. While Nucci, et al. use Grid Search using Weka [9] for tuning the classifiers, we believe finding the optimal set of hyperparameters in a parameter space so vast, for a large number of classifiers involves an extensive amount of computation. A smarter approach would be to use Differential Evolution that performs stochastic jumps across the parameter space as opposed to Grid Search that performs a complete search, which makes DE considerably faster as suggested by Menzies, et al in their research [8].
- **Tunable Components**: Apart from tuning the predictive classifiers using hyperparameter optimization, the process

described in Section 4.2 involves a lot of other tunable components listed as follows:
  - **Feature Selectors**: We focus on three different feature selectors in our study. CFS and two Wrappers with Decision Tree and SVM as their underlying classifiers respectively. While in our experiment, all three feature selectors tried to pick the set of features that resulted in maximum ACC, this goal can be changed based on the requirements which would be a more systematic approach. Also, the number of features to be selected can be varied as required.
  - **SMOTE**: We use SMOTE for oversampling the data with default parameters but it has numerous tunable parameters like *number of nearest neighbours used to construct synthetic samples*, *extrapolation step size*, *number of nearest neighbours used to determine if a minority sample is in danger*, *type of SMOTE algorithm to be used* etc.
- **Clustering**: We used K-means clustering with Elbow method to choose the number of clusters which failed to improve the predictive performance and reduce the computation. Based on the speculations listed in Section 5.4:
  - We believe a better alternative to K-means would be to use DBScan (Density Based Spatial Clustering of Applications with Noise) [4] which is tested to work much more effectively when the distribution of data points across multiple class labels has a lot of overlap and exists in uneven shapes.
  - A more robust method to select the initial number of clusters like the Average Silhouette method or the Gap Statistic [17] method might have resulted in a greater quantity of clusters where the distribution of class labels is 90:10 or better.
- **No Free Lunch Candidates**: The following can be perfectly classified as NFL candidates as there is no specific hypothesis backing these experiments but the only way to check if they work in the favour of detecting code smells is to try them on our datasets.
  - Over-sample the training instances before reducing the number of features.
  - Combine clustering with feature selection.
  - Check if clustering gives better results without over-sampling the data using SMOTE.
  - Try other machine learning classifiers to predict code smells. We use 5 classifiers in our experiment while Nucci, et al. used 32 different variants.
  - Experiment with other over-sampling techniques apart from SMOTE like Repetition or Bootstrapping.
- **Better DATA**: We had to resort to oversampling as the original datasets we started with (before combining the datasets) had very few samples to predict a particular code smell. More data would help, but at the same time, it must be noted that more data is not necessarily better data - collecting quality samples is more likely to help in predicting code smells. As discussed in Section 5.1, the default classifiers frequently do not perform any better than random guessing or one-rule classification, regardless of the specific performance metric or dataset. We believe that quality data has the capability to increase the predictive power of non-trivial classifiers more than anything else.

8 THREATS TO VALIDITY

8.1 Threats to Conclusion Validity

The effect of Class Imbalance [3] on classifier performance is amplified in small datasets. It leads to underperformance on classifying the minority class. In our case, for every dataset, the smelly examples are the minority class, and the non-smelly examples are the majority class. Therefore it is possible the classifiers could perform better at classifying True code smells if the datasets had more examples.

Hyperparameter optimization was not applied to any classifier which could impact how often the classifiers are improved with feature selection.

8.2 Threats to Generalization Validity

Due to time constraints, only a small number of classifiers and feature selection algorithms were included in the experiment which makes it more difficult for our results to generalize to the broader set of classifiers, and feature selection algorithms.

9 AI BASELINES

- **Simple**: Reducing the feature space provides a simpler way to understand what code characteristics lead to code smells.
- **Useful**: Our random guess classifier performance acts as the level zero signpost to benchmark the performance of all other classifiers.
- **Available**: Our project was constructed using open-source Python libraries.

REFERENCES

[1] Francesca Arcelli Fontana, Mika V. Mäntylä, Marco Zanoni, and Alessandro Marino. 2016. Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering* 21, 3 (01 Jun 2016), 1143–1191. https://doi.org/10.1007/s10664-015-9378-4

[2] David Arthur and Sergei Vassilvitskii. 2007. K-means++: The Advantages of Careful Seeding. In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '07)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1027–1035. http://dl.acm.org/citation.cfm?id=1283383.1283494

[3] Swagatam Das, Shounak Datta, and Bidyut B Chaudhuri. 2018. Handling data irregularities in classification: Foundations, trends, and future challenges. *Pattern Recognition* 81 (2018), 674–693.

[4] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. 1996. A Density-based Algorithm for Discovering Clusters a Density-based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD'96)*. AAAI Press, 226–231. http://dl.acm.org/citation.cfm?id=3001460.3001507

[5] B. Walter A. Yamashita F. A. Fontana, J. Dietrich and M. Zanoni. 2016. Antipattern and code smell false positives: Preliminary conceptualization and classification. *JIEEE 23rd International Conference* 1 (2016), 609–613.

[6] P. Braione F. A. Fontana and M. Zanoni. 2012. Automatic detection of bad smells in code: An experimental assessment. *Journal of Object Technology* 11 (2012), 5–1.

[7] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional. http://www.amazon.ca/exec/obidos/redirect?tag=citeulike04-20

[8] Wei Fu, Vivek Nair, and Tim Menzies. 2016. Why is Differential Evolution Better than Grid Search for Tuning Defect Predictors? *CoRR* abs/1609.02613 (2016). arXiv:1609.02613 http://arxiv.org/abs/1609.02613

[9] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. 2009. The WEKA Data Mining Software: An Update. *SIGKDD Explor. Newsl.* 11, 1 (Nov. 2009), 10–18. https://doi.org/10.1145/1656274.1656278

[10] M. Lanza and R. Marinescu. 2006. Object-Oriented Metrics in Practice. *Springer Science & Business Media* (2006).

[11] M. M Lehman. 1980. Programs, life cycles, and laws of software evolution. In *Proceedings of the IEEE, vol. 68, no. 9.* 1060–1076.

[12] Guillaume Lemaître, Fernando Nogueira, and Christos K. Aridas. 2017. Imbalanced-learn: A Python Toolbox to Tackle the Curse of Imbalanced Datasets in Machine Learning. *Journal of Machine Learning Research* 18, 17 (2017), 1–5. http://jmlr.org/papers/v18/16-365

[13] Jundong Li, Kewei Cheng, Suhang Wang, Fred Morstatter, Trevino Robert, Jiliang Tang, and Huan Liu. 2016. Feature Selection: A Data Perspective. *arXiv:1601.07996* (2016).

[14] Nikolaos Mittas and Lefteris Angelis. 2013. Ranking and Clustering Software Cost Estimation Models Through a Multiple Comparisons Algorithm. *IEEE Trans. Softw. Eng.* 39, 4 (April 2013), 537–551. https://doi.org/10.1109/TSE.2012.45

[15] Dario Di Nucci, Fabio Palomba, Damian A. Tamburri, Alexander Serebrenik, and Andrea De Lucia. 2018. Detecting code smells using machine learning techniques: Are we there yet?. In *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018.* 612–621. https://doi.org/10.1109/SANER.2018.8330266

[16] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[17] Robert Tibshirani, Guenther Walther, and Trevor Hastie. [n. d.]. Estimating the number of clusters in a data set via the gap statistic. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 63, 2 ([n. d.]), 411–423. https://doi.org/10.1111/1467-9868.00293 arXiv:https://rss.onlinelibrary.wiley.com/doi/pdf/10.1111/1467-9868.00293

[18] Shull F. Fredericks M. Travassos, G. and V. R. Basili. 1999. Detecting defects in object-oriented designs: using reading techniques to increase software quality. In *Proceedings of the 14th Conference on Object-Oriented Programming, Systems, Languages, and Applications.* 47âĂŞ56.

[19] N. Tsantalis and A. Chatzigeorgiou. 2011. Identification of extract method refactoring opportunities for the decomposition of methods. *Journal of Systems and Software* 84(10) (2011), 1757–1782.

APPENDIX

RQ1 Additional Figures

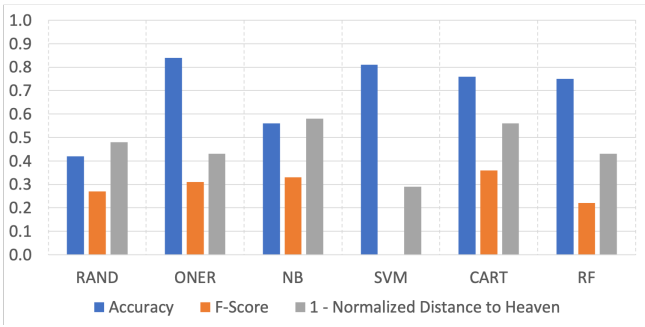

Figure 9: RQ1 - Comparison of classifier performance metrics for the Data Class set.

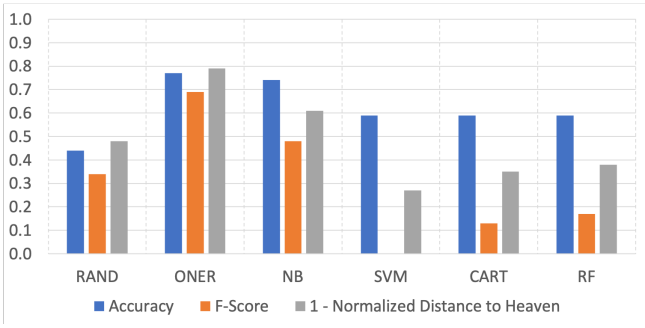

Figure 10: RQ1 - Comparison of classifier performance metrics for the Feature Envy set.

RQ2 Additional Figures

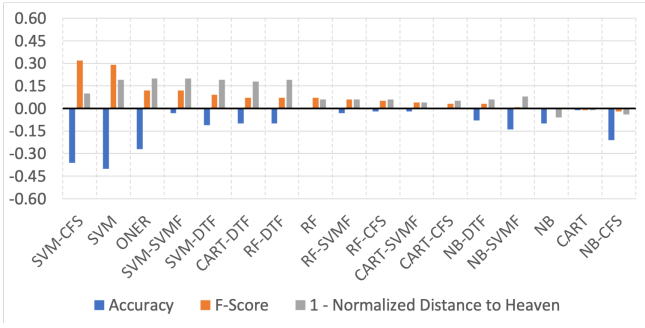

Figure 11: RQ2 - Change in the classifier performance metrics when using SMOTE for the Data Class set.

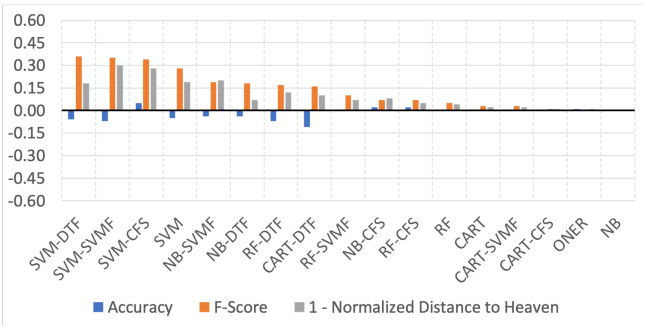

Figure 12: RQ2 - Change in the classifier performance metrics when using SMOTE for the Feature Envy set.

RQ3 Additional Figures

Data Class

	Rank by F-Score			Rank by 1 - Norm DtH		
	Rank	Med	IQR	Rank	Med	IQR
RAND	3	0.27	0.09	2	0.48	0.10
ONER	2	0.31	0.25	3	0.43	0.15
NB	2	0.33	0.08	1	0.58	0.08
NB-CFS	2	0.35	0.09	1	0.55	0.09
NB-DTF	3	0.27	0.25	3	0.43	0.14
NB-SVMF	2	0.31	0.15	3	0.43	0.12
SVM	4	0.00	0.00	4	0.29	0.00
SVM-CFS	4	0.00	0.00	4	0.29	0.00
SVM-DTF	2	0.31	0.25	3	0.43	0.14
SVM-SVMF	2	0.33	0.18	3	0.43	0.08
CART	1	0.36	0.14	1	0.56	0.10
CART-CFS	1	0.42	0.19	1	0.57	0.17
CART-DTF	2	0.33	0.25	3	0.43	0.18
CART-SVMF	1	0.40	0.21	1	0.57	0.15
RF	3	0.22	0.24	3	0.43	0.18
RF-CFS	2	0.33	0.22	2	0.50	0.13
RF-DTF	2	0.33	0.26	3	0.43	0.15
RF-SVMF	1	0.40	0.11	1	0.54	0.07

God Class

	Rank by F-Score			Rank by 1 - Norm DtH		
	Rank	Med	IQR	Rank	Med	IQR
RAND	4	0.36	0.08	5	0.49	0.08
ONER	1	0.68	0.06	1	0.80	0.05
NB	2	0.59	0.14	2	0.72	0.09
NB-CFS	2	0.56	0.13	2	0.69	0.15
NB-DTF	3	0.45	0.18	4	0.54	0.19
NB-SVMF	3	0.51	0.17	3	0.62	0.18
SVM	6	0.00	0.00	7	0.29	0.00
SVM-CFS	6	0.00	0.00	7	0.29	0.00
SVM-DTF	6	0.00	0.11	7	0.29	0.05
SVM-SVMF	3	0.45	0.19	4	0.54	0.13
CART	1	0.69	0.10	1	0.78	0.12
CART-CFS	3	0.50	0.24	4	0.59	0.20
CART-DTF	1	0.76	0.17	1	0.84	0.17
CART-SVMF	2	0.59	0.12	3	0.67	0.12
RF	5	0.22	0.13	6	0.42	0.09
RF-CFS	5	0.25	0.18	6	0.44	0.14
RF-DTF	2	0.56	0.33	3	0.66	0.20
RF-SVMF	3	0.44	0.17	4	0.57	0.15

Feature Envy

	Rank by F-Score			Rank by 1 - Norm DtH		
	Rank	Med	IQR	Rank	Med	IQR
RAND	3	0.34	0.10	3	0.48	0.09
ONER	1	0.69	0.11	1	0.79	0.12
NB	2	0.48	0.07	2	0.61	0.09
NB-CFS	2	0.48	0.18	2	0.61	0.12
NB-DTF	4	0.15	0.19	5	0.35	0.12
NB-SVMF	3	0.33	0.29	3	0.48	0.21
SVM	6	0.00	0.00	7	0.27	0.01
SVM-CFS	6	0.09	0.12	6	0.32	0.07
SVM-DTF	6	0.00	0.13	6	0.29	0.06
SVM-SVMF	5	0.13	0.27	5	0.35	0.13
CART	5	0.13	0.09	5	0.35	0.07
CART-CFS	6	0.10	0.15	6	0.33	0.07
CART-DTF	5	0.11	0.20	5	0.34	0.11
CART-SVMF	6	0.08	0.11	6	0.32	0.06
RF	4	0.17	0.15	5	0.38	0.12
RF-CFS	4	0.21	0.12	4	0.41	0.09
RF-DTF	5	0.13	0.15	5	0.35	0.08
RF-SVMF	5	0.12	0.12	5	0.35	0.08

Long Method

	Rank by F-Score			Rank by 1 - Norm DtH		
	Rank	Med	IQR	Rank	Med	IQR
RAND	1	0.31	0.07	1	0.49	0.10
ONER	3	0.00	0.00	3	0.29	0.00
NB	1	0.24	0.10	1	0.43	0.09
NB-CFS	2	0.21	0.17	1	0.42	0.09
NB-DTF	2	0.13	0.29	2	0.34	0.08
NB-SVMF	3	0.00	0.10	3	0.29	0.05
SVM	3	0.00	0.00	3	0.28	0.02
SVM-CFS	3	0.00	0.00	3	0.28	0.01
SVM-DTF	3	0.00	0.00	3	0.29	0.00
SVM-SVMF	3	0.00	0.00	3	0.29	0.00
CART	3	0.00	0.00	3	0.28	0.01
CART-CFS	3	0.00	0.00	3	0.28	0.01
CART-DTF	3	0.00	0.00	3	0.29	0.00
CART-SVMF	3	0.00	0.00	3	0.28	0.01
RF	3	0.00	0.00	3	0.28	0.01
RF-CFS	3	0.00	0.00	3	0.28	0.01
RF-DTF	3	0.00	0.00	3	0.29	0.00
RF-SVMF	3	0.00	0.00	3	0.28	0.01

Figure 13: RQ3 - Overall Scott-Knott rankings of classifier variations on Non-SMOTE datasets.

Data Class

	Rank by F-Score			Rank by 1 - Norm DtH		
	Rank	Med	IQR	Rank	Med	IQR
RAND	4	0.27	0.09	3	0.48	0.10
ONER	1	0.43	0.03	1	0.63	0.07
NB	3	0.33	0.08	2	0.52	0.08
NB-CFS	3	0.33	0.06	2	0.51	0.06
NB-DTF	3	0.30	0.13	3	0.49	0.12
NB-SVMF	3	0.32	0.08	2	0.51	0.09
SVM	4	0.29	0.11	3	0.48	0.11
SVM-CFS	3	0.32	0.07	3	0.39	0.19
SVM-DTF	2	0.40	0.10	1	0.62	0.10
SVM-SVMF	1	0.45	0.13	1	0.63	0.08
CART	3	0.35	0.22	2	0.55	0.21
CART-CFS	1	0.45	0.14	1	0.62	0.12
CART-DTF	2	0.40	0.13	1	0.61	0.13
CART-SVMF	1	0.44	0.13	1	0.61	0.13
RF	4	0.29	0.17	3	0.49	0.13
RF-CFS	2	0.38	0.15	2	0.56	0.13
RF-DTF	2	0.40	0.12	1	0.62	0.13
RF-SVMF	1	0.46	0.15	1	0.60	0.16

God Class

	Rank by F-Score			Rank by 1 - Norm DtH		
	Rank	Med	IQR	Rank	Med	IQR
RAND	4	0.36	0.08	4	0.49	0.08
ONER	1	0.68	0.06	1	0.80	0.05
NB	2	0.58	0.13	2	0.72	0.09
NB-CFS	2	0.58	0.15	2	0.71	0.19
NB-DTF	3	0.52	0.09	3	0.62	0.10
NB-SVMF	3	0.49	0.09	3	0.59	0.11
SVM	4	0.36	0.10	5	0.44	0.11
SVM-CFS	2	0.58	0.09	2	0.70	0.12
SVM-DTF	3	0.50	0.20	3	0.52	0.25
SVM-SVMF	1	0.64	0.17	2	0.71	0.13
CART	3	0.52	0.09	3	0.63	0.07
CART-CFS	3	0.51	0.33	3	0.65	0.27
CART-DTF	1	0.69	0.14	1	0.78	0.14
CART-SVMF	2	0.56	0.17	2	0.66	0.12
RF	5	0.22	0.13	5	0.42	0.10
RF-CFS	4	0.29	0.23	4	0.47	0.19
RF-DTF	2	0.56	0.24	2	0.69	0.22
RF-SVMF	3	0.48	0.17	3	0.61	0.13

Feature Envy

	Rank by F-Score			Rank by 1 - Norm DtH		
	Rank	Med	IQR	Rank	Med	IQR
RAND	4	0.34	0.10	4	0.48	0.09
ONER	1	0.69	0.10	1	0.80	0.10
NB	3	0.48	0.10	3	0.61	0.07
NB-CFS	2	0.55	0.16	2	0.69	0.12
NB-DTF	4	0.33	0.18	5	0.42	0.17
NB-SVMF	3	0.52	0.14	2	0.68	0.14
SVM	5	0.28	0.10	4	0.46	0.09
SVM-CFS	3	0.43	0.18	3	0.60	0.17
SVM-DTF	4	0.36	0.14	4	0.47	0.11
SVM-SVMF	3	0.48	0.16	3	0.65	0.13
CART	6	0.16	0.11	6	0.37	0.08
CART-CFS	6	0.11	0.11	6	0.34	0.08
CART-DTF	5	0.27	0.13	4	0.44	0.10
CART-SVMF	6	0.11	0.07	6	0.34	0.05
RF	5	0.22	0.18	5	0.42	0.13
RF-CFS	5	0.28	0.14	4	0.46	0.11
RF-DTF	4	0.30	0.12	4	0.47	0.10
RF-SVMF	5	0.22	0.13	5	0.42	0.10

Long Method

	Rank by F-Score			Rank by 1 - Norm DtH		
	Rank	Med	IQR	Rank	Med	IQR
RAND	1	0.31	0.07	1	0.49	0.10
ONER	2	0.26	0.16	2	0.44	0.12
NB	2	0.25	0.08	2	0.44	0.07
NB-CFS	2	0.29	0.12	2	0.42	0.08
NB-DTF	1	0.32	0.07	1	0.48	0.17
NB-SVMF	1	0.32	0.07	1	0.52	0.09
SVM	4	0.14	0.10	4	0.35	0.11
SVM-CFS	4	0.15	0.08	4	0.37	0.06
SVM-DTF	2	0.27	0.11	1	0.48	0.11
SVM-SVMF	3	0.23	0.18	2	0.45	0.15
CART	5	0.00	0.09	5	0.28	0.05
CART-CFS	5	0.00	0.10	5	0.28	0.07
CART-DTF	3	0.19	0.13	2	0.41	0.13
CART-SVMF	5	0.00	0.09	5	0.29	0.05
RF	5	0.00	0.09	5	0.28	0.05
RF-CFS	5	0.00	0.17	5	0.28	0.13
RF-DTF	3	0.19	0.15	2	0.41	0.14
RF-SVMF	5	0.00	0.09	5	0.28	0.06

Figure 14: RQ3 - Overall Scott Knott rankings on SMOTE datasets of classifier variations on SMOTE datasets.

RQ4 Additional Figures

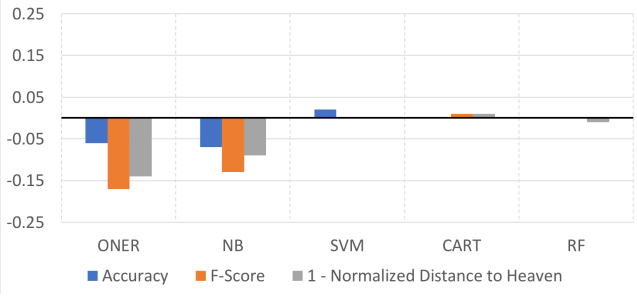

Figure 15: Clustering - Feature Envy Performance Changes

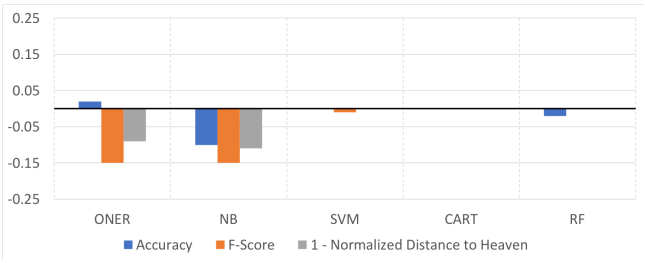

Figure 16: Clustering - Long Method Performance Changes