Just-In-Time Software Defect Prediction

Ali Mohamed, Yifei Gong, Yunhua Zhao

2. Checklist

GitHub link: https://github.com/Yunhua468/Machine_Learning

Poster link: https://github.com/Yunhua468/Machine_Learning/tree/main/poster

3. Problem Description/Goals

Project Category: predict if a code change is bug including.

Data: two datasets:

- Openstack and Qt, which are commonly tested by previous researchers.
- Openstack (10658 negative samples, 1616 positive samples), Qt (23148 negative samples, 2002 positive samples).
- Each sample contains information from a GitHub commit.
- Each sample has 35 features extracted from that commit (lines modified, developer experience, etc.) and 1 target values (buggy or not).

Goals: compete with state-of-the-art work by using handcrafted features

Team Member Contributions:

Previous works all used the same model to train, test, and evaluate on two different datasets. We think different datasets may have different result using different model, and they may share intrinsic similarities that can be connected. That's why we divide the work to 3 parts, one focuses on getting the best out of Openstack, one focuses on getting the best out of Qt, and another one focuses on training on the combined dataset.

Each of us try our best using different methods to get the best result

- Yunhua: Qt
- Ali: Openstack

- Yifei: Openstack and Qt combined

4.: Team Member Roles:

Individual report writeup:

- Yunhua: Page 5 to Page 15
- Yifei: Page 16 to Page 30
- Ali: Page 31 to Page 38

Introduction

Software quality assurance places and important role in developing high quality software system. Software defect prediction plays a key role to keep software quality assurance, but it causes an expensive consequence in a released products, also it may influence companies reputation. Recently, just-in-time software defect prediction attracts more and more attention, which predicts defects or bugs in code changes. JIT-SDP can predict defects earlier and makes it easier to debug due to the fewer lines of code comparing to module level defect prediction, also the code change is still fresh to developers which also saves debug time.

In our project, we want to use methods learnt from our machine learning class to make contributions to JIT-SDP. We use two datasets, qt and OPENSTACK, first we want to dig a little bit the repository to get a basic understanding about how the datasets generated, then we want to see some characteristics of these two datasets, then we will implement data balancing methods to balance our datasets, we do data preprocessing before we feed data to our machine learning models, then we will evaluate our model by accuracy,f1, auc, recall and precision, then we will analysis our results.

Related work

Recently, several machine learning techniques have been widely used in JIT-SDP. Some work builds models based on software metrics. Catolino[2] points that there is still room to improvement in JIT-SDP, they investigate which software metrics matter more in mobile context, if different classifiers impact the performance of cross-project JIT bug prediction models and whether ensemble techniques improves the capabilities of the models. Pascarella[3] al et propose a fine-grained JIT-SDP to predict the specify files which may contain defects.

There are also several researches implement deep learning in JIT-SDP. DAECNN-JDP[3] is based on denoising autoencoder and convolutional neural network, which uses convolutional neural network on software metrics to build CNN model to predict within and cross project. Also, there are some works that do not use software metrics, like HOANG el at[4] propose Deep JIT, they extract features directly from commit message and code changes. This paper is also our project baseline paper, we want to compare our results which is based software metrics with DeepJIT.

Approach

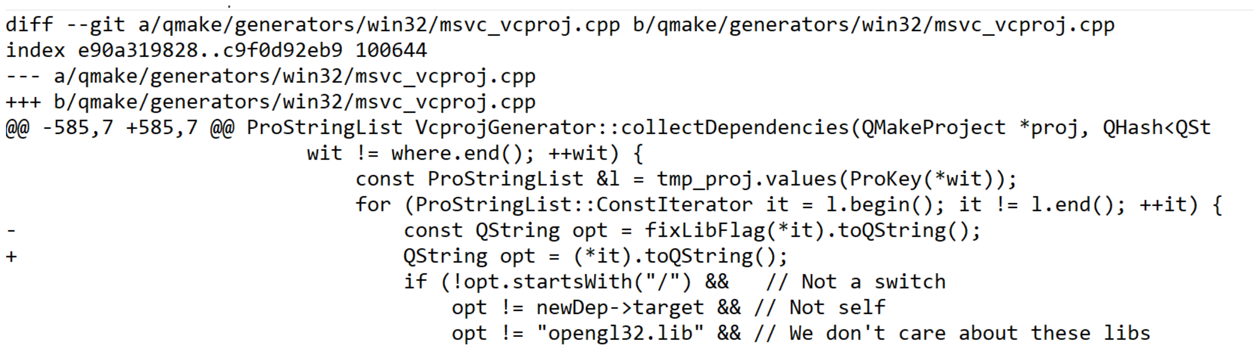
I feel curious about how the dataset is extracted, so I dig the repository first, then preprocessing features, then feed to the models and do the evaluation.

Dig repository

For the data, I use github commands to dig one project of qt, it is named qtbase. The commands I used for example:

- git diff(default algorithm is Mayes)
- git log
- git log --pretty=%P -n 1 "$commit_from"
- git show

So I get some diff files, shown in the figure(I suppose they use the first parent of a commit):



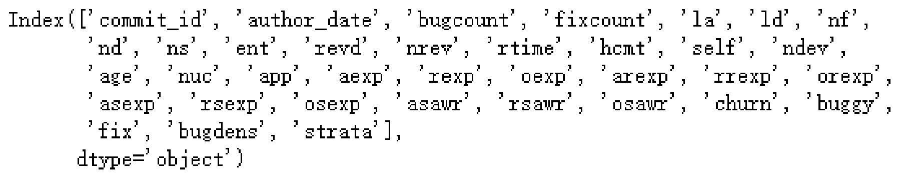
```
diff --git a/qmake/generators/win32/msvc_vcproj.cpp b/qmake/generators/win32/msvc_vcproj.cpp
index e90a319828..c9f0d92eb9 100644
--- a/qmake/generators/win32/msvc_vcproj.cpp
+++ b/qmake/generators/win32/msvc_vcproj.cpp
@@ -585,7 +585,7 @@ ProStringList VcprojGenerator::collectDependencies(QMakeProject *proj, QHash<QSt
                         wit != where.end(); ++wit) {
                             const ProStringList &l = tmp_proj.values(ProKey(*wit));
                             for (ProStringList::ConstIterator it = l.begin(); it != l.end(); ++it) {
-                                const QString opt = fixLibFlag(*it).toQString();
+                                QString opt = (*it).toQString();
                                 if (!opt.startsWith("/") &&   // Not a switch
                                     opt != newDep->target && // Not self
                                     opt != "opengl32.lib" && // We don't care about these libs
```

Figure 1 Example of a code commit from project qtbase

QT and OPENSTACK datasets are from this kind of code diff changes, like line of code added, developer experiences and so on. Based on the basic understanding of dataset qt, then I check metrics in qt.

Preprocessing

QT includes 25150 commits, and there are totally 36 columns, the items are shown in



```
Index(['commit_id', 'author_date', 'bugcount', 'fixcount', 'la', 'ld', 'nf',
       'nd', 'ns', 'ent', 'revd', 'nrev', 'rtime', 'hcmt', 'self', 'ndev',
       'age', 'nuc', 'app', 'aexp', 'rexp', 'oexp', 'arexp', 'rrexp', 'orexp',
       'asexp', 'rsexp', 'osexp', 'asawr', 'rsawr', 'osawr', 'churn', 'buggy',
       'fix', 'bugdens', 'strata'],
      dtype='object')
```

Figure 2 Column names of dataset Qt

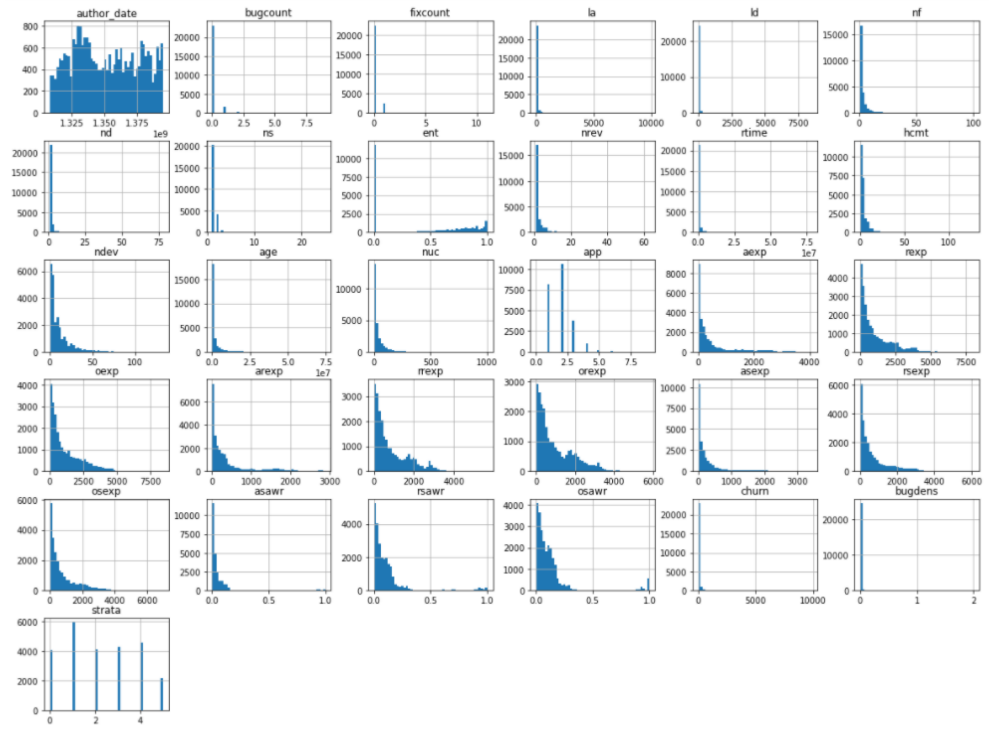


Figure 3 The distribution histogram of these 35 features with relationship of buggy

The ‘commit_id’ and ‘author_date’ are not relative to the label buggy, so I drop them, also, I drop ‘bugcount’ according to the Pearls co-relationship numbers, which ‘bugcount’ result is bigger than 0.8.

Principal component analysis (PCA)

PCA is a linear dimensionality reduction using Singular Value Decomposition of the data to project it to a lower dimensional space. Reducing the number of variables of a data set naturally comes at the expense of accuracy, but the trick in dimensionality reduction is to trade a little accuracy for simplicity.

Class Balance

One of the biggest challenges for classification models is an imbalance of classes in the training data. Severe class imbalances may be masked by relatively good F1 and accuracy scores – the classifier is simply guessing the majority class and not making any evaluation on the under represented class. There are several techniques for dealing with class imbalance such as stratified sampling, down sampling the majority class, weighting and so on.

Polynomial Features

Generate a new feature matrix consisting of all polynomial combinations of the features with degree less than or equal to the specified degree. For example, if an input sample is two dimensional and of the form [a, b], the degree-2 polynomial features are [1, a, b, a^2, ab, b^2].

Experiments

Pre-Processing

For the pre-processing step, table 1 is a summary table of method we used. In the PCA part, I set the 'solver' to 'full' and set 16 components which means the feature number is 16 after PCA function processing, then I scaler the PCA results, this step will speed up the model processing time. Also I implement polynomial features which extend our 16 features to 58 features.

Preprocessing	Parameters
Class balance	Undersampling
PCA	Solver='full', component=16
Polynomial Features	Dimension = 2
Scaler	

Table1 Summary of preprocessing methods

Class balance

QT is an imbalancing dataset, there are 23148 clean commits and 2002 buggy commits. We use the under-sample method to balance classes. We drop 'na' values first and then shuffle the data, then split buggy data into two sets which train: test is 4:1, then we randomly choose same train clean data and 278 clean test data.

Train clean data	Train buggy data	Test clean data	Test buggy data
1586	1586	278	397

Table2 Summary of train and test datasets

Models

We implement supervised machine learning, unsupervised machine learning, deep neural network and I tried to implement reinforcement learning. For supervised machine learning, we implement random forest trees and logistic regression. We implement k-means as our unsupervised machine learning model, we also implement deep neural network on our pre-processed dataset.

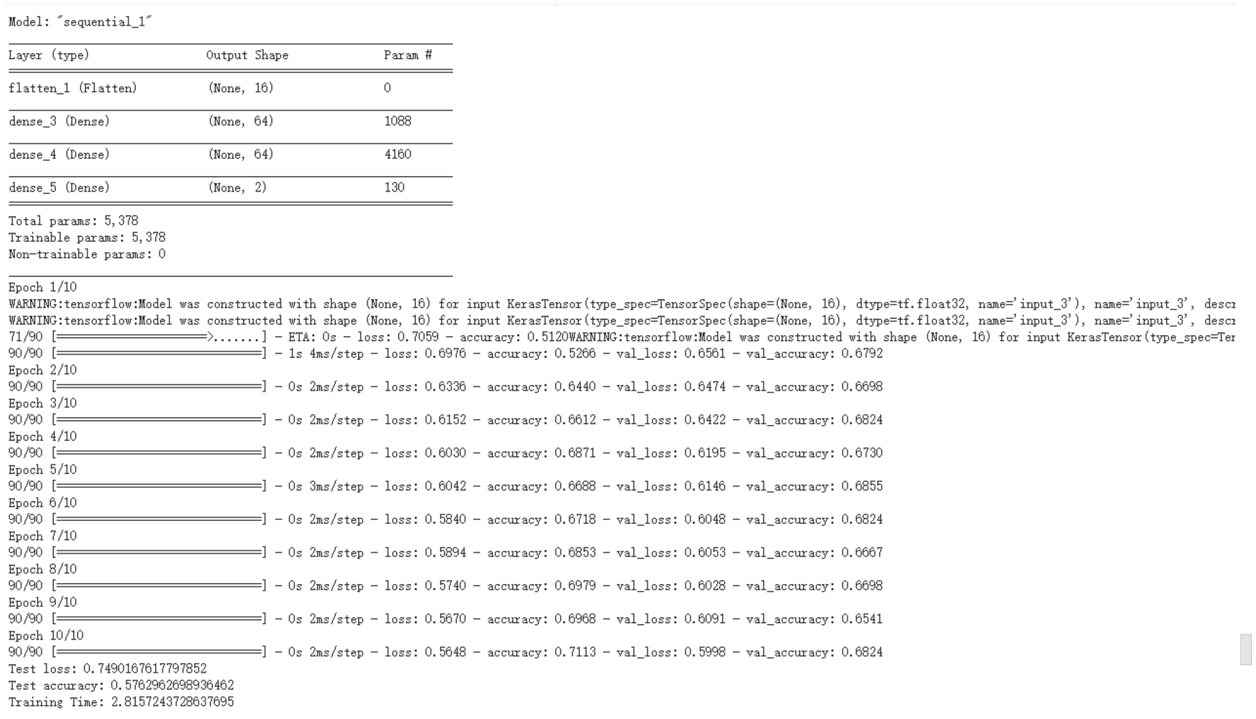
Model	parameter	Pre-processing

Logistic regression	solvers = ['lbfgs', 'liblinear', 'sag']
	C = [1.0, 0.75, 0.5, 0.25, 0.05]
		Class balancing, PCA, Scaler, Poly features

Random forest trees	estimator = [50, 100, 150, 200, 250, 300]
	criterion = ["gini", "entropy"]
		Class balancing, PCA, Scaler, Poly features

K-means	Cluster=2	Class balancing, PCA, Scaler, Poly features
DNN	3 layers(figure 4)	Class balancing, PCA, Scaler

Table3 Summary of model parameters



```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_1 (Flatten)          (None, 16)                0
_________________________________________________________________
dense_3 (Dense)              (None, 64)                1088
_________________________________________________________________
dense_4 (Dense)              (None, 64)                4160
_________________________________________________________________
dense_5 (Dense)              (None, 2)                 130
=================================================================
Total params: 5,378
Trainable params: 5,378
Non-trainable params: 0
_________________________________________________________________
Epoch 1/10
WARNING:tensorflow:Model was constructed with shape (None, 16) for input KerasTensor(type_spec=TensorSpec(shape=(None, 16), dtype=tf.float32, name='input_3'), name='input_3', descr
WARNING:tensorflow:Model was constructed with shape (None, 16) for input KerasTensor(type_spec=TensorSpec(shape=(None, 16), dtype=tf.float32, name='input_3'), name='input_3', descr
71/90 [======================>.......] - ETA: 0s - loss: 0.7059 - accuracy: 0.5120WARNING:tensorflow:Model was constructed with shape (None, 16) for input KerasTensor(type_spec=Ter
90/90 [==============================] - 1s 4ms/step - loss: 0.6976 - accuracy: 0.5266 - val_loss: 0.6561 - val_accuracy: 0.6792
Epoch 2/10
90/90 [==============================] - 0s 2ms/step - loss: 0.6336 - accuracy: 0.6440 - val_loss: 0.6474 - val_accuracy: 0.6698
Epoch 3/10
90/90 [==============================] - 0s 2ms/step - loss: 0.6152 - accuracy: 0.6612 - val_loss: 0.6422 - val_accuracy: 0.6824
Epoch 4/10
90/90 [==============================] - 0s 2ms/step - loss: 0.6030 - accuracy: 0.6871 - val_loss: 0.6195 - val_accuracy: 0.6730
Epoch 5/10
90/90 [==============================] - 0s 3ms/step - loss: 0.6042 - accuracy: 0.6688 - val_loss: 0.6146 - val_accuracy: 0.6855
Epoch 6/10
90/90 [==============================] - 0s 2ms/step - loss: 0.5840 - accuracy: 0.6718 - val_loss: 0.6048 - val_accuracy: 0.6824
Epoch 7/10
90/90 [==============================] - 0s 2ms/step - loss: 0.5894 - accuracy: 0.6853 - val_loss: 0.6053 - val_accuracy: 0.6667
Epoch 8/10
90/90 [==============================] - 0s 2ms/step - loss: 0.5740 - accuracy: 0.6979 - val_loss: 0.6028 - val_accuracy: 0.6698
Epoch 9/10
90/90 [==============================] - 0s 2ms/step - loss: 0.5670 - accuracy: 0.6968 - val_loss: 0.6091 - val_accuracy: 0.6541
Epoch 10/10
90/90 [==============================] - 0s 2ms/step - loss: 0.5648 - accuracy: 0.7113 - val_loss: 0.5998 - val_accuracy: 0.6824
Test loss: 0.7490167617797852
Test accuracy: 0.5762962698936462
Training Time: 2.8157243728637695
```

Figure 4 DNN Parameters

Evaluation

We use accuracy, recall, precision, auc and f1 to evaluate our different models, then we draw the result figures to show the model performance with different parameters, and we plot confusion metrics to show the prediction results. I will choose the highest accuracy result during parameter tuning process.

Analysis

The following figures are the results of our experiments on dataset QT. After poly features, there are total 58 features, we can see random forest tree works better than other methods, and supervised machine learning performs better than unsupervised machine learning. DNN does not show its benefit probably because the small dataset, and also logistic regression based on PCA works better than poly features, recall usually low but precision is relative high.

Model	features	Accuracy(%)	F1(%)	AUC(%)
Logistic regression	PCA	59.3	62.4	64.6
	Poly features	53.2	46.6	60.4
Random forest trees	PCA	62	65	66.3
	Poly features	100	100	100
K-means	PCA	46.5		
	Poly features	50.1		
DNN	PCA	57.6		

Table4 Summary of model results

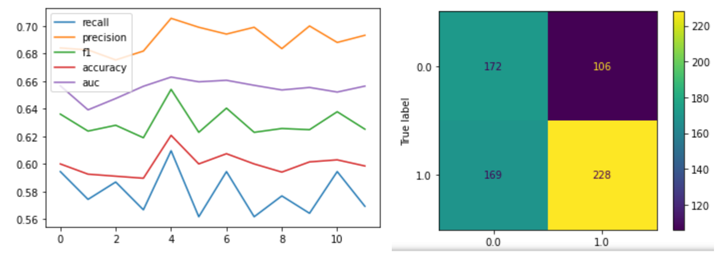


Figure 5 Tune rfc based on pca, left shows the result of different parameter, right shows the best result confusion metrics

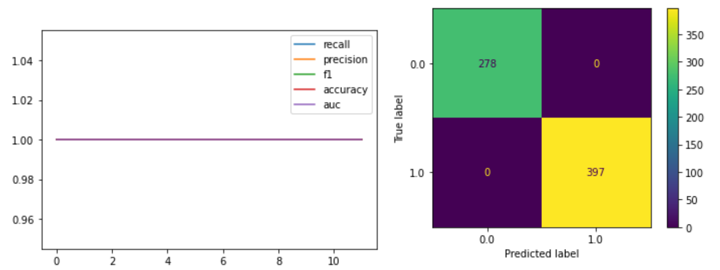


Figure 6 Tune rfc based on poly features, left is the result of different parameter, right shows the best result confusion metrics

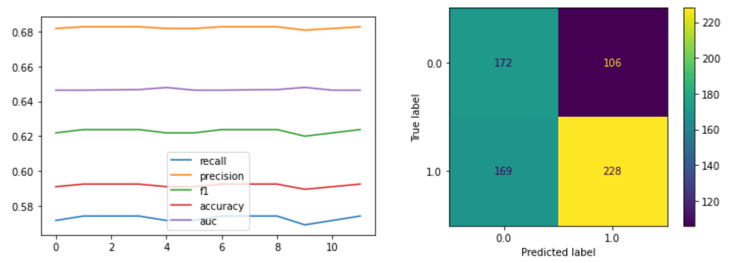


Figure 7 Tune LR based on pca, left shows the result of different parameter, right shows the best result confusion metrics

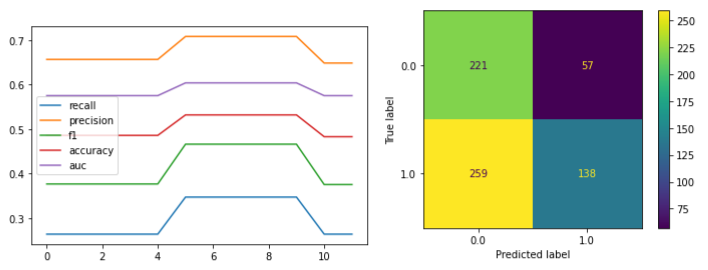


Figure 8 Tune LR based on poly features, left shows the result of different parameter, right shows the best result confusion metrics

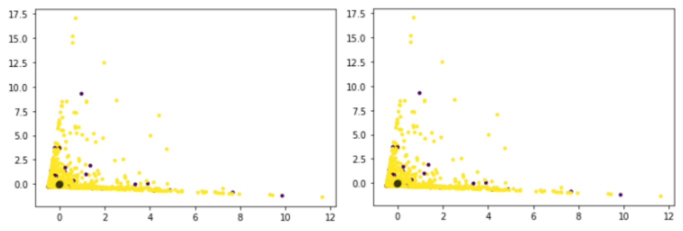


Figure 9 Distribution of features, black dot shows the center of k-mean cluster

Lessons Learned

First, random forest tree with 58 features can totally work perfect on a small dataset, like Qt which is less than 2000 instances after under-sampling balancing. We can see RFC works best, which means here is still space to do more on simple models. Simple model saves time and easier to implement, but it can get the same or better result.

Second, this 3 layers DNN does not perform well, it includes 2 dense layers and one softmax layer, also I shuffled our dataset, although there is no research about the time serious in JIT-SDP based on my knowledge, but I think there is a possibility to dig more if we implement LSTM and follow the commits time serious, because code changes is dependent on precious changes.

Third, supervised machine learning works better than unsupervised machine learning, which also proves some other papers statement about why fewer unsupervised learning models implemented comparing to supervised machine learning.

Fourth, models with software metrics are easier to get data sources compared to models with original codes, although my result fails to DeepJIT.

Fifth, I tried to implement reinforcement learning with some existed library, I used OPENAI, but I did not finish this part code, because several python library version problem, for example they have tensorflow inline, but if my computer tensorflow version is not suitful to the library, there will be an error. Also I did not find any paper on reinforcement learning of JIT-SDP, I think there is a space to do more. I checked some other fields papers, although they say it is not a good idea to use reinforcement learning on the classification problem because of the time consuming, but we still could try more in future since the special of software engineering, which clean code may not real clean.

Yifei's Part

5: State-of-the-art wok

(1) DeepJIT: Generate vector features using CNN on code commit messages.

Thong Hoang, Hoa Khanh Dam, Yasutaka Kamei, David Lo, and Naoyasu Ubayashi. Deepjit: an end-to-end deep learning framework for just-in-time defect prediction. In 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), pages 34–45. IEEE, 2019.

(2) EALR: Logistic Regression on handcrafted features

Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, "A large-scale empirical study of just-in-time quality assurance," Transactions on Software Engineering (TSE), pp. 757–773, 2012.

6 Approach:

6.1 General Picture

There are 2 two dataset Openstack and Qt collected from 2 different projects, for binary classification of buggy commits, they are all extremely imbalanced, Openstack (10658 negative samples, 1616 positive samples), Qt (23148 negative samples, 2002 positive samples), each sample is a GIT commit and positive sample is buggy. Previous works all defined a model and trained separately on the two datasets. My direction is by combining the two datasets together to create more positive samples for training. Because they share the same features (35 handcrafted features), there would not be any feature conflict. The underlying nature of the two datasets combined may offer new findings. I will train on the combined training datasets and test on two datasets separately, to see whether my result will reach the level of previous work. There are four central components in my work, including (1) sampling (not much explored by previous work), (2) traditional classification models (3) ensemble learning (4) feed forward deep neural network

6.2 Sampling and traditional classification models

Because the data is highly imbalanced, where only 10% of the data is positive, it is of critical importance that we sample the data, however, there are miscellaneous sampling method, I applied 6 different oversampling methods, 7 different undersampling methods, and three over and under sampling combined methods. The method that yields the best result later is SMOTE and SMOTE+Edited Nearest Neighbors. Details of those methods will be discussed in the Experiment section. The model for training selected is SVM (Support Vector Machine) based on its performance on all the sampling methods compared with other 6 models including Logistic Regression, Random Forest, etc. After deciding using SMOTE and SVM, I did a hyperparameter tuning with cross-validation on SVM to get the current best result.

6.3 Ensemble Learning

To better improve the result, I decided to use ensemble learning to create a vote classifier, to get the best models for the vote classifier. I did hyperparameter tuning with cross-validation for 13 classifier models including XGBoost, AdaBoost, etc. to get their best parameters. The final selected models include KNN, Stochastic gradient descent classifier, ridge classifier, gradient boost, AdaBoost, and XGBoost. SVM is not selected, because the time complexity for using SVM is just not practical for large dataset that has more than 50k samples. The final result is similar to SVM. However, just out of curiosity, I tested on XGBoost alone where positive class has class weight 2 and negative class has class weight 1, it achieved the best result for now.

6.3 Feed Forward Neural Network

Lastly, I constructed a feed forward neural network using Pytorch, Pytorch offers great versatility for manipulating the structure of your neural network compared with Sklearn mlpClassifier. And the model was run on GPU using Google Colab for faster computation. After a comprehensive hyperparameter tuning, two different neural network structures achieved the best results for the two datasets separately.

7 Experiment: (All the results shown below are testing results)

The order of the following subsections follows the order of my experiment and thinking process in exploring and getting the best results.

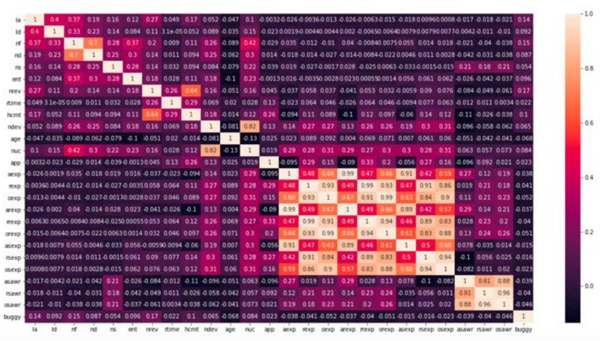
7.1 Preprocessing

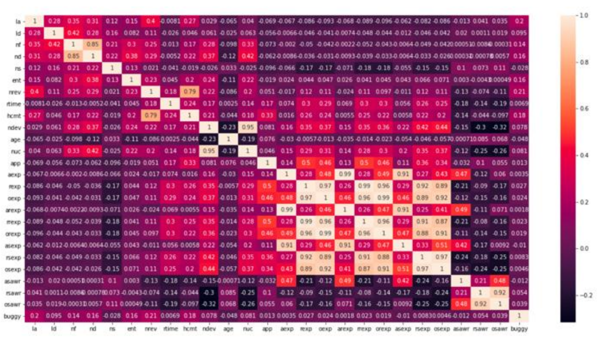
(1) Originally, the dataset has 35 features, 9 are dropped including 'commit_id', 'author_date', 'bugcount', 'fixcount', 'revd', 'churn', 'fix', 'bugdens', 'strata'. They are irrelevant features like bug counts, whether the bug is fixed, fix needed, bug density, lines code, they are not helpful in predicting whether the commit is buggy or not.

(2) A small number of rows that have empty values are dropped, for Openstack (1329 rows out of 25150), for Qt (333 rows out of 12374). After dropping these empty values rows, the two datasets remain the original imbalance shape, for Openstack {0: 21838, 1: 1983}, for Qt {0: 10428, 1: 1613}.

(3) I use one hot vector encoding to convert the True False column ‘self’ into two columns

(4) I use Pearson correlation to remove highly correlated features, the heatmap of the correlation for Openstack is shown on the left. As we can see, there is a region that have features highly correlated together. I removed those with a correlation above 0.8, which are {'osexp', 'nuc', 'orexp', 'rsexp', 'oexp', 'arexp', 'asexp', 'rrexp', 'rsawr', 'osawr'}.





The heatmap of the correlation for Qt is shown on the right.

Not surprisingly, the heat maps of two datasets show similar pattern, indicating the two datasets share similar intrinsic structure. The highly correlate features are {'osexp', 'nuc', 'orexp', 'rsexp', 'oexp', 'arexp', 'asexp', 'rrexp', 'nd', 'osawr'}, they are all selected safely, meaning at least one of the correlated features in the relation is not selected. Both datasets have 9 highly correlated features, 8 of them are the same, I dropped those 8 features. The correlation values of all other features range from -0.06 to 0.16 to our target value, meaning no features are highly correlated to our target value 'buggy'.

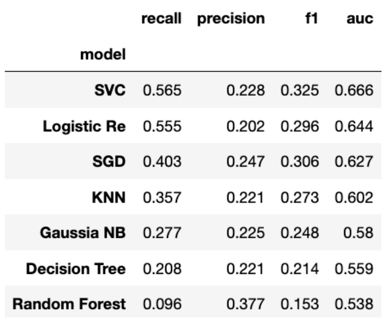
(5) The two datasets are normalized separately using Standard Scaler and then concatenated.

7.2 Oversampling

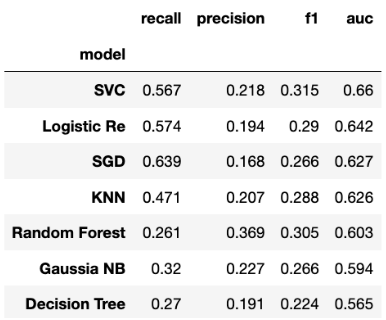
(1) First, I selected 7 traditional classification model for baseline testing without sampling, AUC score is the primary metric we cared about in this field of work. Those 7 models with their AUC score are Logistic Regression (0.51), Gaussian Naïve Bayes (0.555), Stochastic Gradient Descent (0.503), KNN (0.545), Decision Tree (0.549), Random Forest (0.519), SVM (0.501). As we know, AUC score around 0.5 is bad, basically equals to guessing. However, the accuracies are all about 0.9, which are as expected for such an imbalanced dataset.

(2) For 6 oversampling methods, I selected random oversampling, SMOTE, Borderline SMOTE, KMeans SMOTE, SVM Smote, ADASYN (Adaptive Synthetic Sampling). For each method, I tested it on the 7 basic classification models above for fair comparisons, and it also offers insight into which classification model may perform the best.

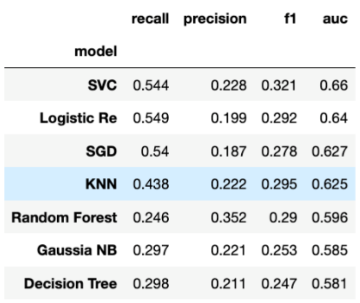
Random Oversampling SMOTE Borderline SMOTE



model	recall	precision	f1	auc
SVC	0.565	0.228	0.325	0.666
Logistic Re	0.555	0.202	0.296	0.644
SGD	0.403	0.247	0.306	0.627
KNN	0.357	0.221	0.273	0.602
Gaussia NB	0.277	0.225	0.248	0.58
Decision Tree	0.208	0.221	0.214	0.559
Random Forest	0.096	0.377	0.153	0.538

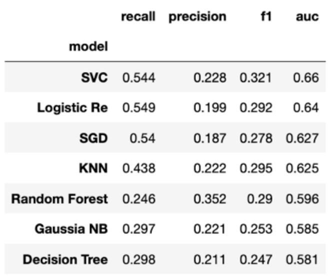


model	recall	precision	f1	auc
SVC	0.567	0.218	0.315	0.66
Logistic Re	0.574	0.194	0.29	0.642
SGD	0.639	0.168	0.266	0.627
KNN	0.471	0.207	0.288	0.626
Random Forest	0.261	0.369	0.305	0.603
Gaussia NB	0.32	0.227	0.266	0.594
Decision Tree	0.27	0.191	0.224	0.565

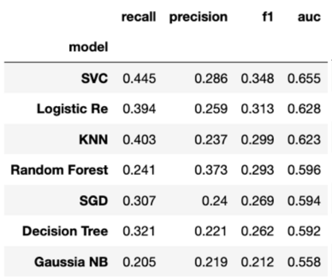


model	recall	precision	f1	auc
SVC	0.544	0.228	0.321	0.66
Logistic Re	0.549	0.199	0.292	0.64
SGD	0.54	0.187	0.278	0.627
KNN	0.438	0.222	0.295	0.625
Random Forest	0.246	0.352	0.29	0.596
Gaussia NB	0.297	0.221	0.253	0.585
Decision Tree	0.298	0.211	0.247	0.581

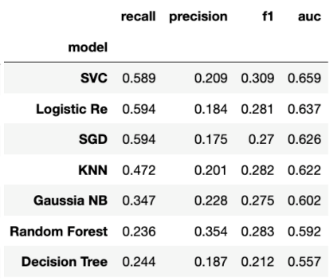
KMeans SMOTE SVM Smote ADASYN



model	recall	precision	f1	auc
SVC	0.544	0.228	0.321	0.66
Logistic Re	0.549	0.199	0.292	0.64
SGD	0.54	0.187	0.278	0.627
KNN	0.438	0.222	0.295	0.625
Random Forest	0.246	0.352	0.29	0.596
Gaussia NB	0.297	0.221	0.253	0.585
Decision Tree	0.298	0.211	0.247	0.581



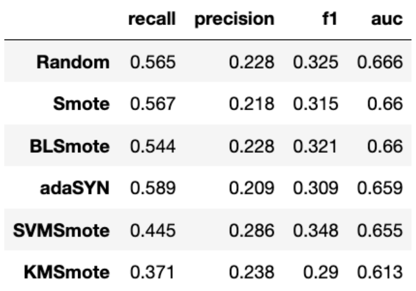
model	recall	precision	f1	auc
SVC	0.445	0.286	0.348	0.655
Logistic Re	0.394	0.259	0.313	0.628
KNN	0.403	0.237	0.299	0.623
Random Forest	0.241	0.373	0.293	0.596
SGD	0.307	0.24	0.269	0.594
Decision Tree	0.321	0.221	0.262	0.592
Gaussia NB	0.205	0.219	0.212	0.558



model	recall	precision	f1	auc
SVC	0.589	0.209	0.309	0.659
Logistic Re	0.594	0.184	0.281	0.637
SGD	0.594	0.175	0.27	0.626
KNN	0.472	0.201	0.282	0.622
Gaussia NB	0.347	0.228	0.275	0.602
Random Forest	0.236	0.354	0.283	0.592
Decision Tree	0.244	0.187	0.212	0.557

As we can see, SVC performed the best in each sampling method.

Below is the SVC performance for each oversampling method, we see except KMeans SMOTE, all method performed similarly in AUC score. I will use SMOTE, as it’s the most popular one.



	recall	precision	f1	auc
Random	0.565	0.228	0.325	0.666
Smote	0.567	0.218	0.315	0.66
BLSmote	0.544	0.228	0.321	0.66
adaSYN	0.589	0.209	0.309	0.659
SVMSmote	0.445	0.286	0.348	0.655
KMSmote	0.371	0.238	0.29	0.613

7.3 Hyperparamter tuning and cross validation on SVM and Logistic Regression

Since SVM and logistic Regression are the best performing model under all oversampling methods, hyperparameter tuning with 5-fold cross validation were performed on each model.

(1) SVM: best parameters: {'C': 1000, 'gamma': 0.001, 'kernel': 'rbf'}

Qt: recall: 0.454, precision: 0.194, f1: 0.272, AUC: 0.637

Openstack: recall: 0.747, precision: 0.249, f1: 0.374, AUC: 0.674

(2) Logistic Regression: {'C': 1000.0, 'penalty': 'l2'}

Qt: recall: 0.579, precision: 0.157, f1: 0.246, AUC: 0.64

Openstack: recall: 0.581, precision: 0.274, f1: 0.373, AUC: 0.654

We can see that SVM works better on Openstack by a large margin and Logistic Regression works better on Qt just slightly.

7.4 Calibrated Probability

I selected two techniques (Platt Scaling, Isotonic Regression) for calibrating probabilities, since they are one kind of the methods that deal with imbalanced data. I trained on oversampled data using SVM as the default embedding model.

(1) Platt Scaling:

Qt: recall: 0.486, precision: 0.168, f1: 0.250, AUC: 0.628

Openstack: recall: 0.749, precision: 0.253, f1: 0.378, AUC: 0.678

(2) Isotonic Regression:

Qt: recall: 0.488, precision: 0.175, f1: 0.258, AUC: 0.634

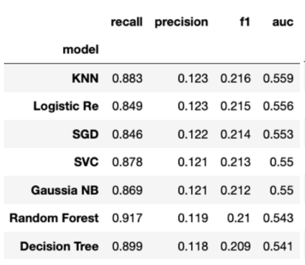
Openstack: recall: 0.741, precision: 0.260, f1: 0.385, AUC: 0.683

Both methods increased the AUC score for Openstack, but not for Qt.

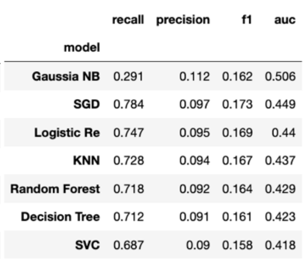
7.5 Undersampling

For fair comparison with section 7.2, I also tried 7 undersampling methods including NearMiss version 1, NearMiss version 2, NearMiss version 3, Tomek Links, ENN (Edited Nearest Neighbors Rule), On-sided Selection, and Neighborhood Cleaning Rule on the same 7 models, but the results are not good. Most AUC scores are below 0.6. They are shown below

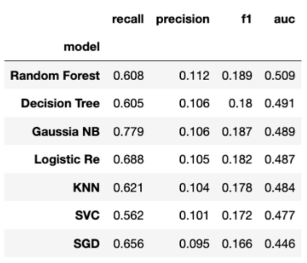
NearMiss1. NearMiss2. NearMiss 3 Tomek Links



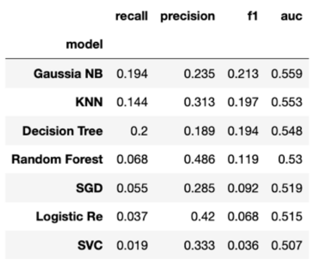
model	recall	precision	f1	auc
KNN	0.883	0.123	0.216	0.559
Logistic Re	0.849	0.123	0.215	0.556
SGD	0.846	0.122	0.214	0.553
SVC	0.878	0.121	0.213	0.55
Gaussia NB	0.869	0.121	0.212	0.55
Random Forest	0.917	0.119	0.21	0.543
Decision Tree	0.899	0.118	0.209	0.541



model	recall	precision	f1	auc
Gaussia NB	0.291	0.112	0.162	0.506
SGD	0.784	0.097	0.173	0.449
Logistic Re	0.747	0.095	0.169	0.44
KNN	0.728	0.094	0.167	0.437
Random Forest	0.718	0.092	0.164	0.429
Decision Tree	0.712	0.091	0.161	0.423
SVC	0.687	0.09	0.158	0.418

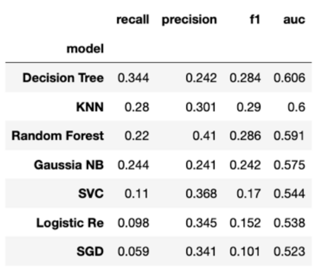


model	recall	precision	f1	auc
Random Forest	0.608	0.112	0.189	0.509
Decision Tree	0.605	0.106	0.18	0.491
Gaussia NB	0.779	0.106	0.187	0.489
Logistic Re	0.688	0.105	0.182	0.487
KNN	0.621	0.104	0.178	0.484
SVC	0.562	0.101	0.172	0.477
SGD	0.656	0.095	0.166	0.446

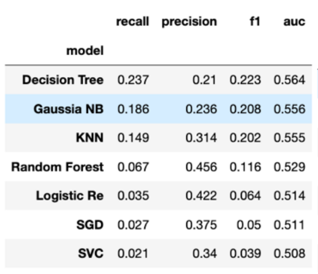


model	recall	precision	f1	auc
Gaussia NB	0.194	0.235	0.213	0.559
KNN	0.144	0.313	0.197	0.553
Decision Tree	0.2	0.189	0.194	0.548
Random Forest	0.068	0.486	0.119	0.53
SGD	0.055	0.285	0.092	0.519
Logistic Re	0.037	0.42	0.068	0.515
SVC	0.019	0.333	0.036	0.507

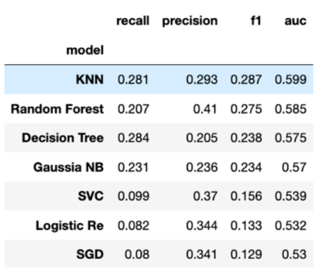
ENN One-sided Selection Neighborhood Cleaning Rule



model	recall	precision	f1	auc
Decision Tree	0.344	0.242	0.284	0.606
KNN	0.28	0.301	0.29	0.6
Random Forest	0.22	0.41	0.286	0.591
Gaussia NB	0.244	0.241	0.242	0.575
SVC	0.11	0.368	0.17	0.544
Logistic Re	0.098	0.345	0.152	0.538
SGD	0.059	0.341	0.101	0.523



model	recall	precision	f1	auc
Decision Tree	0.237	0.21	0.223	0.564
Gaussia NB	0.186	0.236	0.208	0.556
KNN	0.149	0.314	0.202	0.555
Random Forest	0.067	0.456	0.116	0.529
Logistic Re	0.035	0.422	0.064	0.514
SGD	0.027	0.375	0.05	0.511
SVC	0.021	0.34	0.039	0.508



model	recall	precision	f1	auc
KNN	0.281	0.293	0.287	0.599
Random Forest	0.207	0.41	0.275	0.585
Decision Tree	0.284	0.205	0.238	0.575
Gaussia NB	0.231	0.236	0.234	0.57
SVC	0.099	0.37	0.156	0.539
Logistic Re	0.082	0.344	0.133	0.532
SGD	0.08	0.341	0.129	0.53

However, one interesting finding is that for undersampling, there is no unanimous best model.

7.6 XGBoost

One of the downsides of SVM is its time complexity, however, using XGBoost classifier, I achieved similar performance but with less time. SVM takes 2min44s to train, while XGBoost classifier is a decision tree based gradient descent method, it only takes 2.6s to train.

XGBoost can get good result without oversampling by setting the class weight of positive class according to its proportion with negative class. Here positive class weight is 10, negative is 1

(1) XGBoost + unsampled data:

Qt:	recall: 0.385, precision: 0.216, f1: 0.276, AUC: 0.625
Openstack:	recall: 0.537, precision: 0.294, f1: 0.380, AUC: 0.654

(2) XGBoost + oversampled data:

Qt:	recall: 0.505, precision: 0.179, f1: 0.264, AUC: 0.642
Openstack:	recall: 0.592, precision: 0.261, f1: 0.363, AUC: 0.648

(3) Hyperparameter tuning with 5-fold cross validation XGBoost + oversampled data:

Best Parameters: {'learning_rate': 0.1, 'max_depth': 6, 'n_estimators': 200, 'subsample': 0.6}

Qt: recall: 0.620, precision: 0.181, f1: 0.280, AUC: 0.676

Openstack: recall: 0.697, precision: 0.245, f1: 0.363, AUC: 0.658

(4) XGBoost + Best Parameter + Isotonic Regression(Probability Calibrating)+ oversampled data:

Qt: recall: 0.154, precision: 0.362, f1: 0.216, AUC: 0.564

Openstack: recall: 0.292, precision: 0.408, f1: 0.34, AUC: 0.608

As we can see, after parameter tuning, XGBoost can get the best AUC result on Qt so far.

7.7 Oversampling & Undersampling combined

Now I have set the best candidate as XGBoost with the tuned parameters. I tried three other sampling methods which combine oversampling and undersampling together.

(1) XGBoost + Random Oversampling (sampling_strategy=0.25) + Random Undersampling (sampling_strategy=0.5):

Qt: recall: 0.466, precision: 0.180, f1: 0.260, AUC: 0.631

Openstack: recall: 0.554, precision: 0.260, f1: 0.354, AUC: 0.637

(2) XGBoost + Tomek Links + SMOTE:

Qt: recall: 0.516, precision: 0.217, f1: 0.306, AUC: 0.645

Openstack: recall: 0.540, precision: 0.258, f1: 0.349, AUC: 0.632

(3) XGBoost + Edited Nearest Neighbors + SMOTE:

Qt: recall: 0.522, precision: 0.186, f1: 0.274, AUC: 0.652

Openstack: recall: 0.672, precision: 0.258, f1: 0.373, AUC: 0.665

As we can see, the third ENN+SMOTE achieves similar results on SMOTE alone, better Openstack AUC score, worse Qt AUC score. So, I stuck with using SMOTE only.

7.8 Ensemble Learning and Vote Classifier

To build a vote classifier, I need the best models there are, so I included several ensemble learning classifiers and did Hyperparamter tuning with 3-fold cross-validation on 13 classifiers.

The parameters grid details can be looked up inside the code.

model	Qt AUC	Openstack AUC	best parameters
Logistic Regression	0.635	0.657	{'C': 0.01, 'penalty': 'l1', 'solver': 'liblinear'}
Random Forest	0.567	0.613	{'bootstrap': False, 'max_features': 'auto', 'n_estimators': 1000}
Decision Tree	0.574	0.596	{'criterion': 'entropy', 'max_depth': None, 'max_features': 4, 'min_samples_leaf': 7}
KNN	0.623	0.636	{'metric': 'manhattan', 'n_neighbors': 9, 'weights': 'distance'}
Stochastic Gradient Descent	0.630	0.665	{'alpha': 0.001, 'penalty': 'l2'}
Gaussian Naïve Bayes	0.559	0.564	{'var_smoothing': 1.0}
Ridge Classifier	0.619	0.645	{alpha: 0.9}
Bagging Classifier	0.573	0.610	{'n_estimators': 100}
Stochastic Gradient Boosting	0.618	0.689	using default
Adaboost	0.645	0.669	using default
Extra Tree Classifier	0.574	0.623	using default
Hist Gradient Boosting	0.581	0.621	using default
XGBoost	0.670	0.655	{'learning_rate': 0.1, 'max_depth': 6, 'n_estimators': 200, 'subsample': 0.6}

Final selected models include KNN, Stochastic gradient descent classifier, ridge classifier, gradient boost, AdaBoost, and XGBoost, since they have AUC score for both dataset above 0.6.

Vote classifier results:

Qt: recall: 0.500, precision: 0.190, f1: 0.275, AUC: 0.648

Openstack: recall: 0.612, precision: 0.293, f1: 0.396, AUC: 0.675

When compared with our best candidate XGBoost (Openstack 0.670, Qt 0.655), we can see that the vote classifier has a higher AUC for Openstack (0.675), but a lower score for Qt (0.648).

7.9 Feed Forward Neural Network

To give the AUC score a last try, I constructed a feed forward neural network.

ANN structure:

Linear Layer(input 17, output 100) -> Batch Normalization Layer -> Relu Activation -> Dropout Layer (0.2 dropout rate) -> Linear Layer(input 100, output 2) -> SoftMax Layer for final prediction

Loss function: Cross Entropy; Epochs: 5

Optimizer: AdamW with learning rate 0.001, l2 regularization alpha 0.001, Batch Size 64

(1) ANN with oversampled data, AUC 0.5, worst performance, cannot distinguish, probably because ANN is good at overfitting.

Qt:	recall: 1.000, precision: 0.087, f1: 0.161, AUC: 0.5
Openstack:	recall: 1.000, precision: 0.151, f1: 0.262, AUC: 0.5

As we can see, recall is all 1, but precision is very low, meaning it categorizes much of the samples as buggy.

(2) ANN without sampling data

Set positive class weight to 10:1 to negative class in loss function

l2 regularization alpha 0.01

Qt:	recall: 0.486, precision: 0.198, f1: 0.281, AUC: 0.649
Openstack:	recall: 0.741, precision: 0.269, f1: 0.395, AUC: 0.692

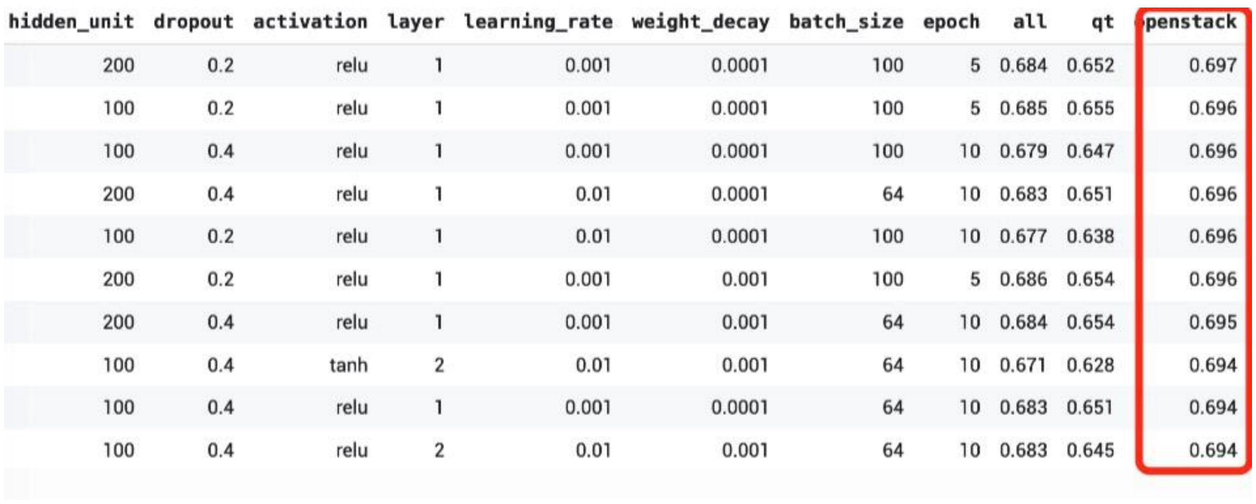
Clearly, we raised the AUC of Openstack to the highest 0.692 so far, and AUC for Qt 0.649 is not as good as XGBoost (0.670)

(3) Hyperparameter tuning for ANN

I constructed a hyperparameter tuning for different structure of the ANN, a total 256 experiments are run with the following changing parameters

Hidden unit number= [100,200], dropout rate=[0.2,0.4], activation function=['relu','tanh'], number of feed forward layer=[1,2], learning rate=[0.01,0.001], l2 alpha=[1e-3,1e-4], batch size=[64,100], epoch=[5,10]

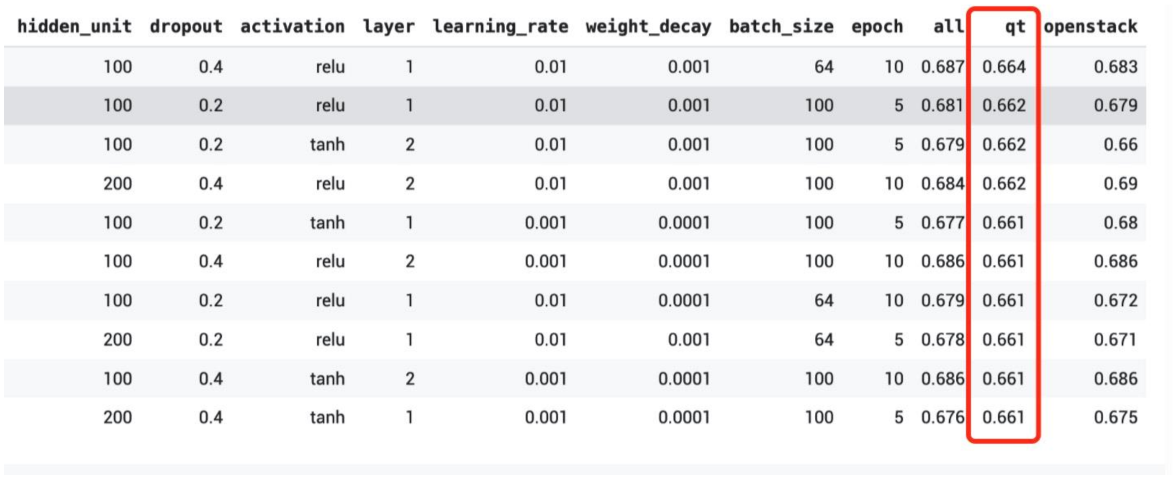
Top 10 results for Openstack (AUC score):



hidden_unit	dropout	activation	layer	learning_rate	weight_decay	batch_size	epoch	all	qt	penstack
200	0.2	relu	1	0.001	0.0001	100	5	0.684	0.652	0.697
100	0.2	relu	1	0.001	0.0001	100	5	0.685	0.655	0.696
100	0.4	relu	1	0.001	0.0001	100	10	0.679	0.647	0.696
200	0.4	relu	1	0.01	0.0001	64	10	0.683	0.651	0.696
100	0.2	relu	1	0.01	0.0001	100	10	0.677	0.638	0.696
200	0.2	relu	1	0.001	0.001	100	5	0.686	0.654	0.696
200	0.4	relu	1	0.001	0.001	64	10	0.684	0.654	0.695
100	0.4	tanh	2	0.01	0.001	64	10	0.671	0.628	0.694
100	0.4	relu	1	0.001	0.0001	64	10	0.683	0.651	0.694
100	0.4	relu	2	0.01	0.001	64	10	0.683	0.645	0.694

AUC score of 0.7 seems like the bottleneck I can't overcome for Openstack, but still 0.697 is the best result across my all endeavors.

Top 10 results for Qt (AUC score):



hidden_unit	dropout	activation	layer	learning_rate	weight_decay	batch_size	epoch	all	qt	openstack
100	0.4	relu	1	0.01	0.001	64	10	0.687	0.664	0.683
100	0.2	relu	1	0.01	0.001	100	5	0.681	0.662	0.679
100	0.2	tanh	2	0.01	0.001	100	5	0.679	0.662	0.66
200	0.4	relu	2	0.01	0.001	100	10	0.684	0.662	0.69
100	0.2	tanh	1	0.001	0.0001	100	5	0.677	0.661	0.68
100	0.4	relu	2	0.001	0.0001	100	10	0.686	0.661	0.686
100	0.2	relu	1	0.01	0.0001	64	10	0.679	0.661	0.672
200	0.2	relu	1	0.01	0.001	64	5	0.678	0.661	0.671
100	0.4	tanh	2	0.001	0.0001	100	10	0.686	0.661	0.686
200	0.4	tanh	1	0.001	0.0001	100	5	0.676	0.661	0.675

The best result 0.664 is slightly worse than XGBoost 0.670.

8 Discussion of results:

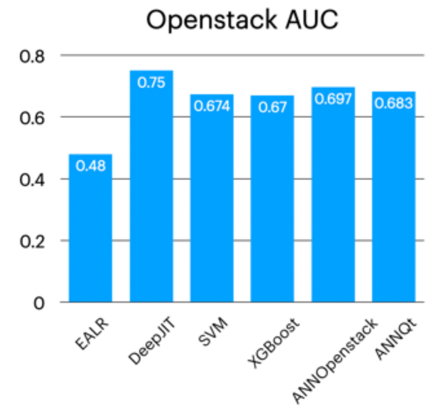
I will compare several of my best candidate with the state-of-the-art works EALR and DeepJIT

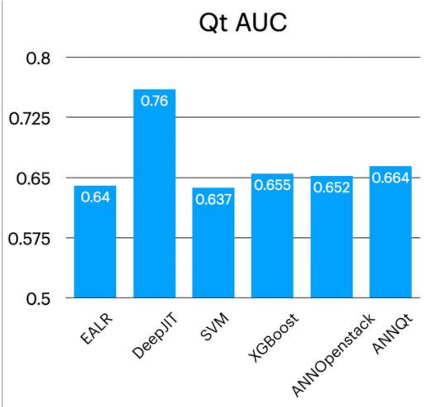
My model 1: SVM, trained on SMOTE oversampled data

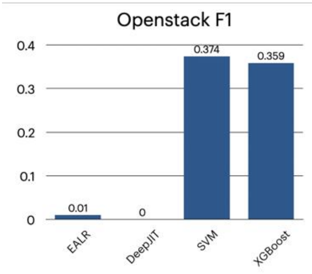
My model 2: XGBoost, trained on SMOTE oversampled data

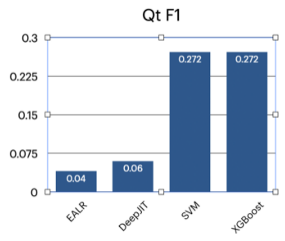
My model 3: ANNOpenstack, trained on unsampled data with positive class weight set to 10:1, with best AUC score on Openstack dataset

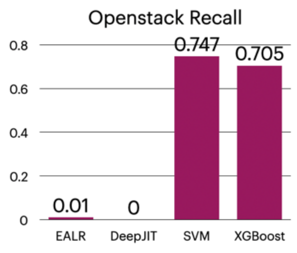
My model 3: ANNQt, trained on unsampled data with positive class weight set to 10:1, with best AUC score on Qt dataset

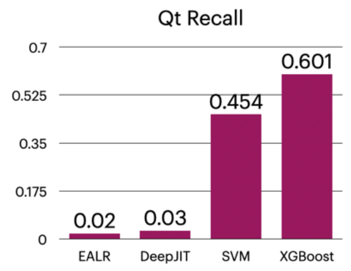


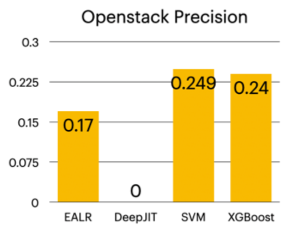


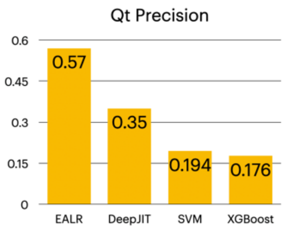












The most important metric AUC score: for Openstack, 4 models all outperform EALR a lot, but cannot catch up with DeepJIT, for Qt, 4 models all have a huge gap when compared with DeepJIT. For f1, 2 models are much better than both EALR and DeepJIT, DeepJIT does not report f1, recall and precision for Openstack. For recall, 2 models still outperform EALR and DeepJIT. For Precision, 2 models cannot compete with EALR and DeepJIT for Qt but outperform EALR on Openstack. Basically, state-of-the-art models have very good precision but poor recall, meaning they probably catches those commits that can be easily classified as buggy, but they cannot identify much of them. The state-of-the-art woks using code commits messages with NLP method still is the best leader in this field.

Generally, although combining the two datasets dose not yield great results that can outperform the state-of-the art work in AUC score, the gap is not big, given enough training resources and time, I am confident to construct a better neural network structure. Also, using handcrafted features are way much faster than NLP method, because NLP method will generate a long feature vector.

9 Lessons Learned:

(1) Imbalance problem:

Most of the classification problems in real life nowadays are imbalanced, buggy commit included. There are many ways to deal with imbalanced dataset. Oversampling, undersampling, over and under combined sampling, calibrated probabilities. In my experiment, oversampling performs the best (SMOTE), followed by over and under combined sampling (SMOTE+ENN), the worst method is undersampling, calibrated probabilities can improve for some models (SVM) but not all, XGBoost gets worse after calibrated probabilities. Another great method is assigning high class weight for the minority class, it can help XGBoost and ANN to perform to top-notch results without oversampling, if the dataset is huge and oversampling is costly, this is the best-case scenario.

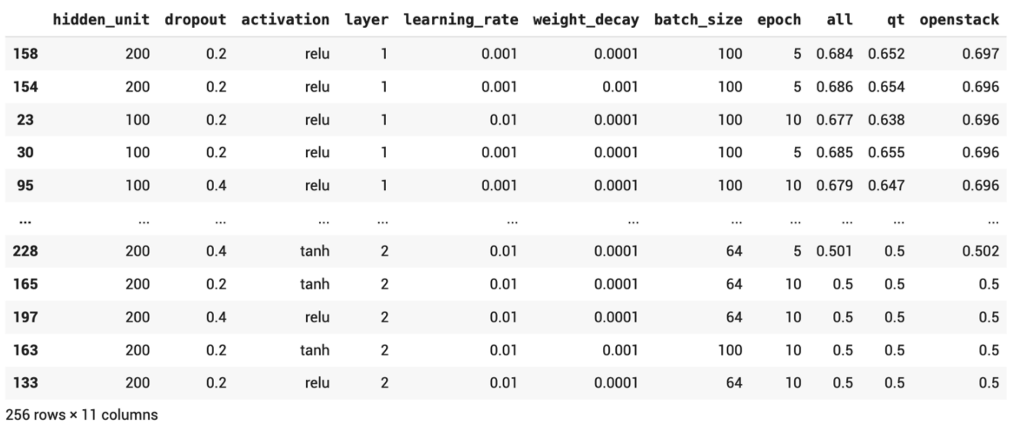
(2) Time complexity of certain algorithm:

SVM can have good result, but the minimal time complexity is quadratic in accordance with the sample size, initially, I am doing a grid search for hyper parameter tuning for SVM, when the regularization parameter gets big, the program never finished, I must interrupt the kernel to break it. So, when the size of the dataset is huge, I would not recommend using SVM.

(3) Ensemble learning:

After doing a hyper parameter tuning on every possible model, it turns out that ensemble learning models often outperform traditional methods. Those models include XGBoost, Stochastic Gradient Boosting, Adaboost.

(4) Neural Network Hyperparameters Matters a Lot



	hidden_unit	dropout	activation	layer	learning_rate	weight_decay	batch_size	epoch	all	qt	openstack
158	200	0.2	relu	1	0.001	0.0001	100	5	0.684	0.652	0.697
154	200	0.2	relu	1	0.001	0.001	100	5	0.686	0.654	0.696
23	100	0.2	relu	1	0.01	0.0001	100	10	0.677	0.638	0.696
30	100	0.2	relu	1	0.001	0.0001	100	5	0.685	0.655	0.696
95	100	0.4	relu	1	0.001	0.0001	100	10	0.679	0.647	0.696
...	...	...	...	...	...	...	...	...	...	...	...
228	200	0.4	tanh	2	0.01	0.0001	64	5	0.501	0.5	0.502
165	200	0.2	tanh	2	0.01	0.0001	64	10	0.5	0.5	0.5
197	200	0.4	relu	2	0.01	0.0001	64	10	0.5	0.5	0.5
163	200	0.2	tanh	2	0.01	0.001	100	10	0.5	0.5	0.5
133	200	0.2	relu	2	0.01	0.0001	64	10	0.5	0.5	0.5

256 rows × 11 columns

By looking at the top 5 best results, and the top 5 worst results from parameter tuning, we see that the combination of different parameters can create significant variations in the result. For example, the best result (1st row) and the worst result (last row) only differ in activation function, number of layers, batch size and epoch numbers.

10 Challenges:

When constructing a feed forward neural network, it is hard to know what the best structure and parameters are. I tested 256 combinations from 8 parameters, however for structure, you can set different number of layers, with or without batch normalization, drop out, and there are different activation functions that can be applied differently at different layers. There are also the class weight, the regularization parameter, and the learning rate for loss function and optimizer, which all have significant influence over the results. Small batch size may yield good result but is extremely slow. The right epoch number is also critical for preventing overfitting. I hope I can learn more about how to create the best deep learning structures based on temporary results. Because for our result, I have confidence in its AUC score catching up with the state-of-art-work that use lengthy code commits for prediction.

Ali's part:

State-of-the-art Work:

T. Hoang, H. Khanh Dam, Y. Kamei, D. Lo and N. Ubayashi, "DeepJIT: An End-to-End Deep Learning Framework for Just-in-Time Defect Prediction," 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), 2019, pp. 34-45, doi: 10.1109/MSR.2019.00016.

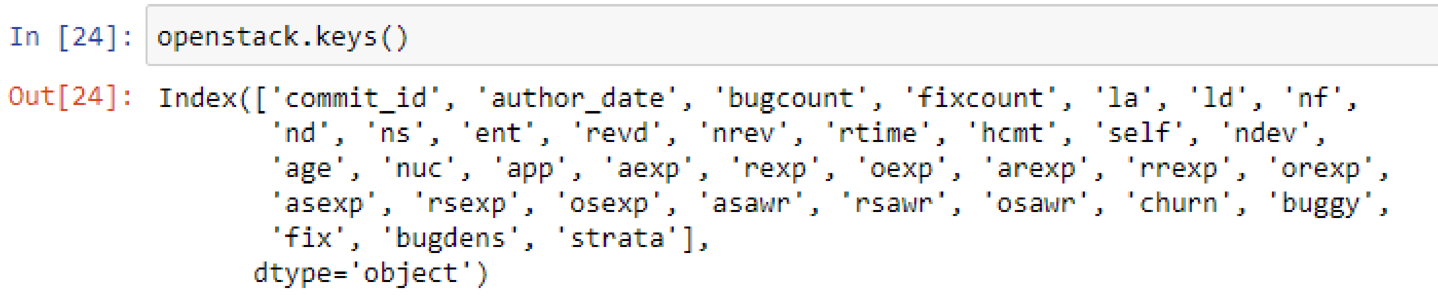
Our work is to be compared to the baseline work of Hoang and his team.

The Dataset:

Let us first introduce the dataset:

I will be working with the openstack dataset, since the 2 most widely used datasets are the QT dataset, and the openstack dataset.

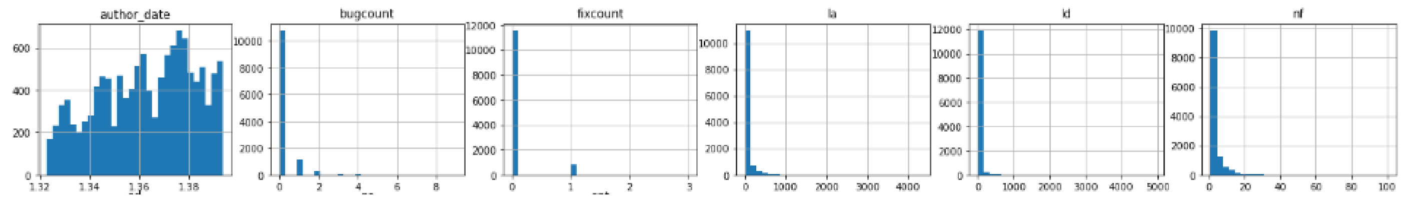
The openstack dataset consists of 36 columns, namely:



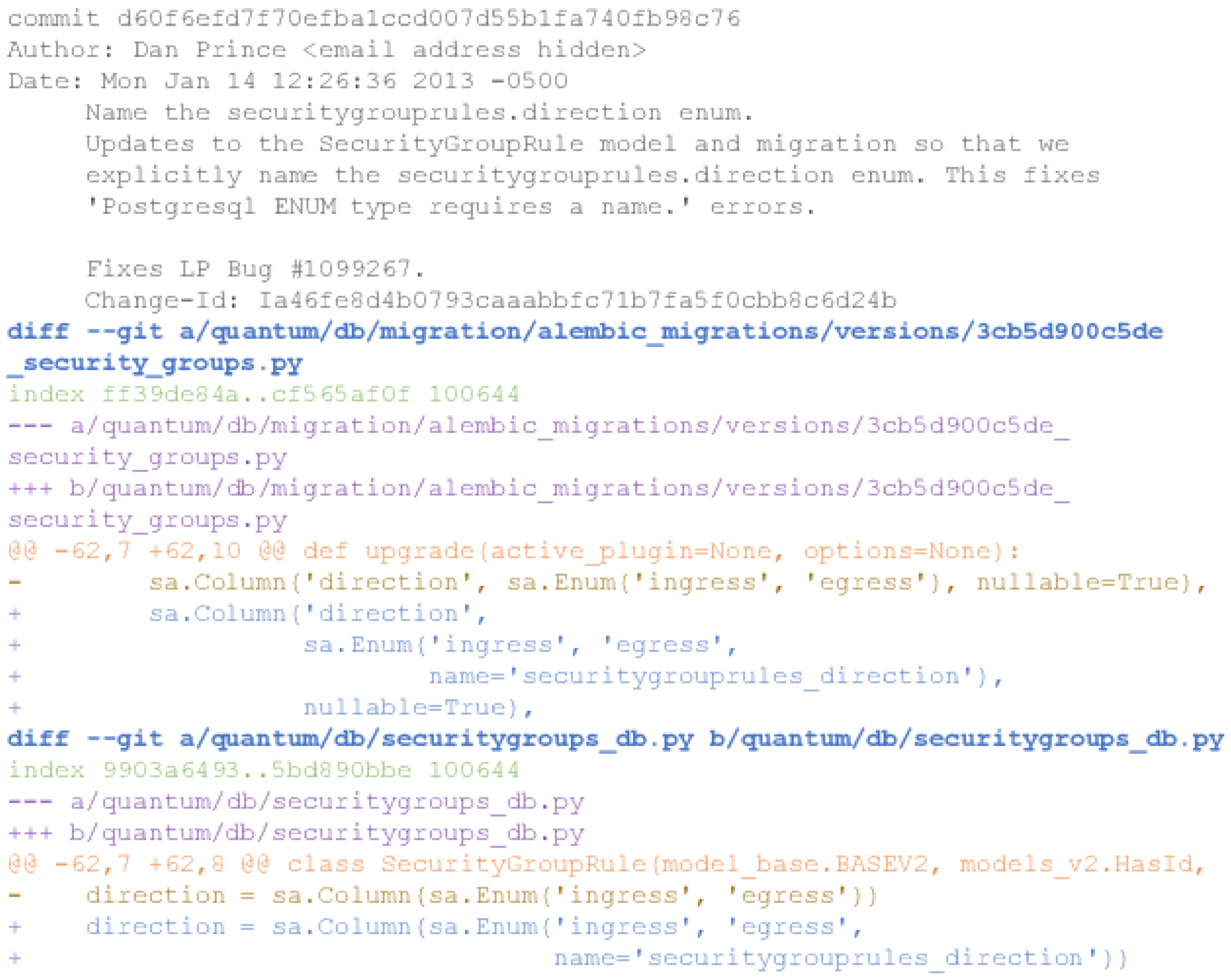
```
In [24]: openstack.keys()

Out[24]: Index(['commit_id', 'author_date', 'bugcount', 'fixcount', 'la', 'ld', 'nf',
                'nd', 'ns', 'ent', 'revd', 'nrev', 'rtime', 'hcmt', 'self', 'ndev',
                'age', 'nuc', 'app', 'aexp', 'rexp', 'oexp', 'arexp', 'rrexp', 'orexp',
                'asexp', 'rsexp', 'osexp', 'asawr', 'rsawr', 'osawr', 'churn', 'buggy',
                'fix', 'bugdens', 'strata'],
               dtype='object')
```

Histograms of some of the columns:



Here is an example of a commit:



```
commit d60f6efd7f70efba1ccd007d55b1fa740fb98c76
Author: Dan Prince <email address hidden>
Date: Mon Jan 14 12:26:36 2013 -0500
     Name the securitygrouprules.direction enum.
     Updates to the SecurityGroupRule model and migration so that we
     explicitly name the securitygrouprules.direction enum. This fixes
     'Postgresql ENUM type requires a name.' errors.

     Fixes LP Bug #1099267.
     Change-Id: Ia46fe8d4b0793caaabbfc71b7fa5f0cbb8c6d24b
diff --git a/quantum/db/migration/alembic_migrations/versions/3cb5d900c5de
_security_groups.py
index ff39de84a..cf565af0f 100644
--- a/quantum/db/migration/alembic_migrations/versions/3cb5d900c5de_
security_groups.py
+++ b/quantum/db/migration/alembic_migrations/versions/3cb5d900c5de_
security_groups.py
@@ -62,7 +62,10 @@ def upgrade(active_plugin=None, options=None):
-         sa.Column('direction', sa.Enum('ingress', 'egress'), nullable=True),
+         sa.Column('direction',
+                   sa.Enum('ingress', 'egress',
+                           name='securitygrouprules_direction'),
+                   nullable=True),
diff --git a/quantum/db/securitygroups_db.py b/quantum/db/securitygroups_db.py
index 9903a6493..5bd890bbe 100644
--- a/quantum/db/securitygroups_db.py
+++ b/quantum/db/securitygroups_db.py
@@ -62,7 +62,8 @@ class SecurityGroupRule(model_base.BASEV2, models_v2.HasId,
-    direction = sa.Column(sa.Enum('ingress', 'egress'))
+    direction = sa.Column(sa.Enum('ingress', 'egress',
+                                  name='securitygrouprules_direction'))
```

Class Imbalance:

The OpenStack dataset is extremely imbalanced, albeit not as imbalanced as the QT dataset – but imbalanced, nonetheless. Now, of course, class imbalance could become a huge obstacle if we want to properly classify using supervised learning, so we have to find a way to fix that. There are ways of balancing the class such as TomekLinks, Synthetic Minority Oversampling Technique, among others, but I will start by simply dropping all NA values, since they are useless. To me, the commit_ID would serve to be somewhat useful in that we can determine the buggy commit by identification number once it is found through the model. The openstack dataset contains 10658 negative samples, and 1616 positive samples (otherwise known as clean and buggy commits, respectively). Exploring the openstack dataset, it is similar to the QT datset in that it has 35 features, and 1 target value, that value being whether the commit is buggy or not.

Pre-Processing:

When pre-processing, we used undersampling for the openstack dataset, as well, and in the PCA part, we went with using 16 features rather than all of them, for the feature reduction step. Lastly, polynomial features were implemented, as well, which extended our 16 features to a total of 58 features.

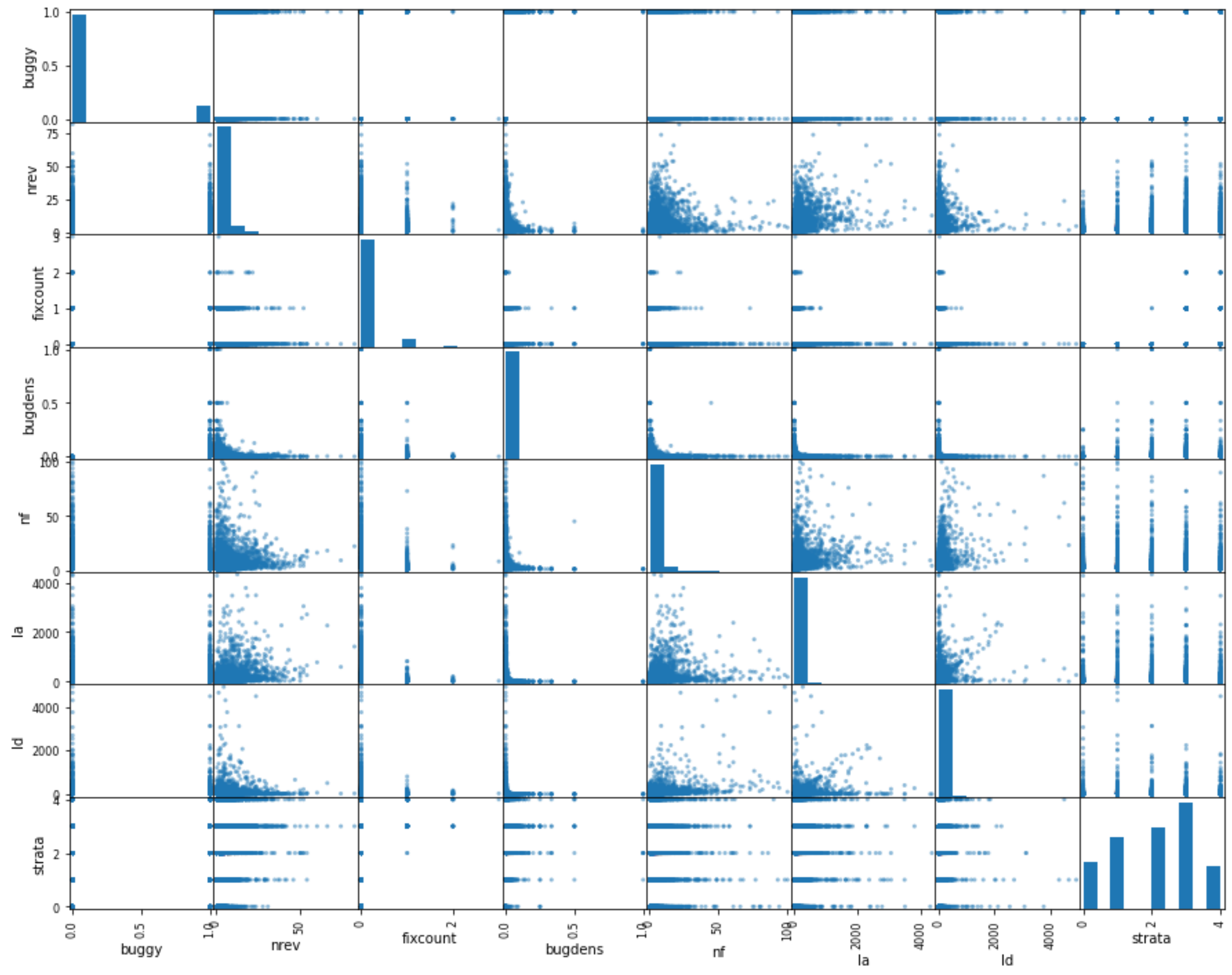
Methodologies:

I will be implementing and observing the Logistic Regression and Random forest methods for the openstack dataset, both of which fall under the category of supervised machine learning. Future work may include K-means, Deep neural networks, and other types of learning.

In the end, maybe some of the accuracies were not as high as we would have liked, but it was a good experiment, nevertheless.

There were a few features that needed to be converted to integer types before being able to send them through training and testing models, for both Random Forest and Linear Regression. For the openstack dataset, it is no different from the QT dataset in terms of features and the target – both are the same. The difference lies within the number of records available within the dataset itself.

It would help to plot some of the data values to be able to visualize the relationships:



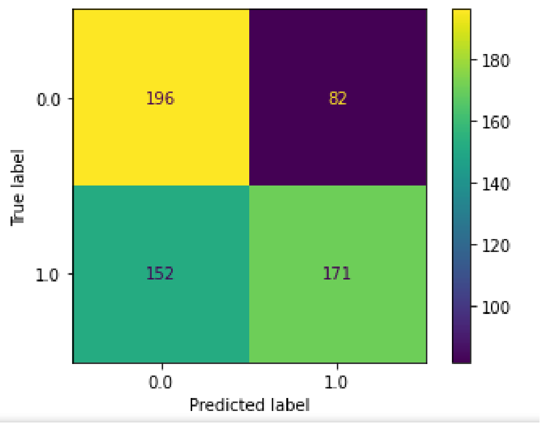
Evaluation:

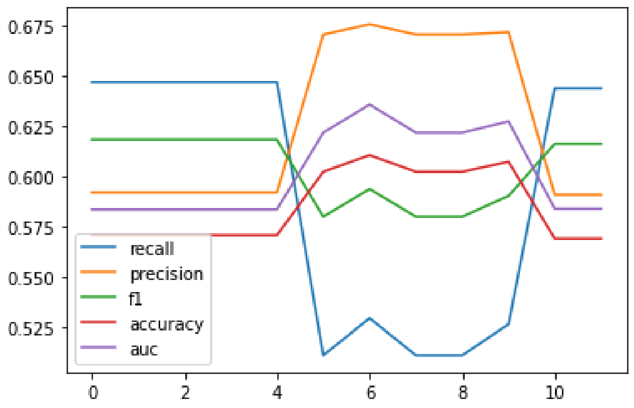
For evaluation, I will use an ROC-AOC curve to show the results, as well as its accuracy, precision, F1 score, and recall score. After experimenting with various parameters, the best accuracy obtained throughout the project was, depending on the approach:

Model	features	Accuracy(%)	F1(%)	AUC(%)
Logistic regression	PCA	59.7%	51.7%	69.5%
	Poly features	61.1%	59.4%	62.1%
Random forest	PCA	63.2%	62.6%	70.1%

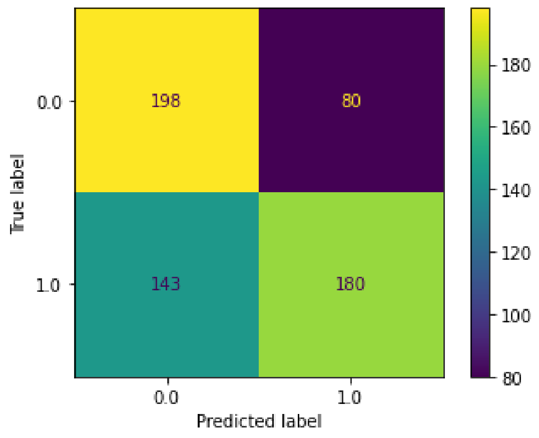
Here are the results based on the above:

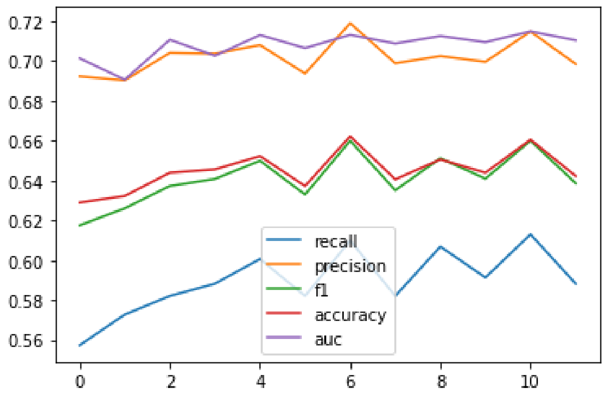
Logistic Regression:





Random Forest:





Takeaway from experimentation:

Firstly, the logistic regression overall seemed to have a slightly worse performance than the Random forest model, perhaps due to the parameters that were set. The random forest model seemed to work really well on a not-so-big dataset (openstack is about half the size of QT dataset). Because this is the case, and random forest is one of the more easier models to implement, that means that so far, we have attained more success using these ‘simpler’, or easier models.

Most of the papers that cover this topic claim that supervised machine learning is the best type of learning when it comes to software defect prediction – from the experiments done here, that does seem to be the case. The accuracy, F1, Recall, and precision are not too bad when it comes to scores.

If we can achieve similar results using both simple and non-simple models, this opens up many venues and opportunities for future experimentation with other models, such as Deep Neural Networks, and the like.

Future Work:

We can work on implementing Deep neural networks with the openstack dataset, as well as semi-supervised and reinforcement learning. We were able to experiment with the kmeans model for k = 2, but we can take that further and see the best value through more experimentation. Overall, there is a lot of room for growth in this field, as 60%~70% accuracies can always get better, whether it is through fine parameter tuning, utilizing different models, or finding a different approach.

Conclusion

In our project, we separately formalize JIT-SDP on different datasets with several methods. Simple supervised machine methods like RFC and LR perform better in dataset Qt, and it is easier to implement. DNN does not perform well when the data is too small. RFC with polynomial features gives an incredible result since too many features and too small dataset. We failed comparing to DeepJIT, but software metrics could make it much easier to access closed sources.

For the experiments trained on the combined dataset of Qt and Openstack together, we found out that a single layer feed forward neural network, XGBoost, and SVM all perform well in AUC score by outperforming state-of-the-art work EALR on both datasets. However, when compared with state-of-the-art work DeepJIT, there is still a little gap of 0.05 to 0.1in AUC score on both datasets where better optimization of the neural network may have the potential of catching up. In general, using handcrafted features offer fast training and testing time when compared with using commit messages, and offer solid performance.

Models performance on Qt alone and OPENSTACK alone show similar AUC results with the merged dataset of Qt and OPENSTACK together, but single dataset shows a better f1 from the merged dataset result, which is reasonable, because the assumption is that Qt and OPENSTACK have similar characteristic, like the distribution of bugs.

There is still space to improve in future, we can still improve the performance of models, we can try LSTM following the time serious of code changes, and also reinforcement learning can be implemented in this area.

Reference:

Cross-Project Just-in-Time Bug Prediction for Mobile Apps: An Empirical Assessment

A Large-Scale Empirical Study of JIT quality assurance

Fine-grained just-in-time defect prediction

DeepJIT: an end-to-end deep learning framework for just-in-time defect prediction