Project

ICS485

Group #11

	Work distribution
Abdullah Alsalamah (201570790)	**33%**
Faris Hijazi (201578750)	**33%**
Rayan Ghadi (201532590)	**33%**

Table of Contents

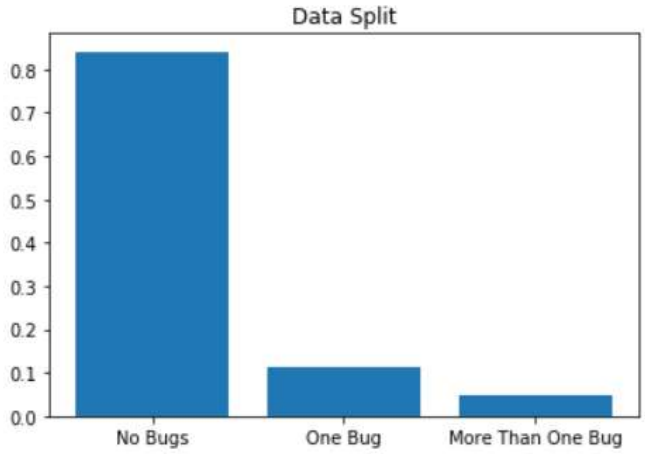
Introduction

In this project we will build three different models and a neural network to help predict if given code has bugs or not, and how many bugs.

We have been given 5 datasets from different IDEs which each contains the same features so we upload all the data in one DataFrame using pandas and dropped the fires column(feature) which is the name of the class because it will not affect the learning.

After combing all data from different IDEs now we have 5371 samples and each has 17 features without any missing data which is good because we don't need to deal with the missing values, but, we had the problem of imbalanced data where for binary classification 84.12% of

the samples have no bugs while only 15.88% have bugs we tried to solve this problem by using different technique for examples oversampling and undersampling.

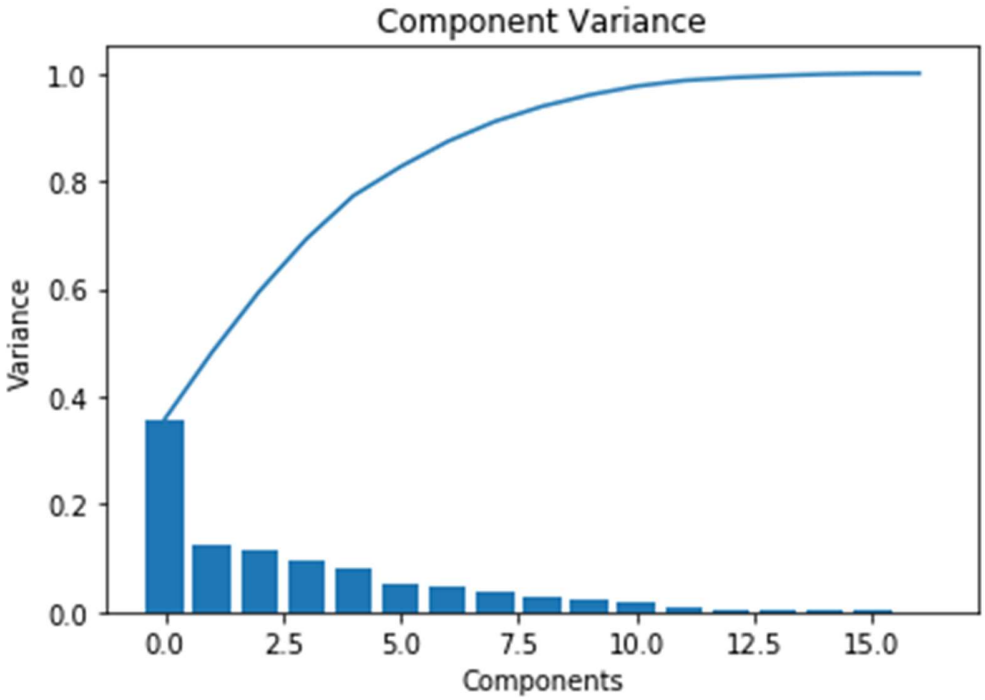



```
84.12% of the samples has no bugs (4518 samples out of 5371)
11.13% of the samples has one bug (598 samples out of 5371)
4.75% of the samples has more than one bug (255 samples out of 5371)
```

Finally, we used PCA which helps us to drop some features that has the least variances. As well we plotted a graph to show the correlation between the features and the labels so we can drop some features that has low correlations with the label.

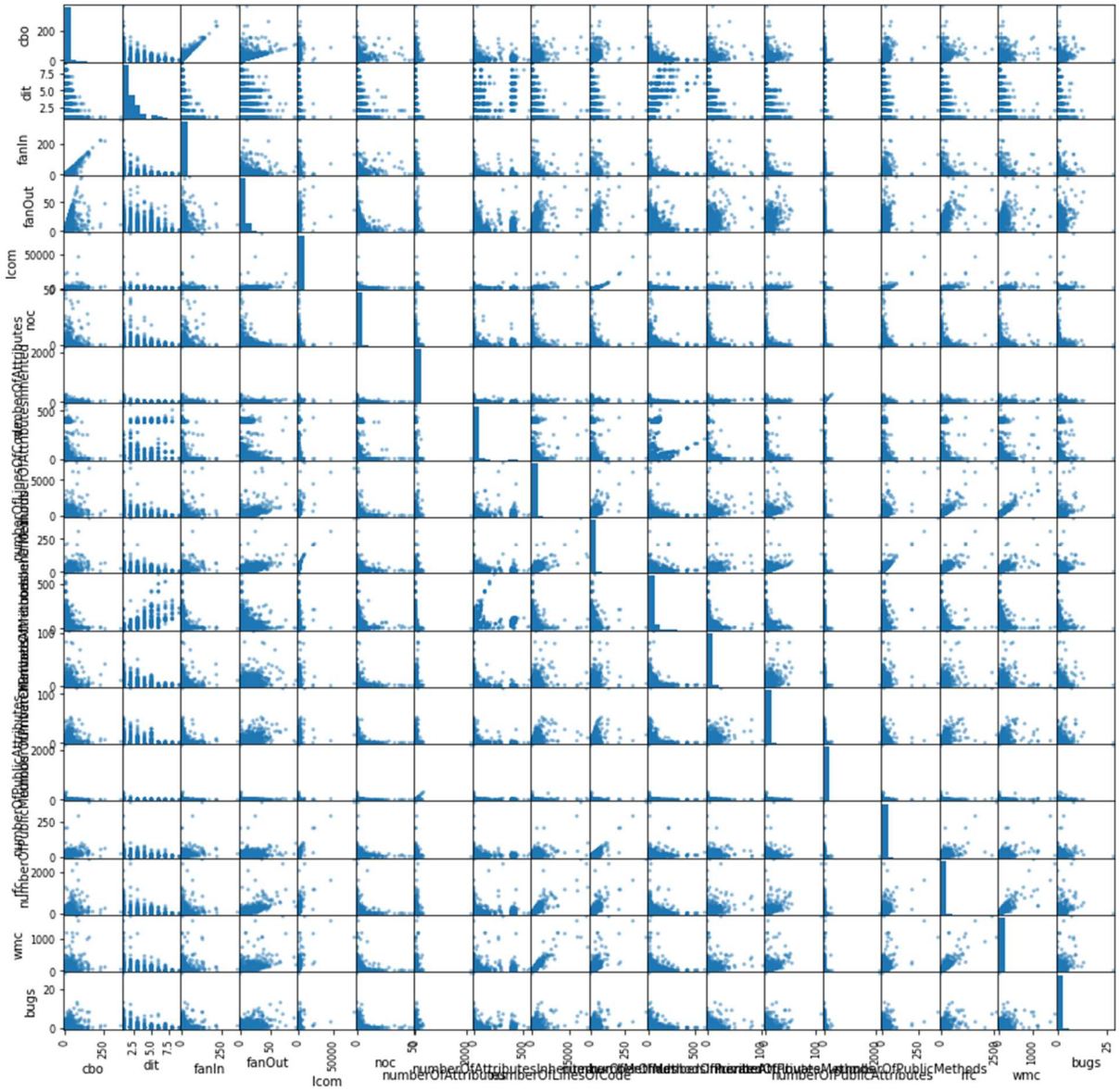
1. Preprocessing

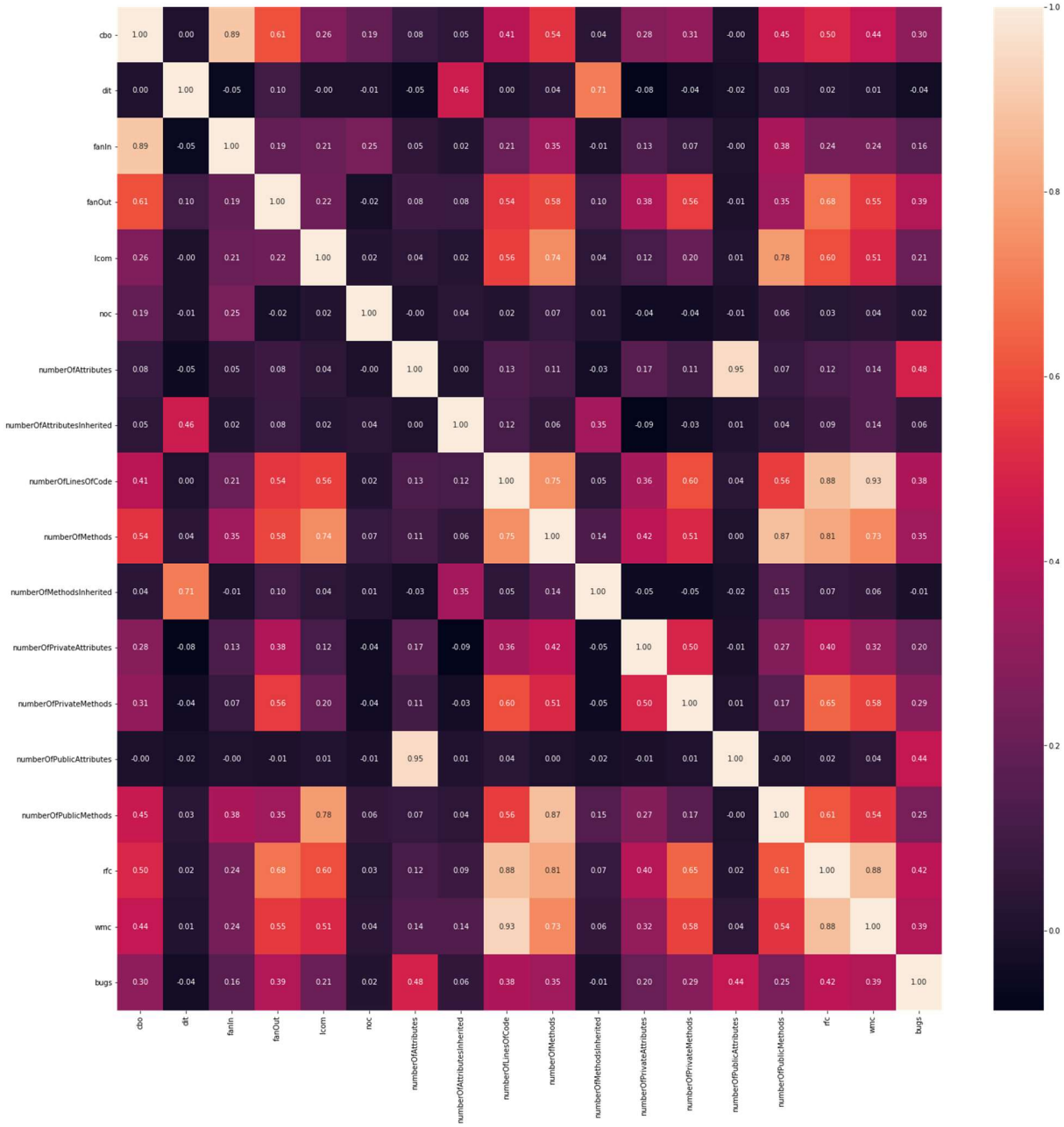
1.1. Principle Component Analysis (PCA)



As shown in the figure above most of the variance(around 90%) is contains in the first 10 components. So, based on that we transformed our data to these 10 components to do a prediction using only these components which will drop 7 of our features, we trained our bagging model using GridSearchCV with cross validation(cv) = 5 and got 86% accuracy.

1.2. Dropping Least Correlated Features




cbo 1.00 0.00 0.89 0.61 0.26 0.19 0.08 0.05 0.41 0.54 0.04 0.28 0.31 -0.00 0.45 0.50 0.44 0.30
dit 0.00 1.00 -0.05 0.10 -0.00 -0.01 -0.05 0.46 0.00 0.04 0.71 -0.08 -0.04 -0.02 0.03 0.02 0.01 -0.04
fanIn 0.89 -0.05 1.00 0.19 0.21 0.25 0.05 0.02 0.21 0.35 -0.01 0.13 0.07 -0.00 0.38 0.24 0.24 0.16
fanOut 0.61 0.10 0.19 1.00 0.22 -0.02 0.08 0.08 0.54 0.58 0.10 0.38 0.56 -0.01 0.35 0.68 0.55 0.39
lcom 0.26 -0.00 0.21 0.22 1.00 0.02 0.04 0.02 0.56 0.74 0.04 0.12 0.20 0.01 0.78 0.60 0.51 0.21
noc 0.19 -0.01 0.25 -0.02 0.02 1.00 -0.00 0.04 0.02 0.07 0.01 -0.04 -0.04 -0.01 0.06 0.03 0.04 0.02
numberOfAttributes 0.08 -0.05 0.05 0.08 0.04 -0.00 1.00 0.00 0.13 0.11 -0.03 0.17 0.11 0.95 0.07 0.12 0.14 0.48
numberOfAttributesInherited 0.05 0.46 0.02 0.08 0.02 0.04 0.00 1.00 0.12 0.06 0.35 -0.09 -0.03 0.01 0.04 0.09 0.14 0.06
numberOfLinesOfCode 0.41 0.00 0.21 0.54 0.56 0.02 0.13 0.12 1.00 0.75 0.05 0.36 0.60 0.04 0.56 0.88 0.93 0.38
numberOfMethods 0.54 0.04 0.35 0.58 0.74 0.07 0.11 0.06 0.75 1.00 0.14 0.42 0.51 0.00 0.87 0.81 0.73 0.35
numberOfMethodsInherited 0.04 0.71 -0.01 0.10 0.04 0.01 -0.03 0.35 0.05 0.14 1.00 -0.05 -0.05 -0.02 0.15 0.07 0.06 -0.01
numberOfPrivateAttributes 0.28 -0.08 0.13 0.38 0.12 -0.04 0.17 -0.09 0.36 0.42 -0.05 1.00 0.50 -0.01 0.27 0.40 0.32 0.20
numberOfPrivateMethods 0.31 -0.04 0.07 0.56 0.20 -0.04 0.11 -0.03 0.60 0.51 -0.05 0.50 1.00 0.01 0.17 0.65 0.58 0.29
numberOfPublicAttributes -0.00 -0.02 -0.00 -0.01 0.01 -0.01 0.95 0.01 0.04 0.00 -0.02 -0.01 0.01 1.00 -0.00 0.02 0.04 0.44
numberOfPublicMethods 0.45 0.03 0.38 0.35 0.78 0.06 0.07 0.04 0.56 0.87 0.15 0.27 0.17 -0.00 1.00 0.61 0.54 0.25
rfc 0.50 0.02 0.24 0.68 0.60 0.03 0.12 0.09 0.88 0.81 0.07 0.40 0.65 0.02 0.61 1.00 0.88 0.42
wmc 0.44 0.01 0.24 0.55 0.51 0.04 0.14 0.14 0.93 0.73 0.06 0.32 0.58 0.04 0.54 0.88 1.00 0.39
bugs 0.30 -0.04 0.16 0.39 0.21 0.02 0.48 0.06 0.38 0.35 -0.01 0.20 0.29 0.44 0.25 0.42 0.39 1.00
cbo
dit
fanIn
fanOut
lcom
noc
numberOfAttributes
numberOfAttributesInherited
numberOfLinesOfCode
numberOfMethods
numberOfMethodsInherited
numberOfPrivateAttributes
numberOfPrivateMethods
numberOfPublicAttributes
numberOfPublicMethods
rfc
wmc
bugs
1.0
0.8
0.6
0.4
0.2
0.0

From the above two figures we can see that there is almost no correlation between the label(“bugs”) and the features “dit”, “noc”,

“numberOfAttributesInherited” and “numberOfMethodsInherited” so we dropped them and then we tried our three models (Random Forest, AdaBoost, DecisionTree) again on the new data and it got almost the same results without these features.

2. Task 1: Binary Classifiers

2.1. Results Without Feature Engineering

First of all we upload the data and used standard scaler to scale the data because after looking at the data distribution we noticed that there is a huge different between the values of the features for example numberOfLinesOfCode can go from 0 to 7509 while dit can go from 0 to 93 so if we leave the data without scaling numberOfLinesOfCode will effect our model the most which will affect the results of our classification.

After we scaled the data we tried different models (using GreadSearchCV with cross validation = 5) to look at the performance of the models without any features engineering and we got the following results

2.1.1. Decision Tree

```
TreeDecision_grid.best_score_
```

```
0.8488959829741952
```

```
predTree = TreeDecision_grid.predict(testx)
np.sum((predTree == testy) / len(testy))
```

```
0.8449131513647641
```

Decision Tree got 84.89% on the training set and 84.49% on the testing set which is a good result and it didn't overfit.

2.1.2. Bagging

```
bagging_grid.best_score_
```

```
0.8560787443469008
```

```
predBagging = bagging_grid.predict(testx)
np.sum((predBagging == testy) / len(testy))
```

```
0.8542183622828782
```

Bagging got 85.61% on the training set and 85.42% on the testing set which is a good result and it didn't overfit.

2.1.3. AdaBoost

```
AdaBoost_grid.best_score_
```

```
0.8443735035913806
```

```
predBoost = AdaBoost_grid.predict(testx)
np.sum((predBoost == testy) / len(testy))
```

```
0.8442928039702231
```

AdaBoost got 84.44% on the training set and 84.43% on the testing set which is a good result and it didn't overfit.

2.1.4. Random Forest

```
rf_grid.best_score_
```

```
0.8574088853418462
```

```
predRF = rf_grid.predict(testx)
np.sum((predRF == testy) / len(testy))
```

```
0.856699751861042
```

Random Forest got 85.74% on the training set and 85.67% on the testing set which is a good result and it didn't overfit.

2.1.5. Gaussian

```
ypred = clf.predict(Gtestx)
(sum(ypred == Gtesty) / len(ypred))
```

```
0.8451228592702904
```

Finally we tried Gaussaion on the data that are not scaled and we got 84.51% accuracy on the testing set

2.2. The Results with Feature Engineering

In this section we tried to find a solution for the imbalanced data by over sampled the minority class and then tried under sampled the majority class and test the results on three different classifiers RandomForest, AdaBoost and DecisionTree by using 70% of the data as training set, 15% as validation set and 15% as a test set. Finally, we tested our results using accuracy score and F beta score with different values of betas in this kind of problem we want to give more weights to the Recall so we are focusing in the scores with high value of beta.

For binary classifiers in the normal training set we had 3163 for class -1 (no bugs) and 596 for class 1 (there is a bug) which is obviously imbalanced so we tried oversampling by increasing the number of samples in class 1 so the number of samples in class 1 will be equal to the number of samples in class -1, finally, we tried undersampling by decreasing the number of samples of class 1 to be equal to the number of samples in class -1.

2.2.1. RandomForest

2.2.1.1. Normal Data set

```
_____________________________________________
Accuracy of the train set = 92.924%
Accuracy of the validation set = 85.732%
Accuracy of the test set = 86.973%

_____________________________________________
Results for F_Beta_Score:
beta = 0.001000, score = 77.05%
beta = 0.010000, score = 77.04%
beta = 0.500000, score = 69.56%
beta = 1.000000, score = 64.17%
beta = 5.000000, score = 61.19%
beta = 10.000000, score = 61.12%
```

2.2.1.2. Oversampled Dataset

```
_____________________________________________
Accuracy of the train set = 99.810%
Accuracy of the validation set = 85.236%
Accuracy of the test set = 85.856%

_____________________________________________
Results for F_Beta_Score:
beta = 0.001000, score = 71.26%
beta = 0.010000, score = 71.26%
beta = 0.500000, score = 69.01%
beta = 1.000000, score = 66.60%
beta = 5.000000, score = 64.39%
beta = 10.000000, score = 64.29%
```

2.2.1.3. Undersampled Dataset

```
Accuracy of the train set = 86.493%
Accuracy of the validation set = 74.566%
Accuracy of the test set = 73.945%

Results for F_Beta_Score:
beta = 0.001000, score = 58.57%
beta = 0.010000, score = 58.57%
beta = 0.500000, score = 58.60%
beta = 1.000000, score = 59.25%
beta = 5.000000, score = 62.39%
beta = 10.000000, score = 62.72%
```

2.2.2. AdaBoost

2.2.2.1. Normal Dataset

```
Accuracy of the train set = 99.681%
Accuracy of the validation set = 84.491%
Accuracy of the test set = 85.608%

Results for F_Beta_Score:
beta = 0.001000, score = 70.46%
beta = 0.010000, score = 70.45%
beta = 0.500000, score = 67.76%
beta = 1.000000, score = 65.08%
beta = 5.000000, score = 62.84%
beta = 10.000000, score = 62.75%
```

2.2.2.2. Oversampled Dataset

```
Accuracy of the train set = 99.810%
Accuracy of the validation set = 84.119%
Accuracy of the test set = 85.360%

Results for F_Beta_Score:
beta = 0.001000, score = 69.54%
beta = 0.010000, score = 69.54%
beta = 0.500000, score = 66.38%
beta = 1.000000, score = 63.46%
beta = 5.000000, score = 61.29%
beta = 10.000000, score = 61.21%
```

2.2.2.3. Undersampled Dataset

```
Accuracy of the train set = 99.832%
Accuracy of the validation set = 72.457%
Accuracy of the test set = 73.449%

Results for F_Beta_Score:
beta = 0.001000, score = 59.96%
beta = 0.010000, score = 59.96%
beta = 0.500000, score = 59.86%
beta = 1.000000, score = 60.62%
beta = 5.000000, score = 65.31%
beta = 10.000000, score = 65.83%
```

2.2.3. DecisionTree

2.2.3.1. Normal Dataset

```
Accuracy of the train set = 87.257%
Accuracy of the validation set = 83.871%
Accuracy of the test set = 84.615%

Results for F_Beta_Score:
beta = 0.001000, score = 66.84%
beta = 0.010000, score = 66.84%
beta = 0.500000, score = 63.87%
beta = 1.000000, score = 61.23%
beta = 5.000000, score = 59.43%
beta = 10.000000, score = 59.38%
```

2.2.3.2. Oversampled Dataset

```
Accuracy of the train set = 99.810%
Accuracy of the validation set = 79.777%
Accuracy of the test set = 80.273%

Results for F_Beta_Score:
beta = 0.001000, score = 60.30%
beta = 0.010000, score = 60.30%
beta = 0.500000, score = 60.23%
beta = 1.000000, score = 60.11%
beta = 5.000000, score = 59.95%
beta = 10.000000, score = 59.95%
```

2.2.3.3. Undersampled Dataset

```
_________________________________________
Accuracy of the train set = 73.238%
Accuracy of the validation set = 73.077%
Accuracy of the test set = 72.953%

_________________________________________
Results for F_Beta_Score:
beta = 0.001000, score = 60.48%
beta = 0.010000, score = 60.48%
beta = 0.500000, score = 60.25%
beta = 1.000000, score = 61.02%
beta = 5.000000, score = 66.59%
beta = 10.000000, score = 67.23%
```

Most of the models are overfitted on the training set which most of them get around 99% accuracy score, but as well they got good results in the testing sets where most of the models succeeded to reach around 85% accuracy but they did poorly on the F beta score for small values of beta and large values of beta.

The best model which get the best accuracy results in the test set is Random Forest which got 86.97% accuracy. But our goal is the get the best accuracy on the F beta score with high value of beta which will give higher weight for the recall so we can increase our confidence which we will not miss a program that has a bug. Decision tree on under sampled dataset got the best F beta score with beta = 10 with score = 67.23% even it got the worse accuracy on the training and test set but it got the best F beta score on beta = 10.

3. Task 2: Multi-Classifiers

3.1. Results Without Feature Engineering

1.1.1. Decision Tree

```
TreeDecision_grid.best_score_
```

```
0.8427773343974462
```

```
predTreeT = TreeDecision_grid.predict(testxT)
np.sum((predTreeT == testyT) / len(testyT))
```

```
0.8399503722084365
```

The results we got for Multi-Classifiers using Decision Tree are 84.28% on the training set and 84.00% on the testing set

1.1.2. AdaBoost

```
AdaBoost_grid.best_score_
```

```
0.8496940675711625
```

```
predBoostT = AdaBoost_grid.predict(testxT)
np.sum((predBoostT == testyT) / len(testyT))
```

```
0.8399503722084365
```

The results we got for Multi-Classifiers using AdaBoost are 84.97% on the training set and 84.00% on the testing set

1.1.3. Random Forest

```
rf_grid.best_score_
```

```
0.8502261239691408
```

```
predRFT = rf_grid.predict(testxT)
np.sum((predRFT == testyT) / len(testyT))
```

```
0.8449131513647641
```

The results we got for Multi-Classifiers using Random Forest are 85.02% on the training set and 84.49% on the testing set

From the results above all the classifiers have similar results but Random Forest Classifier got the best on the training and testing set for both Binary and multi classifiers.

3.2. The Results with Feature Engineering

For multi classifiers in the normal training set we had 3163 for class 0 (no bugs), 421 for class 1 (there is a bug), and 175 for class 2 (there is more than one bug) and as we mentioned before the data are highly imbalanced so we will try our classifiers on the normal dataset then we will do oversampling on the training set by making all the class have 3163 samples after that we will try to do undersampling on the training set by making all the class have 175 samples

3.2.1. RandomForest

3.2.2. Normal Data set

```
Accuracy of the train set = 96.276%
Accuracy of the validation set = 84.119%
Accuracy of the test set = 85.980%

Results for F_Beta_Score:
beta = 0.001000, score = 62.89%
beta = 0.010000, score = 62.87%
beta = 0.500000, score = 47.64%
beta = 1.000000, score = 41.89%
beta = 5.000000, score = 39.89%
beta = 10.000000, score = 39.87%
```

3.2.3. Oversampled Dataset

```
Accuracy of the train set = 99.874%
Accuracy of the validation set = 83.747%
Accuracy of the test set = 84.988%

Results for F_Beta_Score:
beta = 0.001000, score = 57.54%
beta = 0.010000, score = 57.53%
beta = 0.500000, score = 52.82%
beta = 1.000000, score = 48.63%
beta = 5.000000, score = 45.44%
beta = 10.000000, score = 45.32%
```

3.2.4. Undersampled Dataset

```
Accuracy of the train set = 58.857%
Accuracy of the validation set = 68.486%
Accuracy of the test set = 68.486%

Results for F_Beta_Score:
beta = 0.001000, score = 43.43%
beta = 0.010000, score = 43.43%
beta = 0.500000, score = 43.39%
beta = 1.000000, score = 44.69%
beta = 5.000000, score = 53.67%
beta = 10.000000, score = 54.88%
```

3.2.5. AdaBoost

3.2.5.1. Normal Dataset

```
Accuracy of the train set = 99.681%
Accuracy of the validation set = 83.995%
Accuracy of the test set = 85.484%

Results for F_Beta_Score:
beta = 0.001000, score = 54.85%
beta = 0.010000, score = 54.84%
beta = 0.500000, score = 44.99%
beta = 1.000000, score = 41.09%
beta = 5.000000, score = 39.75%
beta = 10.000000, score = 39.74%
```

3.2.5.2. Oversampled Dataset

```
Accuracy of the train set = 99.874%
Accuracy of the validation set = 83.747%
Accuracy of the test set = 85.484%

Results for F_Beta_Score:
beta = 0.001000, score = 56.66%
beta = 0.010000, score = 56.65%
beta = 0.500000, score = 47.40%
beta = 1.000000, score = 42.59%
beta = 5.000000, score = 40.52%
beta = 10.000000, score = 40.49%
```

3.2.5.3. Undersampled Dataset

```
Accuracy of the train set = 99.810%
Accuracy of the validation set = 63.524%
Accuracy of the test set = 61.663%

Results for F_Beta_Score:
beta = 0.001000, score = 40.46%
beta = 0.010000, score = 40.46%
beta = 0.500000, score = 39.47%
beta = 1.000000, score = 39.67%
beta = 5.000000, score = 48.68%
beta = 10.000000, score = 50.18%
```

3.2.6. DecisionTree

3.2.6.1. Normal Dataset

```
_____________________________________________
Accuracy of the train set = 86.034%
Accuracy of the validation set = 82.258%
Accuracy of the test set = 83.499%

_____________________________________________
Results for F_Beta_Score:
beta = 0.001000, score = 42.81%
beta = 0.010000, score = 42.81%
beta = 0.500000, score = 40.15%
beta = 1.000000, score = 38.32%
beta = 5.000000, score = 37.58%
beta = 10.000000, score = 37.58%
```

3.2.6.2. Oversampled Dataset

```
_____________________________________________
Accuracy of the train set = 99.831%
Accuracy of the validation set = 75.931%
Accuracy of the test set = 75.062%

_____________________________________________
Results for F_Beta_Score:
beta = 0.001000, score = 38.38%
beta = 0.010000, score = 38.38%
beta = 0.500000, score = 38.37%
beta = 1.000000, score = 38.40%
beta = 5.000000, score = 38.52%
beta = 10.000000, score = 38.54%
```

3.2.6.3. Undersampled Dataset

```
Accuracy of the train set = 64.952%
Accuracy of the validation set = 57.692%
Accuracy of the test set = 61.538%

Results for F_Beta_Score:
beta = 0.001000, score = 39.82%
beta = 0.010000, score = 39.82%
beta = 0.500000, score = 38.96%
beta = 1.000000, score = 39.19%
beta = 5.000000, score = 47.31%
beta = 10.000000, score = 48.60%
```

Most of the models suffers from the overfitting problems where almost all of them reach 99% accuracy on the training set and got around 85% accuracy on the testing set. The best model with the highest accuracy in the test set is Random Forest with 85.98% but there is no huge different between it and the other models. But our focus is on the F beta score with beta = 10 where all the models performed poorly. The best model with F beta score is AdaBoost with score = 54.88% which almost got the result of the random classifier.

4. Task 3: Neural Network classifier

This section discusses the neural network implementation on the given problem, as well as the methods for choosing hyperparameters and evaluation metrics.

We trained a 3-layer deep neural network with dropouts to decrease overfitting.

For details, please refer to the section's notebook, it was made brief and easy to read, as it mainly has the outputs, (little code was shown in the notebook).

4.1. Methods

Since there are multiple methods that will be compared and measured, the code was structured to have all the different methods defined at the same time and stored in a dictionary to be compared later.

```
model_suffixes = [
    '',

    ## binary classifiers
    'bc', # normal, no undersampling
    'weighted_bc',
    'usamp_bc',
    'usamp_reduced_bc',

    # lda
    'lda_bc', # normal, no undersampling
    'lda_weighted_bc',
    'lda_usamp_bc',

    ## multi-class classification
    'mc', # normal, no undersampling
    'weighted_mc',
    'usamp_mc',
    'usamp_reduced_mc',

    'lda_mc',
    'lda_weighted_mc',
    'lda_usamp_mc',
]
```

Figure 1 dictionary of the compbination of approaches

This helps in testing all the following combinations:

- Unweighted/Weighted data
- Raw/Under sampled data
- Binary labels/multiclass labels
- Raw/LDA transformed

4.2. Preprocessing

The data was imported, and *NaN* values were dropped. Usually, categorical values would be one-hot-encoded, but in this case the values were just too many and were dropped as well.

Linear Discriminant Analysis (LDA) was also performed on the data, some of the data was transformed, and some of the data also had the dimensions reduced to the most important principle components.

4.3. Observations

4.3.1. Chaotic validation loss for reduced datasets

It was noticed that the models with the reduced data tend to be hectic, so they need lower learning rate to about half.

```
Epoch:(450/500) 00.13s (90.1%)  Loss: 0.593084  VLoss: 0.566     accuracy: 67.06%
Epoch:(500/500) 00.12s (99.8%)  Loss: 0.592195  VLoss: 0.620     accuracy: 66.27%
Completed 01m:00010s in with a minimum validation loss of: 0.5621113078668714
```

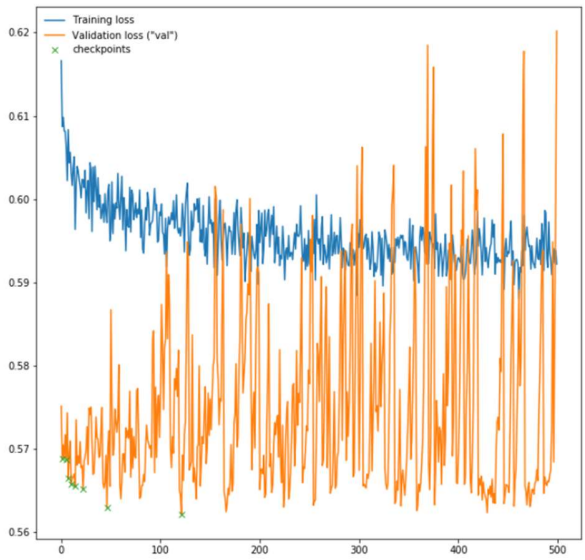


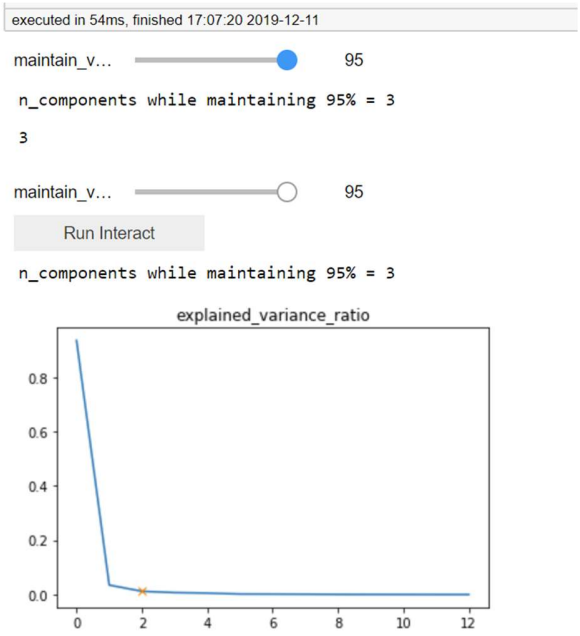
Figure 2 Chaotic behavior for models running on the reduced dataset

```
lr = 0.01
if 'reduced' in keywords:
    lr *= 0.5
```

Figure 3 lowering learning rate for "reduced" datasets

4.3.2. LDA also performs standard scaling

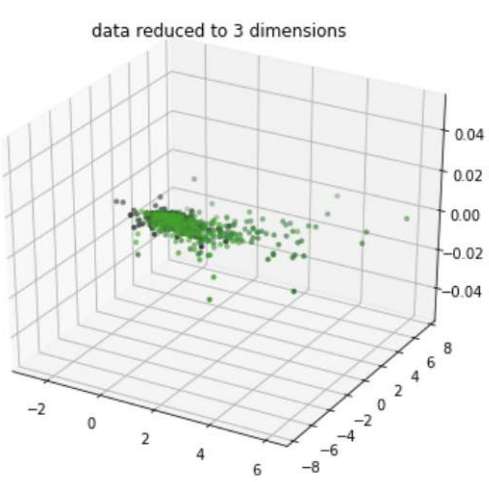
It was also noticed that after performing LDA transformation, scaling the data makes no difference, meaning that LDA does some sort of scaling or normalizing the data.

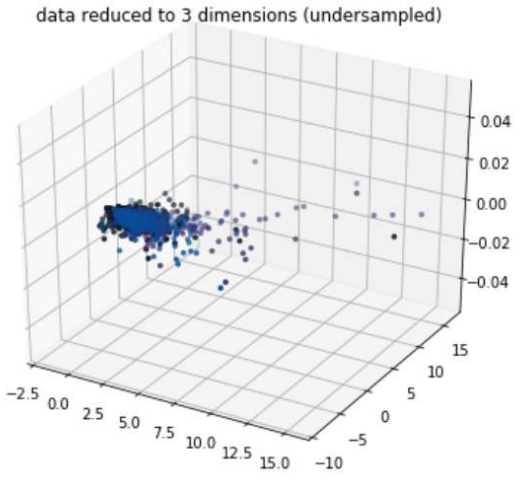


7]: <function __main__.plot_varience(maintain_varience=95)>

Interactive widget to find minimum components needed for a given maintained variance (after doing LDA).

And as can be seen, only 3 dimensions are needed to preserve 95% of the variance. And the data was reduced to 3 dimensions accordingly to this criterion. The data can be seen in the below graphs:

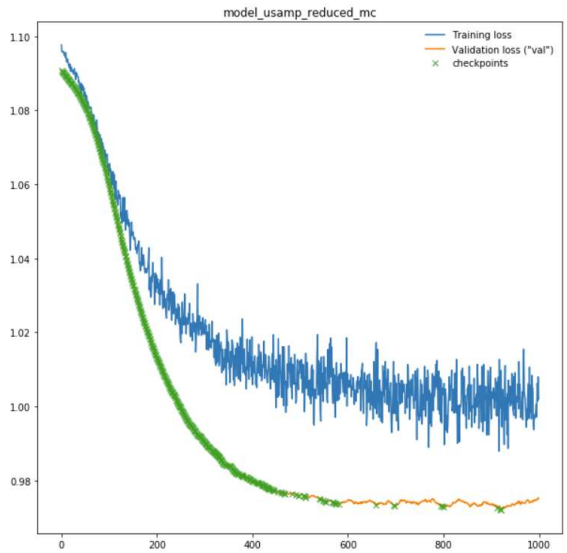




We can see that the problem itself is extremely difficult, even with the top 3 components, it's still hard to see where the decision boundry should be, the data itself is not very indicative, so it wouldn't be a surprise if the models didn't perform great.

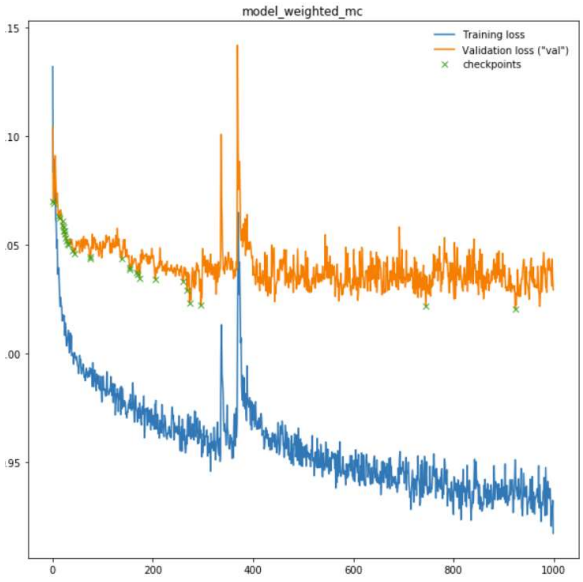
4.3.3. Validation loss lower than training loss

This phenomenon seems to only happen for the "undersampled_reduced_multiclass" model. It could be because of a bad random seed of choosing the validation samples, but this keeps reoccurring across different runs.



4.3.4. Strange loss spike

This spike was happening when I was using the *AdaM* optimizer (Adaptive Momentum), so I thought it could be the problem (maybe the momentum suddenly shifted), so I switched over to *SGD* optimizer (stochastic gradient descent). And most models stopped this behavior, but some still continued.



4.4. Implementation details

The neural network was built using PyTorch, which supports GPU (however GPU actually runs slower on this data because it's relatively small).

The architecture having 1 fully connected layer of size 64, followed by another FC layer of size 32, followed by the output layer.

Now for binary classification, the neural network doesn't need to have 2 output nodes, it is possible to implement using only 1 output node and conclude the other value by subtracting 1 from the sigmoid output. The advantage to this is not only computing power, but also the neural network will have less weights to train, meaning the training samples can be utilized to train the other weights, resulting in a less noisy model.

The bellow code shows the definition of the neural network and the implementation of this optimization.

In [20]:

```
class ModelMC(nn.Module):
    def __init__(self, input_size, n_classes, drop_prob=0.2):
        super(ModelMC, self).__init__()
        self.input_size = input_size
        self.n_classes = n_classes

        self.dropout = nn.Dropout(drop_prob)

        self.fc1 = nn.Linear(input_size, 64)
        self.fc2 = nn.Linear(64, 32)
        self.fc3 = nn.Linear(32, n_classes)

    def forward(self, x):
        batch_size = x.size(0)

        # fully connected layer 1
        x = self.fc1(x)
        x = F.leaky_relu(x)
        x = self.dropout(x)

        # fully connected layer 2
        x = self.fc2(x)
        x = F.leaky_relu(x)
        x = self.dropout(x)

        # fully connected layer 3
        x = self.fc3(x)

        if self.n_classes == 1: # special case optimization
            # adding another column that contains the inverted probablities
            # this acts like the neural network has 2 output neurons,
            # but it doesn't have as many weights to train (optimization :D)
            x = add_inverted_binary_dimension(x)
            x = F.log_softmax(x, dim=1)
        else:
            x = F.log_softmax(x, dim=1)

        return x

def add_inverted_binary_dimension(x):
    return torch.cat((x, 1.0-x), dim=1)
```

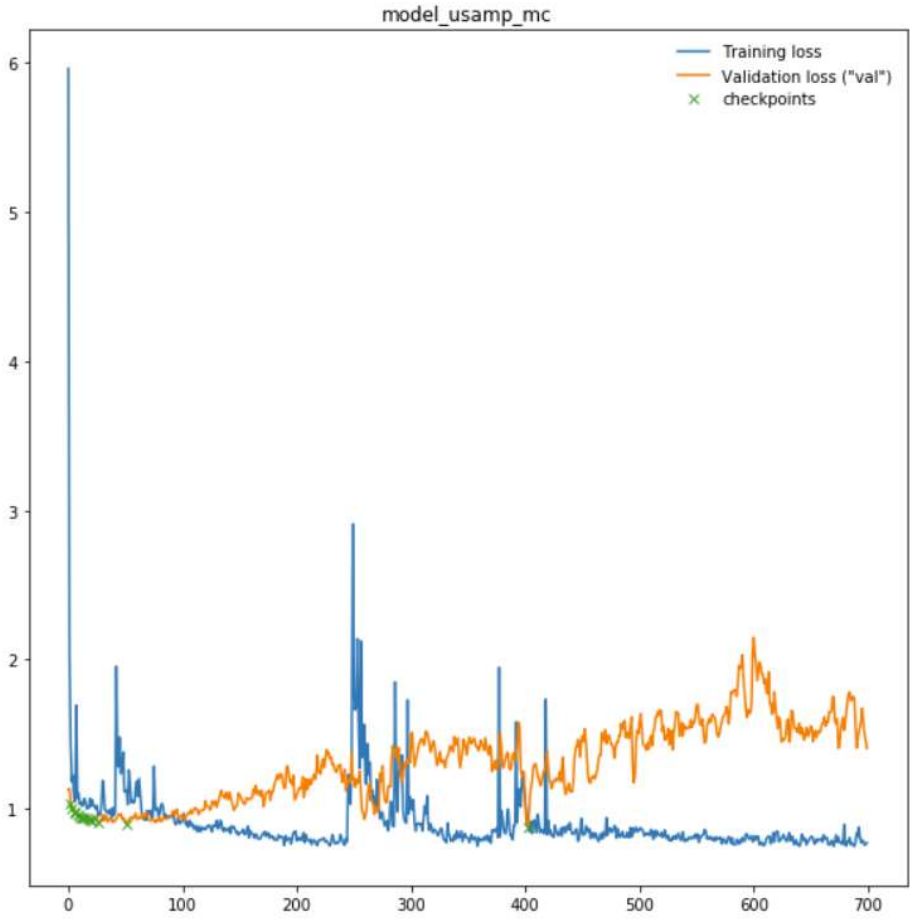
4.5. Training

Training is straight forward, but the performance must be evaluated to choose the best hyperparameters.

For each model, the training plots the training loss and the validation loss vs epochs.

```
Training model: usamp_mc

Estimated time needed:  26.57s. Come back at around:    15/12/2019 17:15
Epoch:(50/700) 00.03s (8.0%)    Loss: 1.113971  VLoss: 0.930     accuracy: 48.25%
Epoch:(100/700) 00.03s (16.0%)  Loss: 0.915289  VLoss: 0.953     accuracy: 46.49%
Epoch:(150/700) 00.04s (20.9%)  Loss: 0.823620  VLoss: 1.105     accuracy: 45.61%
Epoch:(200/700) 00.02s (35.4%)  Loss: 0.850023  VLoss: 1.066     accuracy: 43.86%
Epoch:(250/700) 00.04s (39.9%)  Loss: 2.909139  VLoss: 1.200     accuracy: 37.72%
Epoch:(300/700) 00.03s (47.8%)  Loss: 1.079074  VLoss: 1.309     accuracy: 42.11%
Epoch:(350/700) 00.04s (52.3%)  Loss: 0.807953  VLoss: 1.362     accuracy: 46.49%
Epoch:(400/700) 00.03s (62.2%)  Loss: 1.120740  VLoss: 1.243     accuracy: 41.23%
Epoch:(450/700) 00.03s (66.6%)  Loss: 0.935298  VLoss: 1.450     accuracy: 42.11%
Epoch:(500/700) 00.03s (74.5%)  Loss: 0.849203  VLoss: 1.504     accuracy: 45.61%
Epoch:(550/700) 00.03s (82.6%)  Loss: 0.829920  VLoss: 1.640     accuracy: 43.86%
Epoch:(600/700) 00.03s (88.4%)  Loss: 0.778860  VLoss: 2.045     accuracy: 47.37%
Epoch:(650/700) 00.03s (94.2%)  Loss: 0.799951  VLoss: 1.520     accuracy: 44.74%
Epoch:(700/700) 00.03s (99.9%)  Loss: 0.771891  VLoss: 1.405     accuracy: 43.86%
Completed 26.93s in with a minimum validation loss of: 0.8785055577754974
```



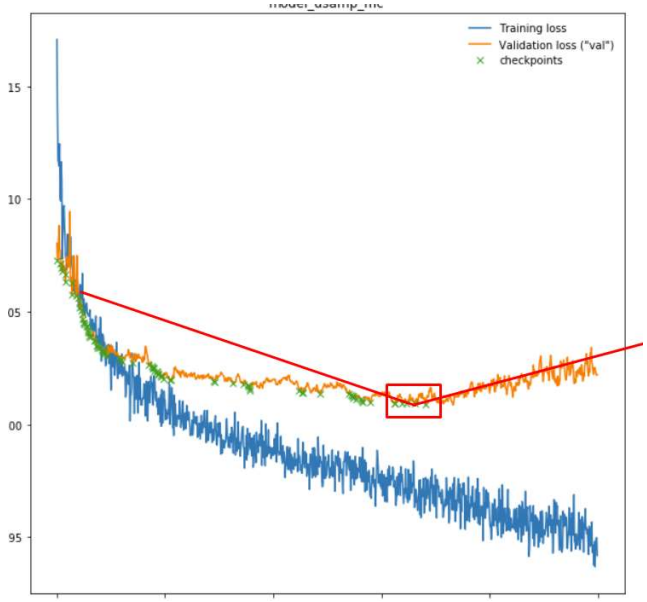
The architecture was chosen based on the fact that first layer should be large (to learn some features), and then gets smaller as the features

become more complex, and the output layer then chooses the class based on the high-level features.

Now for preventing over fitting, a dropout layer with probability 20% was added after every fully connected layer to prevent overfitting, as well as a ReLu for the nonlinearity.

4.5.1. Choosing when to stop/ epochs

This is a tricky question, but one method is to use the validation loss and take the model that has the least validation loss. The intuition behind this is that we're choosing the model right before it overfits. So each model trained for many epochs (700), and the one with the least validation loss was taken.



The model with low validation loss is saved to disk and then loaded later for testing.

4.6. Results

There are 14 models in total, the scores and results can be found in the appendix. But for this section, only some results will be discussed.

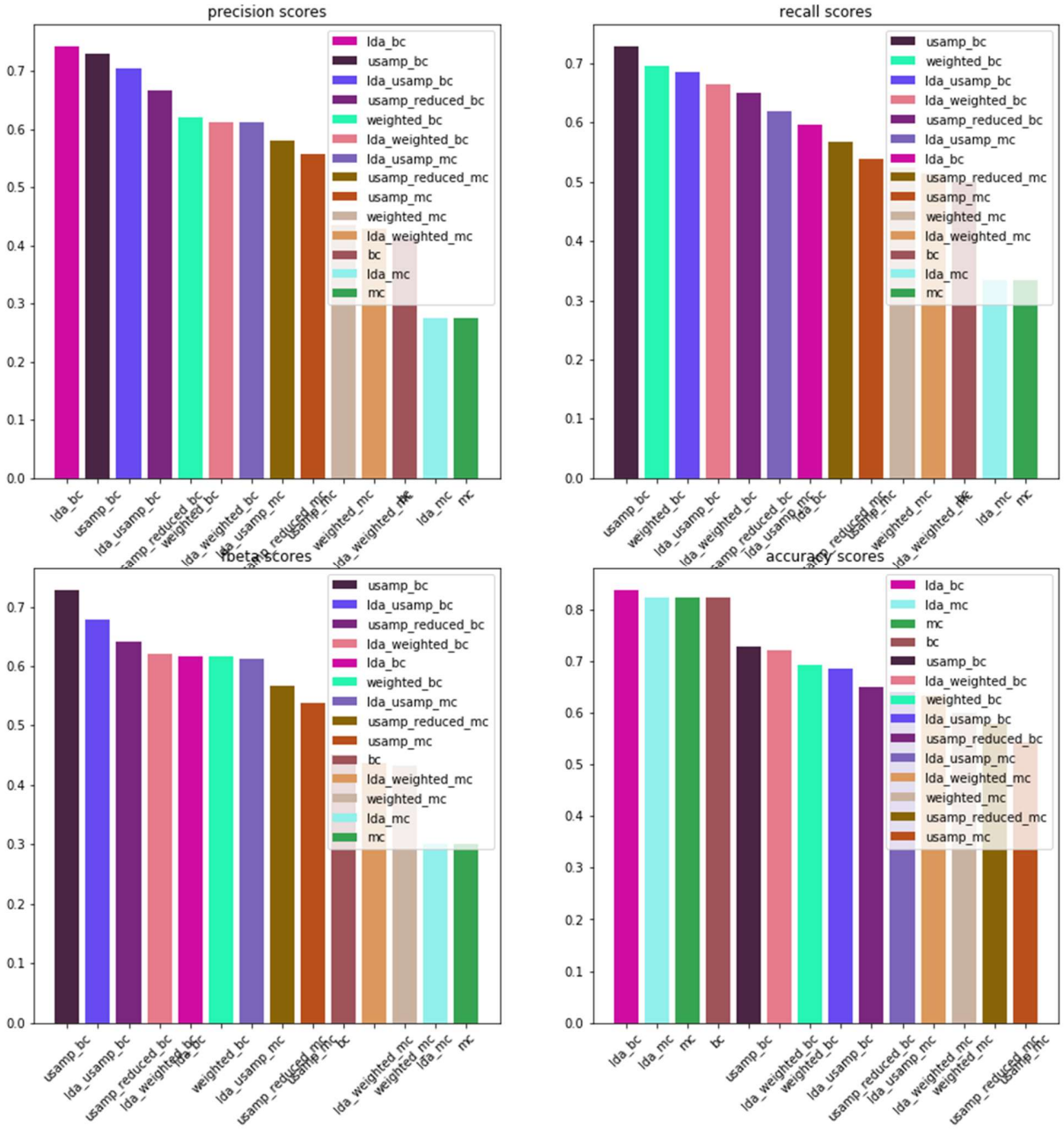
First, we need a baseline to compare these models to, for that we're going to be using a majority classifier for the unbalanced dataset, and a random classifier for the balanced dataset. The majority classifier achieves an accuracy of 82% (663/805).

Now accuracy score alone isn't enough since the dataset is extremely skewed/unbalanced.

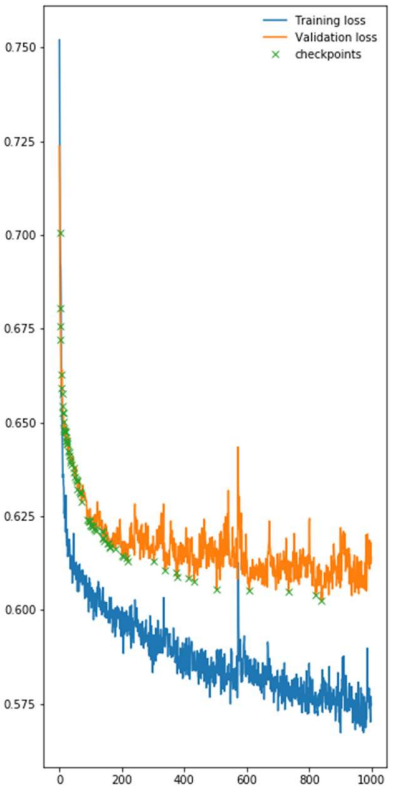
The highest accuracy achieved is 84.22% for the binary class problems and 75.22 by the LDA undersampled Multiclass-classifier for the multi-class problem.

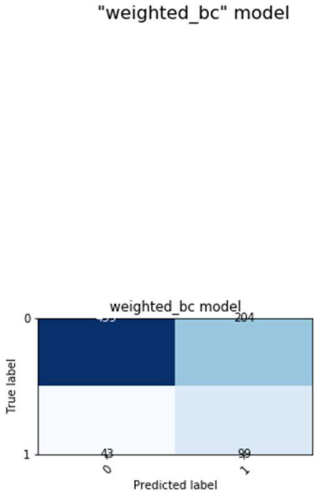
The bellow graphs are the recall/precision/fbeta/accuracy scores averaged across the classes.

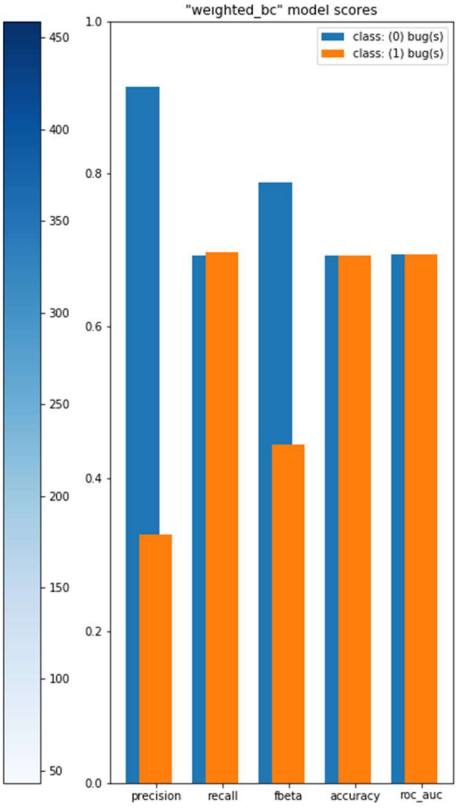
It is clear that under sampling and weighing the data does improve the final score. Also, the binary classifiers perform much better than the multiclass classifiers (this is expected).

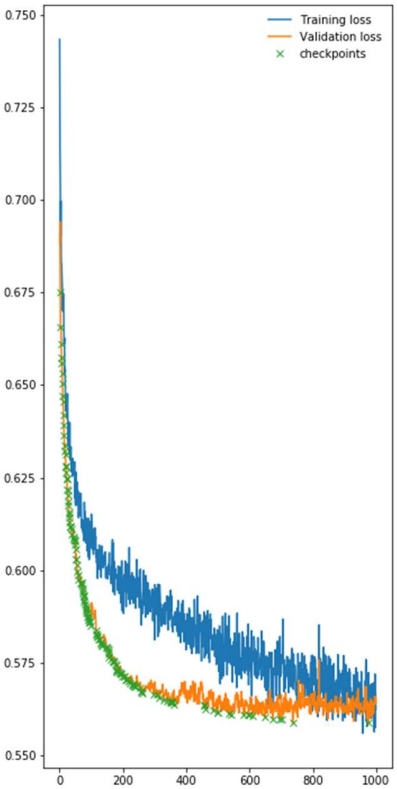

precision scores
lda_bc
usamp_bc
lda_usamp_bc
usamp_reduced_bc
weighted_bc
lda_weighted_bc
lda_usamp_mc
usamp_reduced_mc
usamp_mc
weighted_mc
lda_weighted_mc
bc
lda_mc
mc
recall scores
usamp_bc
weighted_bc
lda_usamp_bc
lda_weighted_bc
usamp_reduced_bc
lda_usamp_mc
lda_bc
usamp_reduced_mc
usamp_mc
weighted_mc
lda_weighted_mc
bc
lda_mc
mc
fbeta scores
usamp_bc
lda_usamp_bc
usamp_reduced_bc
lda_weighted_bc
lda_bc
weighted_bc
lda_usamp_mc
usamp_reduced_mc
usamp_mc
bc
lda_weighted_mc
weighted_mc
lda_mc
mc
accuracy scores
lda_bc
lda_mc
mc
bc
usamp_bc
lda_weighted_bc
weighted_bc
lda_usamp_bc
usamp_reduced_bc
lda_usamp_mc
lda_weighted_mc
weighted_mc
usamp_reduced_mc
usamp_mc

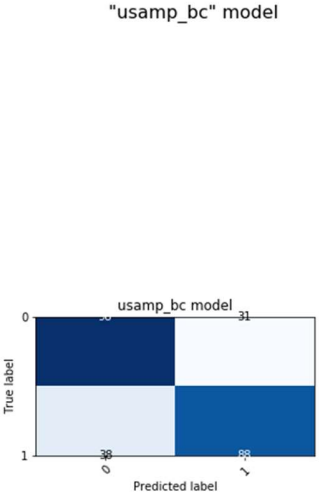
Appendix

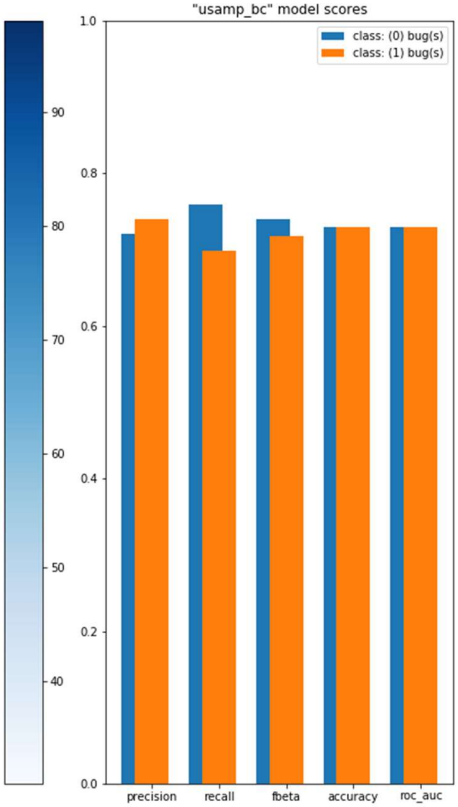

Training loss
Validation loss
checkpoints
0.750
0.725
0.700
0.675
0.650
0.625
0.600
0.575
0
200
400
600
800
1000

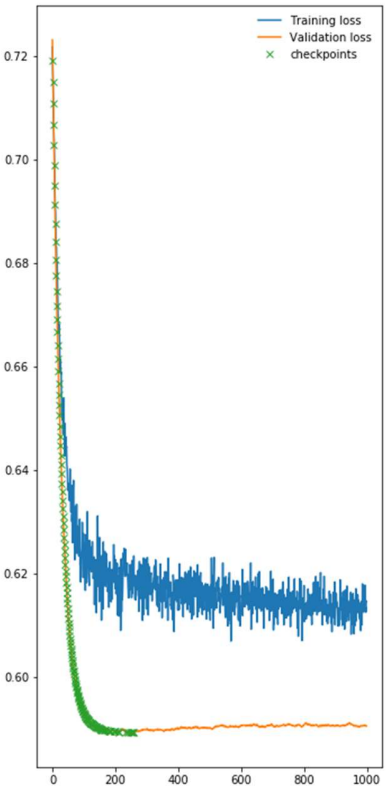

"weighted_bc" model
weighted_bc model
True label
Predicted label
0
1
204
43
99

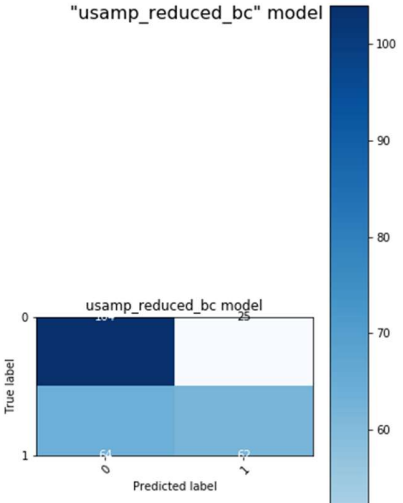

"weighted_bc" model scores
class: (0) bug(s)
class: (1) bug(s)
precision
recall
fbeta
accuracy
roc_auc

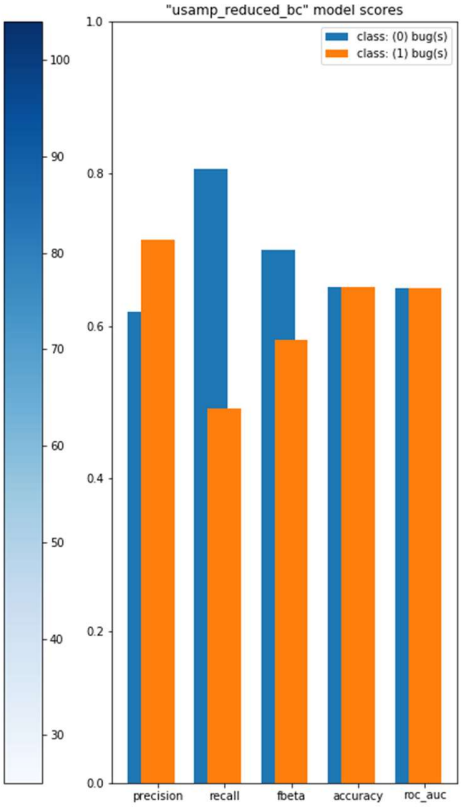

Training loss
Validation loss
checkpoints
0.750
0.725
0.700
0.675
0.650
0.625
0.600
0.575
0.550
0
200
400
600
800
1000

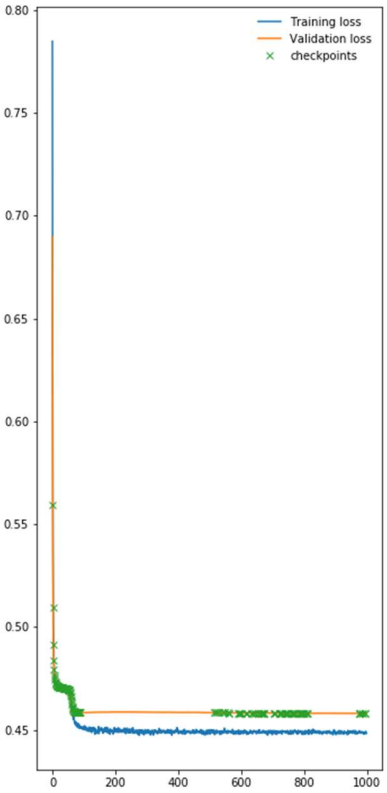

"usamp_bc" model
usamp_bc model
True label
Predicted label
0
1
31
38
88

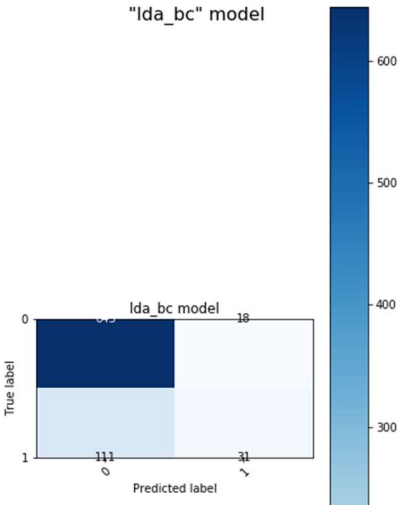

"usamp_bc" model scores
class: (0) bug(s)
class: (1) bug(s)
precision
recall
fbeta
accuracy
roc_auc

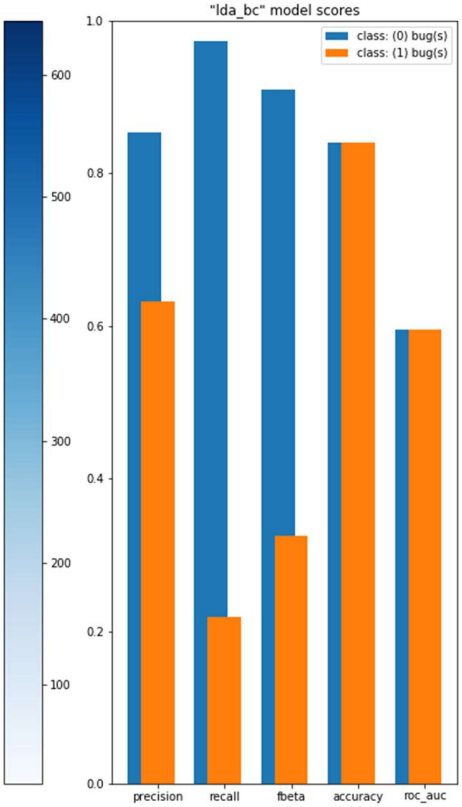

Training loss
Validation loss
checkpoints
0.72
0.70
0.68
0.66
0.64
0.62
0.60
0
200
400
600
800
1000

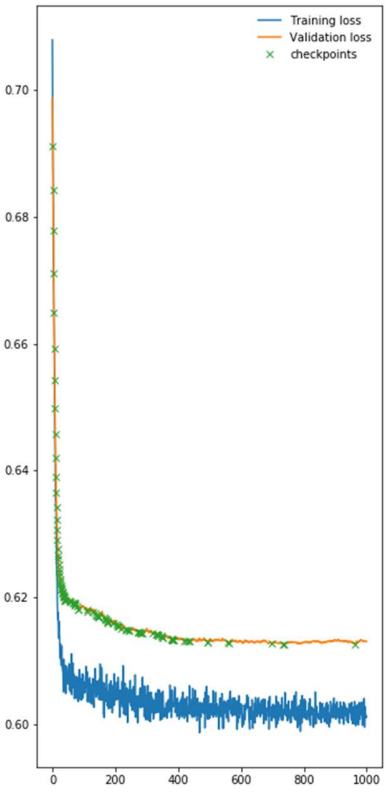

"usamp_reduced_bc" model
usamp_reduced_bc model
True label
Predicted label
0
1
25
57

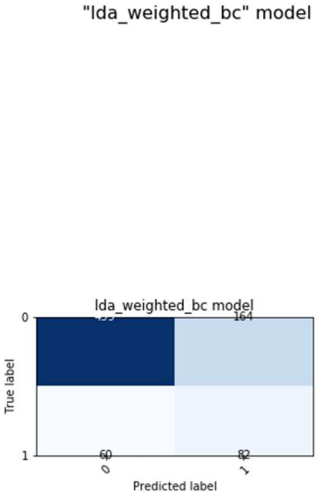

"usamp_reduced_bc" model scores
class: (0) bug(s)
class: (1) bug(s)
precision
recall
fbeta
accuracy
roc_auc

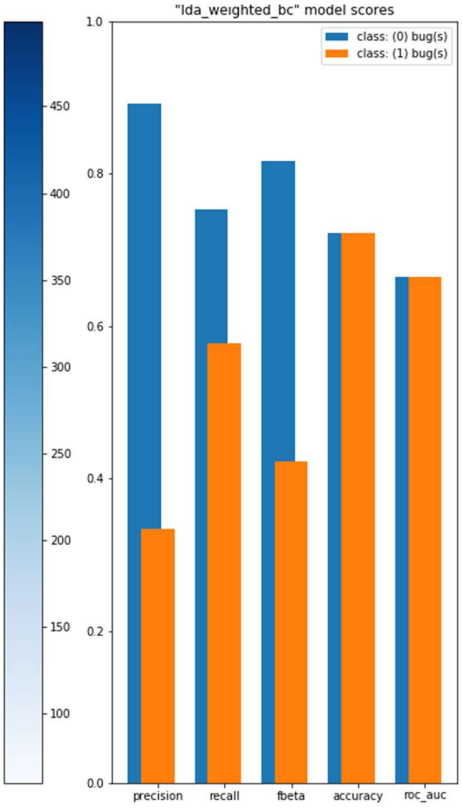

Training loss
Validation loss
checkpoints
0.80
0.75
0.70
0.65
0.60
0.55
0.50
0.45
0
200
400
600
800
1000

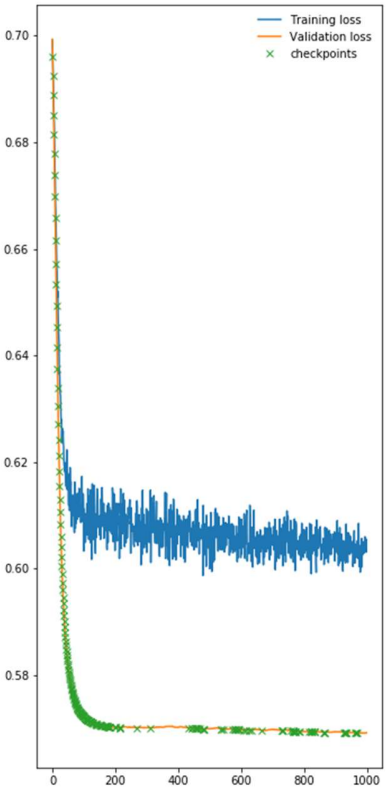

"lda_bc" model
lda_bc model
True label
Predicted label
0
1
18
111
31

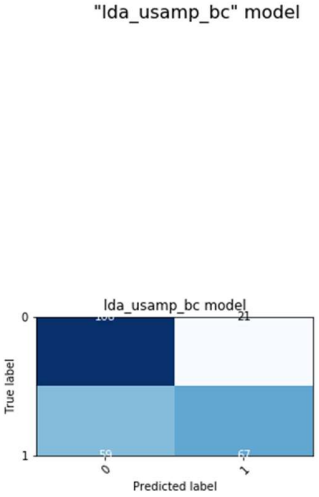

"lda_bc" model scores
class: (0) bug(s)
class: (1) bug(s)
precision
recall
fbeta
accuracy
roc_auc

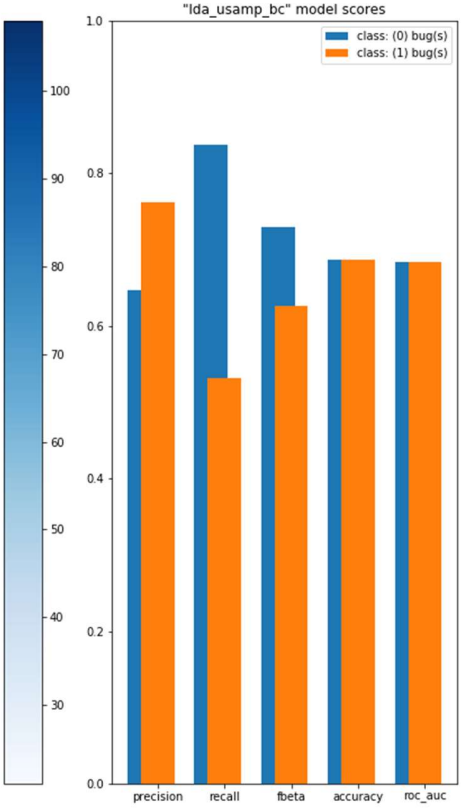

Training loss
Validation loss
checkpoints
0.70
0.68
0.66
0.64
0.62
0.60
0
200
400
600
800
1000

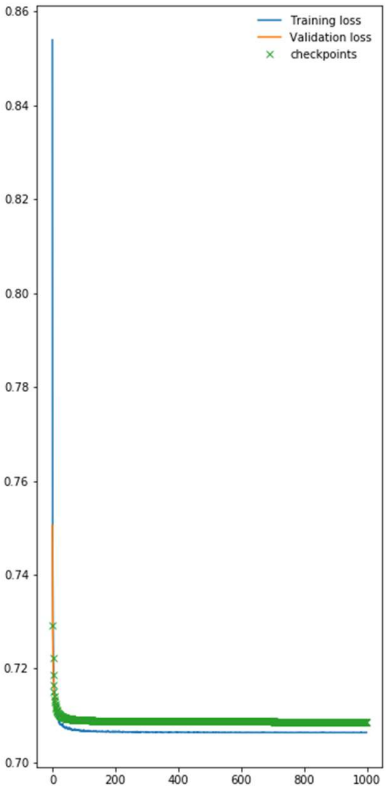

"lda_weighted_bc" model
lda_weighted_bc model
True label
Predicted label
0
1
164
60
82

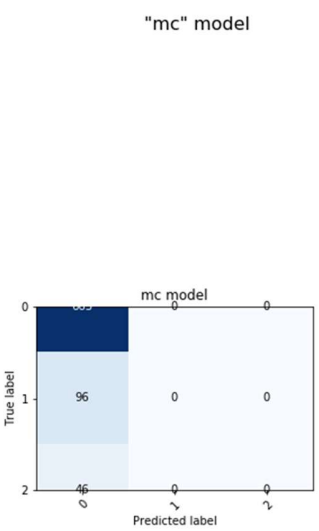

"lda_weighted_bc" model scores
class: (0) bug(s)
class: (1) bug(s)
precision
recall
fbeta
accuracy
roc_auc

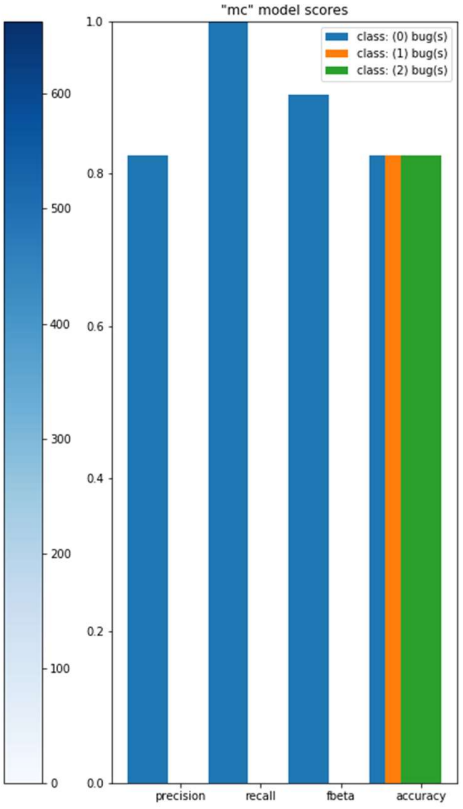

Training loss
Validation loss
checkpoints
0.70
0.68
0.66
0.64
0.62
0.60
0.58
0
200
400
600
800
1000

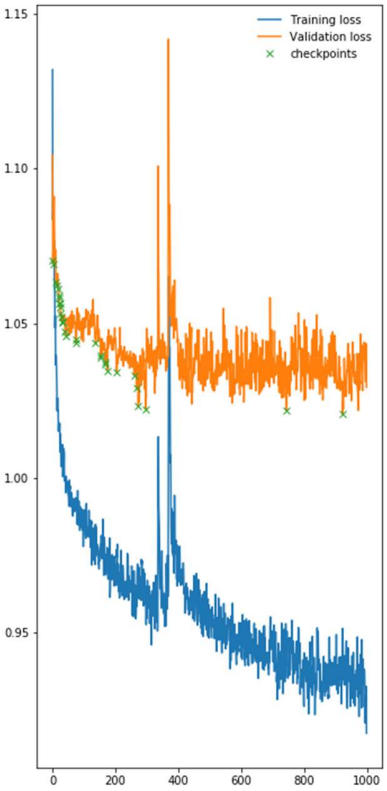

"lda_usamp_bc" model
lda_usamp_bc model
True label
Predicted label
0
1
21
59
67

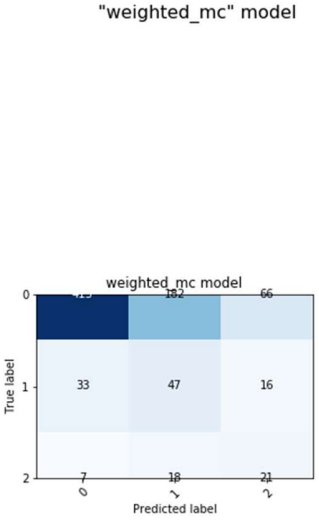

"lda_usamp_bc" model scores
class: (0) bug(s)
class: (1) bug(s)
precision
recall
fbeta
accuracy
roc_auc

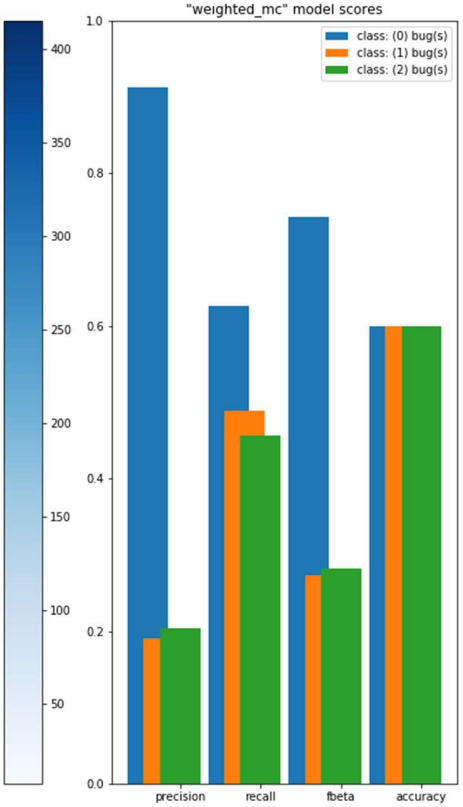

Training loss
Validation loss
checkpoints

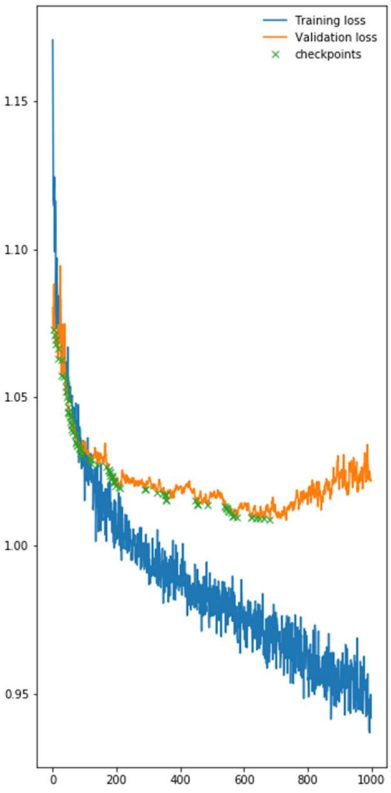

"mc" model
mc model
True label
Predicted label
96
46

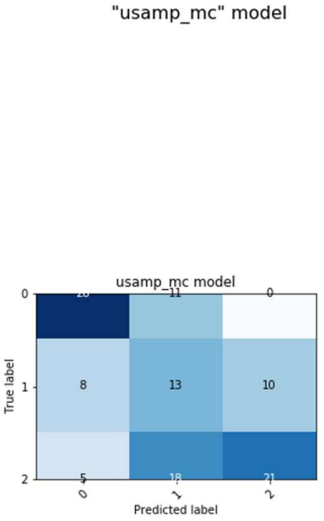

"mc" model scores
class: (0) bug(s)
class: (1) bug(s)
class: (2) bug(s)
precision
recall
fbeta
accuracy


Training loss
Validation loss
checkpoints

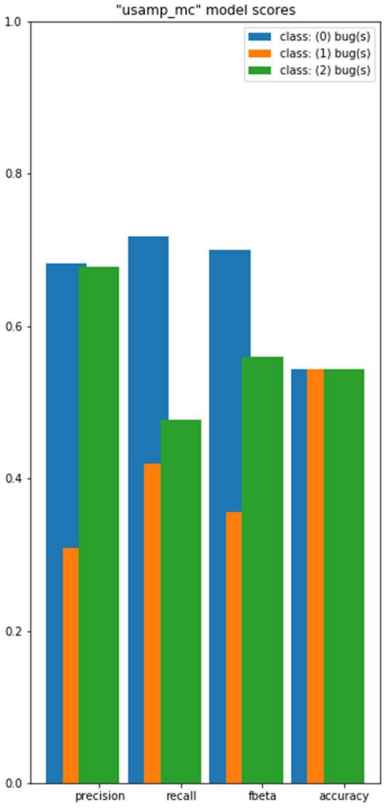

"weighted_mc" model
weighted_mc model
True label
Predicted label
33
47
16
7
18
21

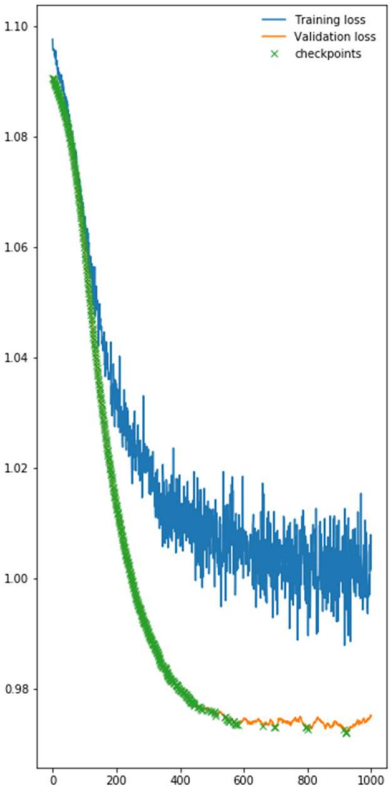

"weighted_mc" model scores
class: (0) bug(s)
class: (1) bug(s)
class: (2) bug(s)
precision
recall
fbeta
accuracy

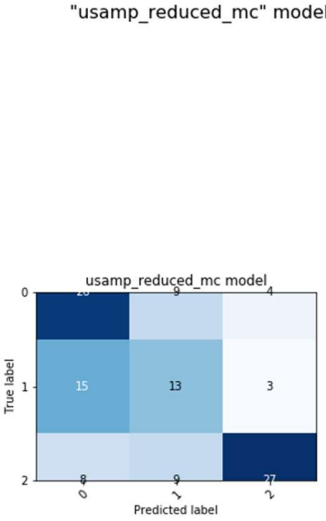

Training loss
Validation loss
checkpoints
1.15
1.10
1.05
1.00
0.95
0
200
400
600
800
1000


"usamp_mc" model
usamp_mc model
True label
Predicted label
0
1
2
11
0
8
13
10
5
18
21

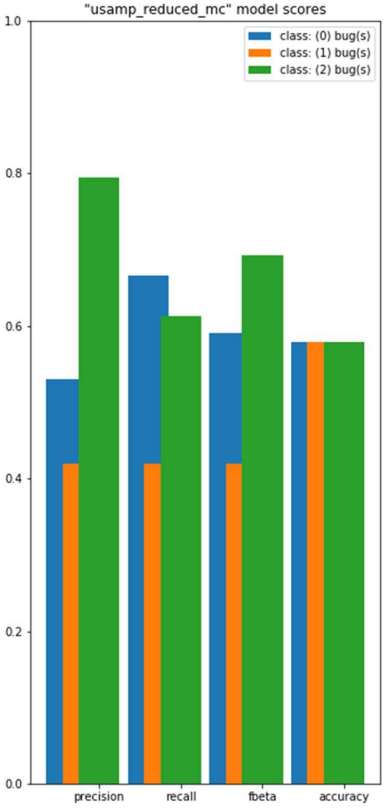

25
20
15
10
5
0

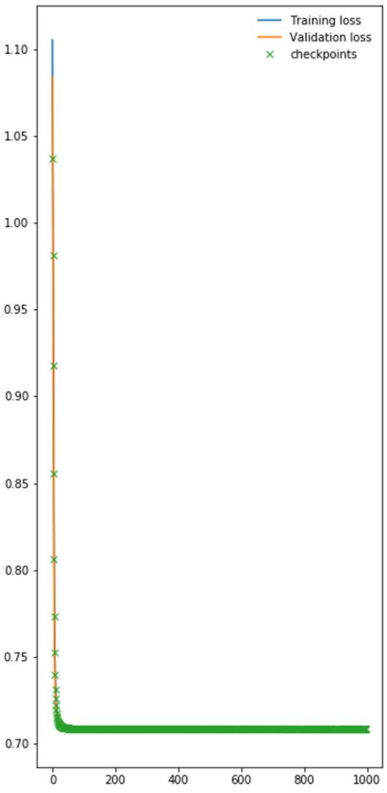

"usamp_mc" model scores
class: (0) bug(s)
class: (1) bug(s)
class: (2) bug(s)
1.0
0.8
0.6
0.4
0.2
0.0
precision
recall
fbeta
accuracy

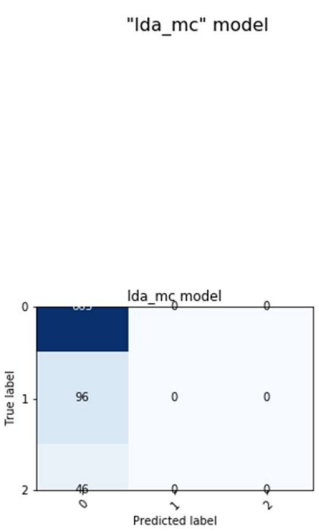

Training loss
Validation loss
checkpoints
1.10
1.08
1.06
1.04
1.02
1.00
0.98
0
200
400
600
800
1000

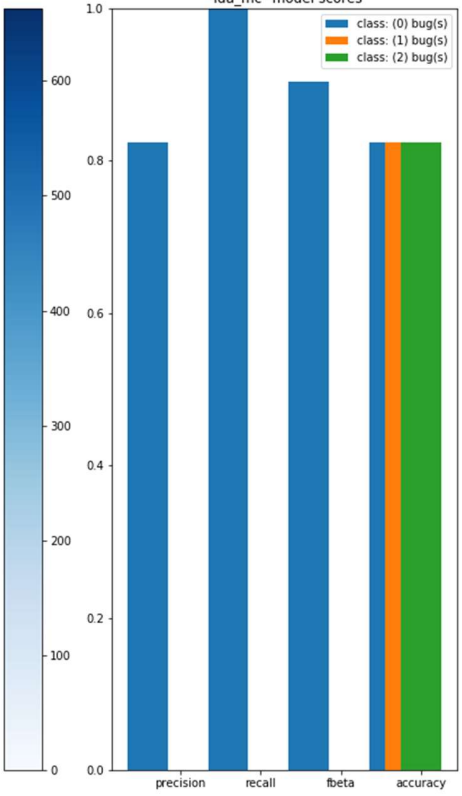

"usamp_reduced_mc" model
usamp_reduced_mc model
True label
Predicted label
0
1
2
9
4
15
13
3
8
9
27

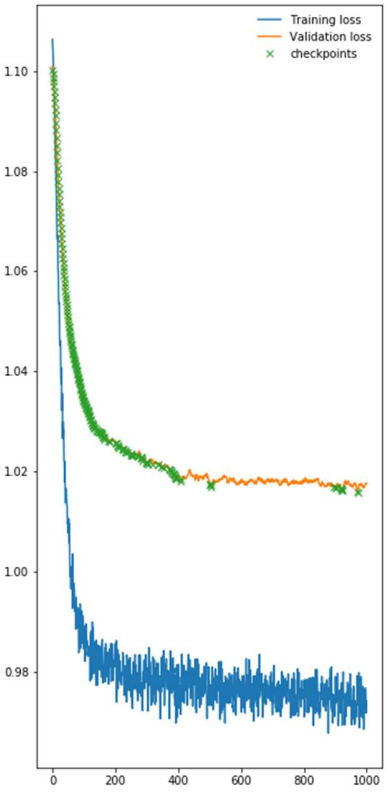

25
20
15
10
5
0

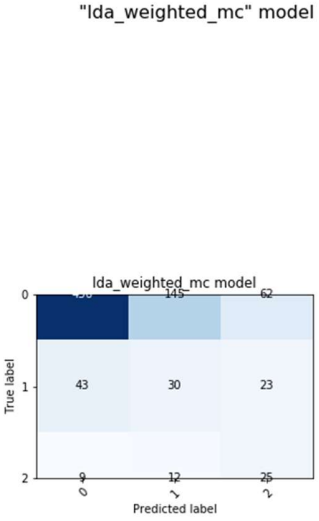

"usamp_reduced_mc" model scores
class: (0) bug(s)
class: (1) bug(s)
class: (2) bug(s)
1.0
0.8
0.6
0.4
0.2
0.0
precision
recall
fbeta
accuracy

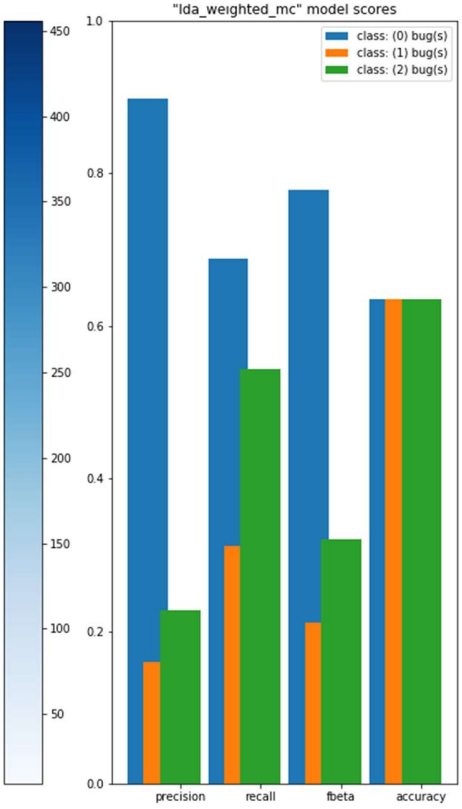

Training loss
Validation loss
checkpoints

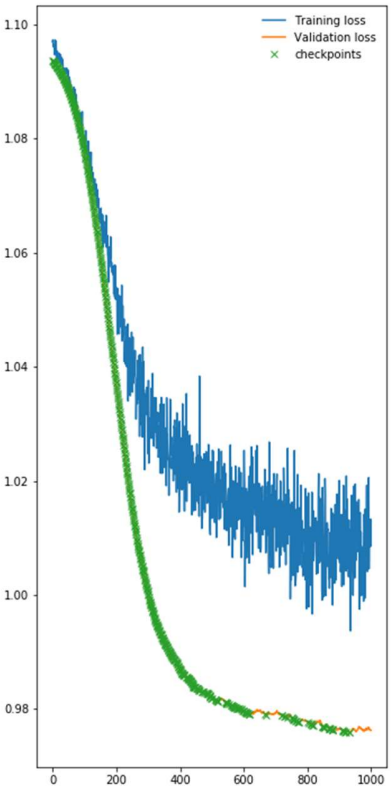

"lda_mc" model
lda_mc model
True label
Predicted label
96
46

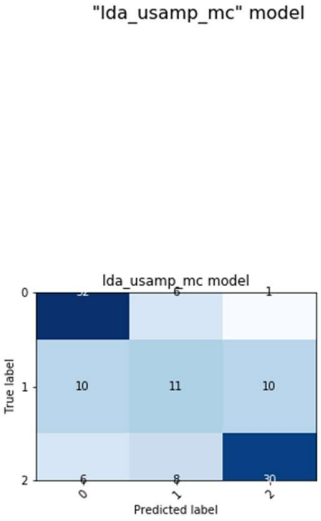

"lda_mc" model scores
class: (0) bug(s)
class: (1) bug(s)
class: (2) bug(s)
precision
recall
fbeta
accuracy

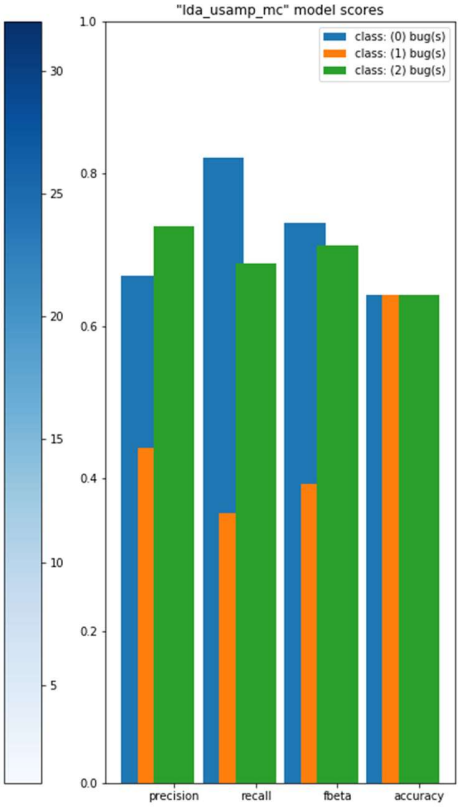

Training loss
Validation loss
checkpoints

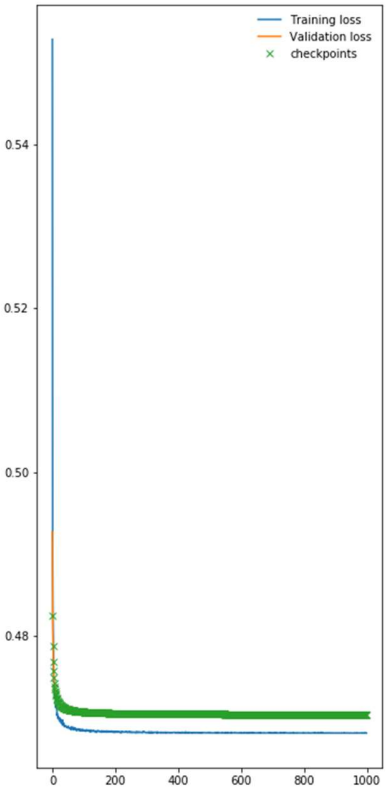

"lda_weighted_mc" model
lda_weighted_mc model
True label
Predicted label
145
62
43
30
23
9
12
25

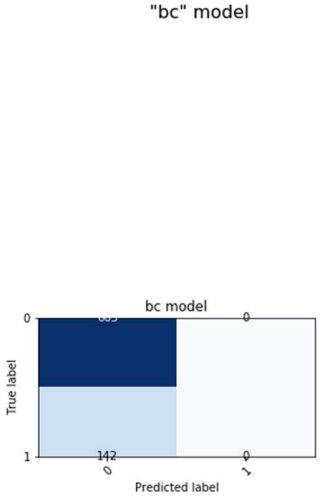

"lda_weighted_mc" model scores
class: (0) bug(s)
class: (1) bug(s)
class: (2) bug(s)
precision
recall
fbeta
accuracy

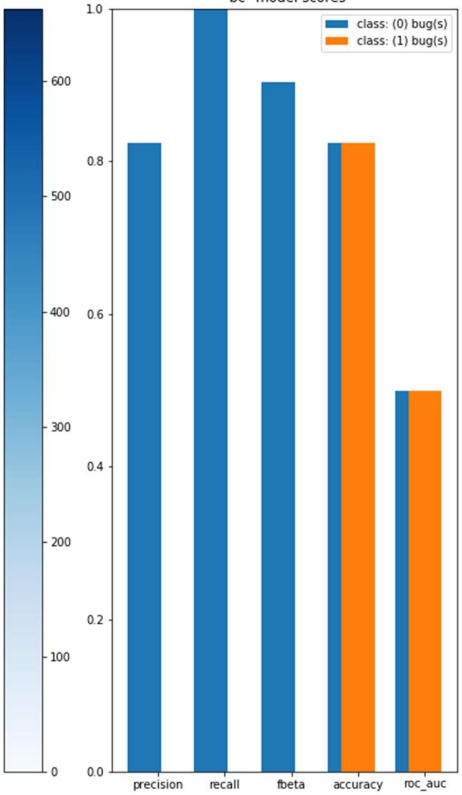

Training loss
Validation loss
checkpoints


"lda_usamp_mc" model
lda_usamp_mc model
True label
Predicted label


"lda_usamp_mc" model scores
class: (0) bug(s)
class: (1) bug(s)
class: (2) bug(s)
precision
recall
fbeta
accuracy


Training loss
Validation loss
checkpoints


"bc" model
bc model
True label
Predicted label


"bc" model scores
class: (0) bug(s)
class: (1) bug(s)
precision
recall
fbeta
accuracy
roc_auc