University of Bucharest
Faculty of Mathematics and Computer Science
Department of Computer Science

Artificial Intelligence Master
May 2020

Big Data Project Report
- Insolvency Analysis -

Supervisors
Bogdan Ichim
Ciprian Păduraru

Students
Matei Bejan
Tiberiu Marin
Florin Iordache
Andreea Cîrstoiu
David Ivan
Mihaela Nistor

Dataset

a. Overview

The main purpose of this project is the study of company insolvency in the pre-covid-19 economical setting. For this we have employed both data science and big data methodologies, in order to scale our models up to the size of the given datasets.

In this project we will compare two Python machine learning libraries: scikit-learn and Spark's MLlib, in order to assess which one offers the best scalability-versus-accuracy trade-off. Furthermore, we have added Apache's Hadoop DFS and Apache Kafka in our architecture in order to increase scalability and to be able to process real-time data streams, respectively.

The project has been developed in Python. The libraries used are: numpy, pandas, scikit-learn, pySpark and softimpute. The technologies we used are Apache Spark, HDFS and Kafka.

b. Data

Our data is formed of two datasets: Firme.csv and FirmeInInsolventa.csv and have been forwarded to us for this study by prof. Bogdan Ichim. Both datasets contain raw economic KPIs for romanian companies. Given that the current study occurs during the Covid-19 pandemic, the results cannot be applied to companies in the current economic setting, but may offer insight on the economy before Covid-19 and may be useful in the future, after the pandemic goes away.

Firme.csv contains 115 features collected over 5 years, labeled t0 to t4, thus 23 features per year per company.

FirmeInInsolventa.csv contains 92 features collected over 4 years - labeled t1 to t4 - and for companies that have declared insolvency.

In total there are a bit over 170 000 samples on both datasets.

As the data received is raw, data cleaning and feature engineering are mandatory for the models to provide accurate information. This will be discussed in the **Preprocessing** section.

c. Goal

We are challenged with a classification problem. We ought to build and train our models on both Firme.csv and FirmeInInsolventa.csv by using the data for years t1-t4 and apply our predictors on the t0-t3 data on Firme.csv.

Our goal for this project was to provide an accurate classifier by testing both the sklearn and Spark MLlib option, while also tackling the big data aspect by using appropriate technologies like HDFS and extending our application's use cases to real-time data flows with Kafka.

Preprocessing

The first step in our preprocessing pipeline was to prepare our data for the imputation phase.

We replaced all “-” values - which signify a missing element - with numpy.nan, so that our imputation techniques can properly detect the NaN values. We also decided to deal with categorical values via mode imputation and to map all non-numeric columns to integers.

d. Splitting

The data underwent a 60-30-10 train-validation-test split.

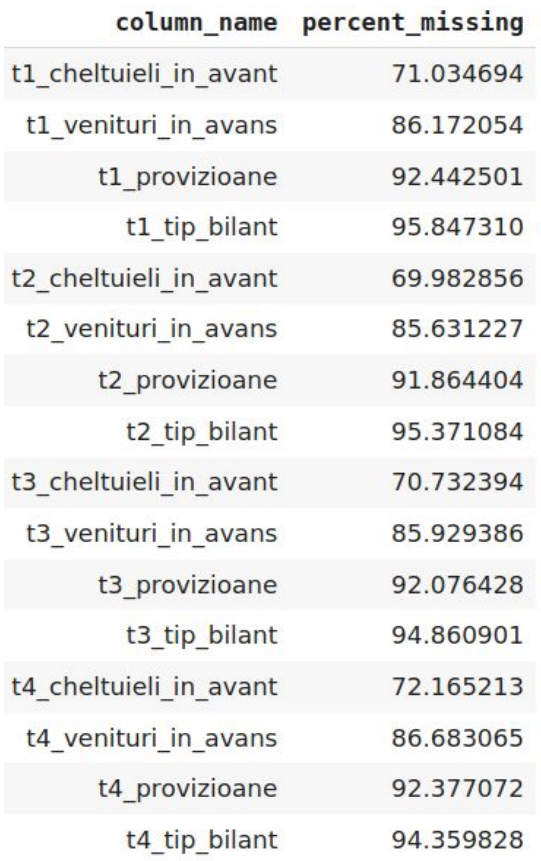
In this scenario, the test data represents the values on which our models are supposed to be called on (see section b. Data). However, since none of the companies in Firme.csv have declared insolvency, we have a unique label for all samples in the test set. That is why we decided to create a validation set, containing both types of labels, which can give us a better understanding of our models’ predictive power.

e. Dropping features with high NaN count

We decided on removing all features whose NaN percentage exceeded a 50% threshold. Our reasoning is that those features will fail to provide enough useful information for both the imputation and the prediction phases. In fact, regarding imputation the lack of information in this case would not only bring little to no useful data to our imputation methods, but it would damage the results of those which are more complex than simple mean or median NaN replacement.

This is why all those features represented in the table to the right right have been removed from the dataset.

Note that this image illustrates NaN counts for the training set. However, it is illustrative for both the validation and test sets, whose NaN counts we haven’t uploaded due to them being highly similar to the train set.



column_name	percent_missing
t1_cheltuieli_in_avant	71.034694
t1_venituri_in_avans	86.172054
t1_provizioane	92.442501
t1_tip_bilant	95.847310
t2_cheltuieli_in_avant	69.982856
t2_venituri_in_avans	85.631227
t2_provizioane	91.864404
t2_tip_bilant	95.371084
t3_cheltuieli_in_avant	70.732394
t3_venituri_in_avans	85.929386
t3_provizioane	92.076428
t3_tip_bilant	94.860901
t4_cheltuieli_in_avant	72.165213
t4_venituri_in_avans	86.683065
t4_provizioane	92.377072
t4_tip_bilant	94.359828

f. Missing data imputation

Please note that the imputation methods were applied separately on our three datasets and thus provide pristing real-life-like data (by which we mean that those values from the train set do not interfere in the test set and vice versa).

In total, we have produced 18 imputed datasets, 3 per imputation method.

i. Mean and median

Following this simple approach, we replaced all missing data in our train, validation and test sets with the mean or the median.

These are used relatively frequently and they can provide a good benchmark for the other imputation methods.

ii. K-Nearest Neighbours

This approach is based on Beretta and Santaniello's insight on the matter of nearest neighbour imputation. In their paper, the authors suggest that kNN gives an adequate trade-off between the precision of imputation and the ability to preserve the nature of the data when the maximum percentage of missing values lies between 15% and 30%. Furthermore, they note that a value of k = 3 has given the best results in respect to the precision-preservation trade-off mentioned above.

Since our own data rarely presented features that had an above-10% NaN count, we decided to use the same approach as Beretta and Santaniello suggest in their paper. That is, set k to 3.

iii. Softimpute

The softimputation method offered by the fancyimpute Python library is inherited from the R package with the same name.

Softimpute is an iterative method for matrix completion that uses nuclear-norm regularization. That being said, for this method to converge, a preliminary normalization of the data via fancyimpute's BiScaler procedure. One downside of bi-scaling is that it is not guaranteed to converge. However, it works well in practice.

Note that if we apply bi-scaling to our data, there is no way of re-scaling our data back to its original space. Moreover, it makes no sense to apply standard scaling or to normalize a second time.

g. Standardization and normalization

We applied both standardization and normalization to each of our imputed datasets, apart from the softimputed one, as it was already normalized via the bi-scaling procedure. The reason why we chose to do both types of data transformations is the uncertainty regarding the data's distribution, so we planned on trying both approaches.

h. Feature engineering

As we have previously specified, our data contains raw features. This being the case, we ought to process them and generate new features based on the ones.

Our main source of inspiration for calculating financial KPIs based on our data was Wikipedia: https://en.wikipedia.org/wiki/Financial_ratio.

The new features that have been added to the dataset in this phase are as follows: gross margin, profit margin, total assets, return on assets, return on net assets, debt ratio, current liquidity, immediate liquidity and asset turnover.

As a final step, we have removed those features used in computing the new features since they will only be increasing our data's dimensionality and therefore increasing our models' training time and lowering their accuracy. The dropped features are: cifra_de_afaceri_neta, profit_brut, profit_net, active_circulante, datorii, capital_social, stocuri and venituri_total.

Algorithms used

Scikit-Learn was a pioneer on this front. It provides an interface with which users can chain together their operations into a pipeline, where all preprocessing, feature creation, and feature indexing is done in stages. The pipeline can feed directly into a model, creating a unified workflow for both feature preprocessing and modeling. Scikit-Learn also provides users with a set of pre-baked preprocessing steps that are commonly used in machine learning pipelines, but also allows you to define your own steps for custom data transformations. Spark ML also uses the standard that Scikit-Learn pioneered, with the difference that its transformers do not modify data in place, they append a new column.

We started modelling by creating a parameter grid - a dictionary of lists - on which the best parameters will be searched on. We have then fed this list to the RandomisedSearchCV procedure, with 10 iterations and negative mean squared error as scoring metric. Other parameters that we have chosen are the number of concurrent workers, 100, so the search completes relatively fast, and the level of verbosity, 10, in order to trace back any error that might come up.

Another important parameter for the RandomisedSearchCV is the cross-validation option, for which I have employed the K-Fold algorithm with 10 folds. Using K-Fold helps reduce both underfitting, since the whole initial training set is used for training, and overfitting, since the whole training set is used for testing. Our choice of K is based on James' and Witten's proposal that the value of K should be either 5 or 10, since these values lead to a healthy bias-variance trade-off.

We chose the randomized grid search over its exhaustive counterpart because, even though the latter yearns the best set of parameters, the search space scales exponentially with more parameters or more choices for parameters adding to the grid.

Please note that only the sklearn algorithms underwent a hyperparameter search phase, as the MLlib does not offer any option to do that.

a. Modelling with Scikit-Learn

ii. SVM

SVM (Support Vector Machines) is a supervised model that analyzes the data through classification or regression. Support vector machine algorithms search a hyperplane in an N-dimensional space, where N is the number of features, that distinctly classifies the data points. For data that has just 2 features, the hyperplane will be a line and for 3 features, the hyperplane will be a two-dimensional plane. If the number of features is greater than 3, it becomes hard to imagine how the hyperplane looks. Data points on either side of the hyperplane can be attributed to different classes. SVM algorithm uses a regularization parameter (lambda) that serves as a degree of importance that is given to miss-classifications. This parameter helps maximizing the margin between both classes and minimizing the amount of miss-classifications.

In this project, the SVM algorithm was used for classification and for obtaining the best parameters for this dataset, it was used in RandomizedSearch.

The accuracies using SVM from Scikit-Learn, are between 80.3% and 97.1% (obtained with mean and soft normalization).

We didn't plot the ROC/AUC curve because it took too much time.

SVM confusion matrix figure.

iii. KNN

KNN (K-Nearest Neighbors) is a method used for classification and regression that uses for prediction the k closest training examples in the feature space. When it is used for classification, the output is the label of the majority in the group of k nearest neighbours, and when it is used for regression, the output is the average of this group of k values. To find the best number k of neighbours, it was used again in the Randomized Search, and we obtained the number 96.

The accuracies using KNN from Scikit-Learn, are between 82.6% and 96.8% (obtained with mean and knn normalization).

KNN confusion matrix figure.
KNN ROC/AUC plot figure.

iv. Random Forest

Random Forest algorithm, when it is used for classification, it is based on many classification trees. Every classification tree predicts a class for every point in the array of data based on the features and then the forest predicts the classes with the most votes from the trees.

We found the best parameters with Random Searched as being: the number of trees: 1450, the maximum depth: 460 and the function that measures the quality of a split : Gini (Gini measures the probability of a point to be wrongly classified).

The accuracies using the Random Forest classifier from Scikit-Learn, are between 89.1% and 96.4% (obtained with median and soft normalization).

Random Forest confusion matrix figure.

Random Forest ROC/AUC plot figure.

v. XGBoost

XGBoost is basically an acronym which stands for ‘eXtreme Gradient Boosting’. In order to use the XGBoost algorithm in our project, we used a library which is now part of sk-learn. The principle behind this library is that it uses gradient boosting to create an ensemble of ‘weak’ trees whose prediction scores are then summed up in order to get a final classification result. The tree ensemble model consists of a set of classification and regression trees (CART). As per the official documentation, XGBoost creates the same kind of model as the one in random forest, with the only difference being in the way the models are trained.

The ‘boosting’ part is a reference to the fact that the method uses lots of ‘weak’ learners by giving higher weights to the trees with bigger errors, in order to focus on those in the future iterations. The idea behind this is that the cases with bigger errors are the edge cases or the problem. The ‘gradient’ part refers to the gradient descent method used here, as well as in neural networks, to minimize the difference between the expected result and the actual obtained result, from one iteration to another. That’s how the algorithm knows how to modify the parameters of the trees at training time in order to obtain a model that best fits the data.

One thing worth mentioning is that XGBoost is really, really fast, unlike SVM, for example. The library was built from the beginning as very efficient and its creators had this desiderate in mind, to make it highly computationally effective. This proved very useful in this project because we had limited resources and time to actually run the algorithms, because we needed our computers for work.

As specified before, we used RandomizedSearch in order to find the best model for our problem. We chose the parameters used in the RandomizedSearch from the official documentation found here.

The results using boosted trees were very good, taking into consideration the time it took to train and test the model. All imputation methods had a percentage over 90% (with the highest being 98.2%, when the data was imputed using soft_norm), with the exception of the data imputed with KNN, which had a disappointing accuracy of just 25%.

XGBoost confusion matrix figure.

XGBoost ROC/AUC plot figure.

vi. Neural Network

The last method we used from the scikit-learning package was one based on neural networks. We used MLPClassifier, which stands for Multi Layer Perceptron Classifier. We chose a neural network with one hidden layer consisting of 30 neurons and with *tanh* as activation function and Limited Memory BFGS optimization algorithm. We chose this optimization algorithm for speed reasons, considering our limited home-resources.

The results seem to be the poorest of all of the algorithms presented in this section, with an accuracy between 74.9% and 79.4%. This suggests that the neural network was too simple for the complexity of the data, and that a more complex network should be used. However, we settled for this network because we wanted to see how a very simple model will perform on this rather complex dataset. Taking into consideration the really fast training time, we consider that this model is a good starting point and benchmark for the other models we analyzed as part of this assignment. This model can be thought of as being a prototype model.

Neural Network confusion matrix figure.
Neural Network ROC/AUC plot figure.

Model	Best accuracy score	Best imputation method
XGBoost	98.2 %	Soft normalized
SVM	97.1 %	Soft normalized
KNN	96.8 %	KNN normalized
Random Forest	96.4 %	Soft normalized
Neural Network	79.4 %	Soft normalized

i. Modelling with Spark MLlib

Spark uses DataFrame which is a Dataset organized into name columns. This concept is equivalent to a table in a relational database or a data frame in R/Python, but with significant optimizations under the hood. They can be constructed from sources like: structured data files, tables in Hive, external databases or existing RDDs.

We can think of a Spark DataFrame as a table distributed across a cluster and has functionality that is similar to dataframes in R and Pandas. For distributed computation using PySpark, we will have to perform operations on Spark dataframes, not usual Python data types. For ease of use, Pyspark provides a way to convert a Pandas dataframe into a Spark one and the other way around.

One of the key differences between Pandas and Spark dataframes is eager versus lazy execution. In PySpark, operations are delayed until a result is actually needed in the pipeline. For example, you can specify operations for loading a data set from S3 and applying a number of transformations to the dataframe, but these operations won't immediately be applied. Instead, a graph of transformations is recorded, and once the data is actually needed, for example when writing the results back to S3, then the transformations are applied as a single pipeline operation. This approach is used to avoid pulling the full data frame into memory and enables more effective processing across a cluster of machines. With Pandas dataframes, everything is pulled into memory, and every Pandas operation is immediately applied.

In general, it's a best practice to avoid eager operations in Spark if possible, since it limits how much of your pipeline can be effectively distributed.

Spark SQL is a Spark module for structured data processing. Unlike the basic Spark RDD API, the interfaces provided by Spark SQL provide Spark with more information about the structure of both the data and the computation being performed. Internally, Spark SQL uses this extra information to perform extra optimizations. There are several ways to interact with Spark SQL including SQL and the Dataset API. Spark SQL operations are very similar to the regular SQL instructions. We can select columns, filter the data or use aggregate functions.

i. LinearSVC

Should Spark ML and Sklearn behave the same? Yes. They are the same model and algorithm. They do provide slightly different APIs, especially in terms of default algorithm parameters, so comparing them is a delicate subject.

The models are essentially the same: linear model using hinge loss, supporting L1 and/or L2 regularization. MLlib provides a binary classifier in LinearSVC. To get a multiclass version, you can use OneVsRest with LinearSVC as the base estimator. sklearn takes the same approach, except that it automatically calls OneVsRest under the hood.

Several parameters a bit different as well:

- convergence tolerance and maxIter. They can be sensitive to scaling/preprocessing of data, so it can be good to tune them for each use of LinearSVC.
- regParam (MLlib) and C (sklearn): they are (scaled) inverses of each other. MLlib uses regParam to scale the penalty (regularization) term, whereas sklearn uses C to scale the loss (error) term. I.e., use a small regParam or big C to reduce regularization strength; use a big regParam or small C to increase regularization. These are (scaled) inverses of each other. The easiest way to get the best results for each implementation is to tune each separately.

- dual (sklearn param for solving the dual or not): This defaults to True; set this to False to match MLlib

  Linear SVC confusion matrix figure.

ii. Logistic Regression

Logistic regression is the appropriate regression analysis to conduct when the dependent variable is dichotomous (binary). It is the simplest and most extensively used statistical technique for predictive modelling analysis. Logistic regression is used to describe data and to explain the relationship between one dependent binary variable and one or more nominal, ordinal, interval or ratio-level independent variables.

Logistic regression confusion matrix figure.
Logistic regression ROC/AUC plot figure.

iii. Random Forest

Random Forests are algorithms for learning ensembles of decision trees, in parallel.

They can be less prone to overfitting, meaning that the greater the number of trees we use, the smaller likelihood of overfitting. Also, Random Forests can be easier to tune, considering the fact that the performance improves monotonically with the number of trees.

Random forest confusion matrix figure.
Random forest ROC/AUC plot figure.

iv. Boosted Trees Classifier

Gradient-Boosted Trees are ensembles of decision trees. The idea behind them is to iteratively train decision trees in order to minimize a loss function. Just like decision trees, gradient-boosted trees handle categorical features, extend to the multiclass classification setting and there is no need for feature scaling because they are able to capture non-linearities and feature interactions.
The main difference between them and random forests is that GBT’s train one tree at a time, therefore they can take longer time to train. This happens because the process of boosting takes the residuals of each tree and trains the new tree using them. Because of this, training smaller trees using GBT rather than large trees like in Random forests is quite reasonable

Boosted trees confusion matrix figure.

Boosted trees ROC/AUC plot figure.

v. Neural Network

Multilayer perceptron classifier (MLPC) is a classifier based on the feedforward artificial neural network. MLPC consists of multiple layers of nodes. Each layer is fully connected to the next layer in the network. Nodes in the input layer represent the input data. All other nodes map inputs to outputs by a linear combination of the inputs with the node’s weights w and bias b and applying an activation function.A feedforward neural network is an artificial neural network where connections between the nodes do not form a cycle.

Each layer has sigmoid activation function and the output one has softmax. Number of inputs has to be equal to the size of feature vectors. Number of outputs has to be equal to the total number of labels.

We used one hidden layer with 30 neurons.

Neural network confusion matrix figure.

Neural network ROC/AUC plot figure.

vi. Decision Trees

The decision tree is one of the most powerful and popular tools for classification and prediction. Each node represents a test on an attribute and each branch is an outcome of the test and each leaf node holds a class label. Its learning process is called recursive partitioning and the idea behind it is to split the source set into subsets based on an attribute value test. This is repeated on each derived subset. This whole process is completed when the subsets of the nodes have the value of the target variable, when splitting no longer adds value to the predictions.

Advantages:

- decision trees are able to generate understandable rules
- support for both continuous and categorical variables
- the usage of a white box model
- they can handle high dimensional data

Disadvantages:

- they are prone to errors in classification when using a small number of training examples
- high computational price to train
- they are unstable, therefore a small change in the data can cause a significant change in its structure
- relatively inaccurate. A solution to this is using a random forest of decision trees, but it would be far more complicated to interpret.

Decision trees confusion matrix figure.

Decision trees ROC/AUC plot figure.

Model	Best accuracy score	Best imputation method
Gradient Boosted Trees	95.36%	Soft normalized
Decision Trees	95.34%	Mean normalized
Random Forest	95.08%	Soft normalized
Neural Network	85.58%	Mean scaled
Logistic Regression	82.43%	Mean normalized
Linear SVC	77.62%	Mean scaled

Data Engineering

Apache Hadoop

Hadoop is the most used system for big data applications. Its key features are: open source, fast, low-cost, fault tolerant and configurable. They help with large-scale, parallel and distributed data processing.

The architecture of Hadoop revolves around two major components:
distributed computing, which is handled by HDFS, and parallel processing, which is handled by MapReduce. We can say that Hadoop is a combination of these two.

The open source Hadoop is maintained by the Apache Software Foundation, and can be deployed in 3 separate modes:

- Standalone, for simple analysis or debugging
- Pseudo distributed, for simulation of multi-node installations on a single node. Each of the workers run in a separate JVM.
- Distributed, cluster with multiple worker nodes

HDFS

Hadoop Distributed File System is the default storage filesystem in Hadoop, which is distributed, considerably simple in design and extremely scalable, flexible, and with high fault tolerance capability. HDFS architecture has a master-slave pattern due to which the slave nodes can be better managed and utilized. HDFS is designed to be best suited for MapReduce programming.

Main features of HDFS

- Scalability: scalable to petabytes (1 million GBs) and flexible to add or remove nodes
- Reliability and fault tolerance: HDFS replicates the data by storing it in multiple nodes, so if some nodes are down, data can be accessed from other available nodes.
- Data coherency: HDFS uses the Write Once, Read Many model
- Hardware failure recovery
- Portability
- Computation closer to odata: HDFS moves the computation process toward data, which is faster than pulling data out for computation

Architecture

HDFS is managed by daemon processes, from which we mention:

- NameNode
  - The master process daemon server that coordinates all the operations related to storage in Hadoop, including read and writes in HDFS.
  - Manages the filesystem namespace, by holding the metadata above all the file blocks, and in which all nodes of data blocks are present in the cluster.
  - Does not store data. Instead, it caches data and stores metadata in RAM for faster access, so it requires a system with high RAM.
- DataNode
  - Holds the actual data in HDFS and is responsible for creating, deleting and replicating data blocks, as assigned by NameNode.
  - Sends messages to NameNode, which are called as heartbeat in a periodic interval. If a DataNode fails to send the heartbeat message, then NameNode will mark it as a dead node.
  - If the file data present in the DataNode becomes less than the replication factor, then NameNode replicates the file data to other DataNodes.

Data storage in HDFS

In HDFS, files are divided in blocks, which are stored in multiple DataNodes, and their metadata is stored in NameNode. Blocks have a configurable size, with the default of 128 MBytes in newer versions of Hadoop. Block size is high to minimize the cost of disk seek rate, leverage transfer rate and reduce the metadata size in NameNode for a file.

Each file divided into blocks is stored in multiple DataNodes, and the number of replication factors can be configured. The default value is 3. The higher the number of the replication factor, the system will be highly fault tolerant and will occupy that many numbers of time the file is saved, and also increase the metadata in NameNode. This value should be balanced.

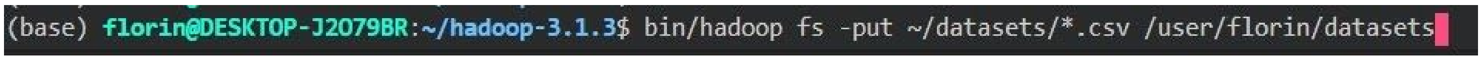
HDFS commands

The Hadoop command line environment is Linux-like. The Hadoop filesystem provides various shell commands to perform operations like copying files and viewing contents of folders.

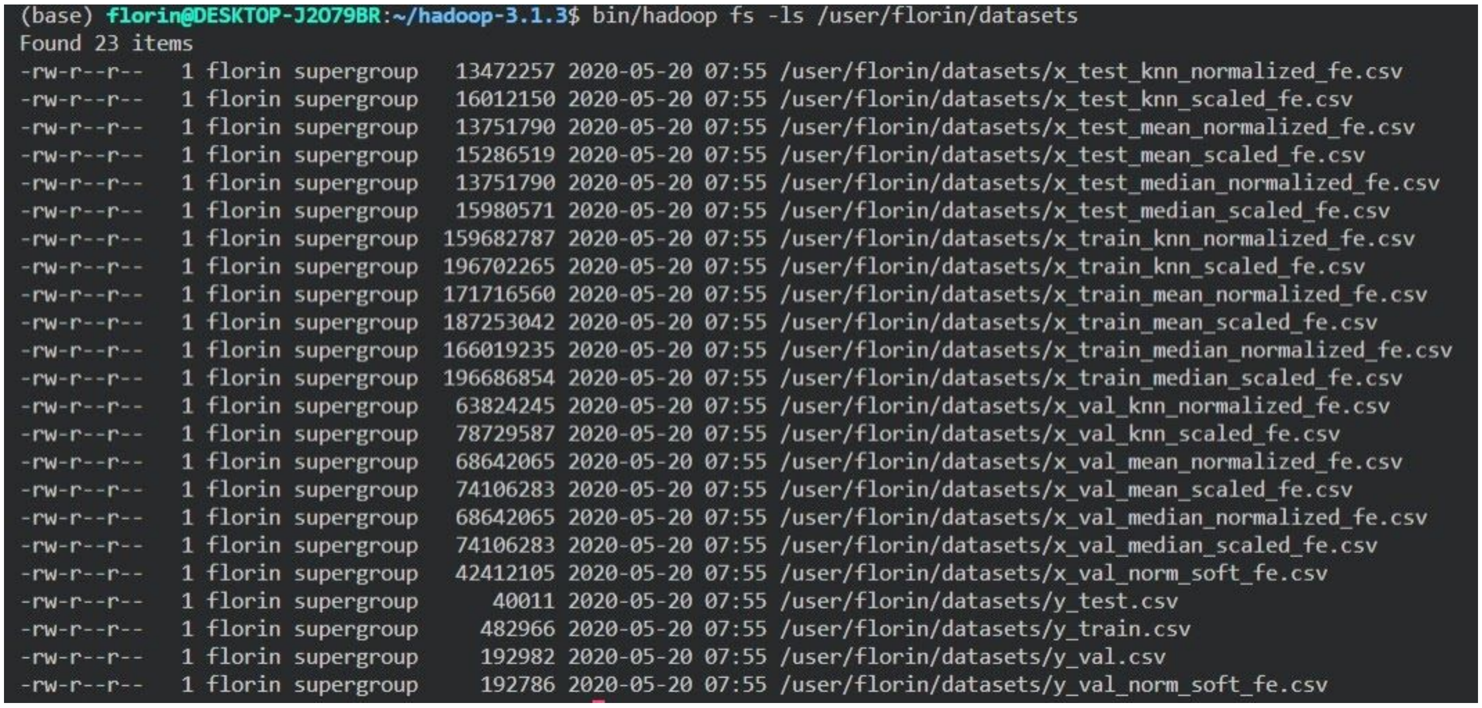
The syntax of Hadoop filesystem shell commands is: hadoop fs <args>

1. Creating a folder: hadoop fs -mkdir <paths>
2. Listing contents of a folder: hadoop fs -ls <args>
3. Adding data to HDFS from local computer: hadoop fs -put <local_src> <hdfs_dest_path>
4. Downloading data from HDFS to local computer: hadoop fs -get <hdfs_src> <local_dest_path>
5. Viewing content of a file: hadoop fs -cat <path_filename>

Sample HDFS put command and its result:



```
(base) florin@DESKTOP-J2079BR:~/hadoop-3.1.3$ bin/hadoop fs -put ~/datasets/*.csv /user/florin/datasets
```



```
(base) florin@DESKTOP-J2079BR:~/hadoop-3.1.3$ bin/hadoop fs -ls /user/florin/datasets
Found 23 items
-rw-r--r--   1 florin supergroup   13472257 2020-05-20 07:55 /user/florin/datasets/x_test_knn_normalized_fe.csv
-rw-r--r--   1 florin supergroup   16012150 2020-05-20 07:55 /user/florin/datasets/x_test_knn_scaled_fe.csv
-rw-r--r--   1 florin supergroup   13751790 2020-05-20 07:55 /user/florin/datasets/x_test_mean_normalized_fe.csv
-rw-r--r--   1 florin supergroup   15286519 2020-05-20 07:55 /user/florin/datasets/x_test_mean_scaled_fe.csv
-rw-r--r--   1 florin supergroup   13751790 2020-05-20 07:55 /user/florin/datasets/x_test_median_normalized_fe.csv
-rw-r--r--   1 florin supergroup   15980571 2020-05-20 07:55 /user/florin/datasets/x_test_median_scaled_fe.csv
-rw-r--r--   1 florin supergroup  159682787 2020-05-20 07:55 /user/florin/datasets/x_train_knn_normalized_fe.csv
-rw-r--r--   1 florin supergroup  196702265 2020-05-20 07:55 /user/florin/datasets/x_train_knn_scaled_fe.csv
-rw-r--r--   1 florin supergroup  171716560 2020-05-20 07:55 /user/florin/datasets/x_train_mean_normalized_fe.csv
-rw-r--r--   1 florin supergroup  187253042 2020-05-20 07:55 /user/florin/datasets/x_train_mean_scaled_fe.csv
-rw-r--r--   1 florin supergroup  166019235 2020-05-20 07:55 /user/florin/datasets/x_train_median_normalized_fe.csv
-rw-r--r--   1 florin supergroup  196686854 2020-05-20 07:55 /user/florin/datasets/x_train_median_scaled_fe.csv
-rw-r--r--   1 florin supergroup   63824245 2020-05-20 07:55 /user/florin/datasets/x_val_knn_normalized_fe.csv
-rw-r--r--   1 florin supergroup   78729587 2020-05-20 07:55 /user/florin/datasets/x_val_knn_scaled_fe.csv
-rw-r--r--   1 florin supergroup   68642065 2020-05-20 07:55 /user/florin/datasets/x_val_mean_normalized_fe.csv
-rw-r--r--   1 florin supergroup   74106283 2020-05-20 07:55 /user/florin/datasets/x_val_mean_scaled_fe.csv
-rw-r--r--   1 florin supergroup   68642065 2020-05-20 07:55 /user/florin/datasets/x_val_median_normalized_fe.csv
-rw-r--r--   1 florin supergroup   74106283 2020-05-20 07:55 /user/florin/datasets/x_val_median_scaled_fe.csv
-rw-r--r--   1 florin supergroup   42412105 2020-05-20 07:55 /user/florin/datasets/x_val_norm_soft_fe.csv
-rw-r--r--   1 florin supergroup      40011 2020-05-20 07:55 /user/florin/datasets/y_test.csv
-rw-r--r--   1 florin supergroup     482966 2020-05-20 07:55 /user/florin/datasets/y_train.csv
-rw-r--r--   1 florin supergroup     192982 2020-05-20 07:55 /user/florin/datasets/y_val.csv
-rw-r--r--   1 florin supergroup     192786 2020-05-20 07:55 /user/florin/datasets/y_val_norm_soft_fe.csv
```

MapReduce

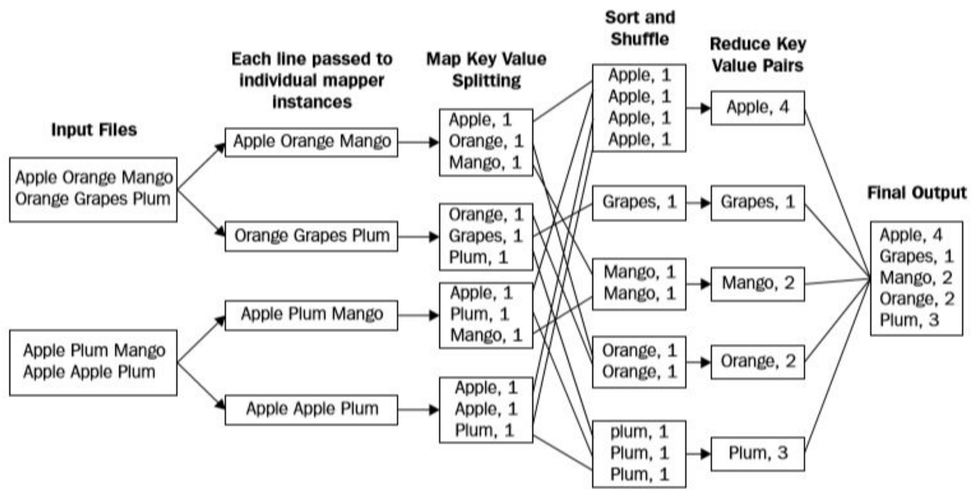
MapReduce is a massive parallel processing framework that processes faster, scalable and fault tolerant data in a distributed environment. It can process a large volume of data in parallel, by dividing a task into independent sub-tasks and also has a master-slave architecture.

Input, output and intermediary results in a MapReduce job are in the form of <Key, Value> pair. Key and Value have to be serializable.

MapReduce architecture

Uses the following two daemon processes:

- JobTracker


Input Files
Apple Orange Mango Orange Grapes Plum
Apple Plum Mango Apple Apple Plum
Each line passed to individual mapper instances
Apple Orange Mango
Orange Grapes Plum
Apple Plum Mango
Apple Apple Plum
Map Key Value Splitting
Apple, 1 Orange, 1 Mango, 1
Orange, 1 Grapes, 1 Plum, 1
Apple, 1 Plum, 1 Mango, 1
Apple, 1 Apple, 1 Plum, 1
Sort and Shuffle
Apple, 1 Apple, 1 Apple, 1 Apple, 1
Grapes, 1
Mango, 1 Mango, 1
Orange, 1 Orange, 1
plum, 1 Plum, 1 Plum, 1
Reduce Key Value Pairs
Apple, 4
Grapes, 1
Mango, 2
Orange, 2
Plum, 3
Final Output
Apple, 4 Grapes, 1 Mango, 2 Orange, 2 Plum, 3

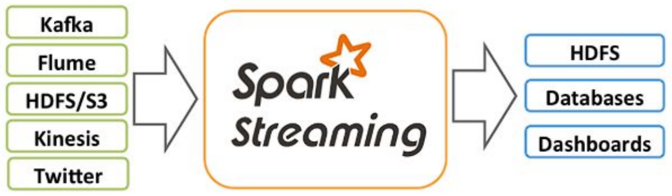
    - Master coordinator daemon process, responsible for coordinating and completing a MapReduce job in Hadoop.
    - Primary functions of JobTracker are resource management, tracking resource availability and task process cycle.
    - It identifies the TaskTracker to perform certain tasks and monitors the progress and status of a task.
- TaskTracker
    - Is the slave daemon process that performs a task assigned by JobTracker.
    - Sends heartbeat messages to JobTracker periodically to notify about the free slots and sends the status to JobStracker about the task, and checks if any task has to be performed.

The principle of MapReduce can be seen in this example, in which we want to count the number of words of a file divided into blocks.

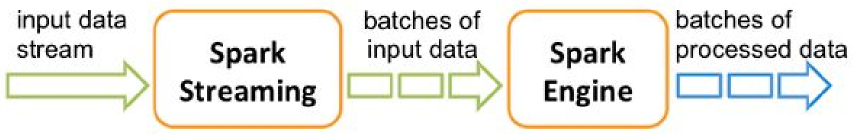
Spark

Apache Spark Streaming

Stream data processing is used when dynamic data is generated continuously, and it is often found in big data use cases. In most instances data is processed in near-real time, one record at a time, and the insights derived from the data are also used to provide alerts, render dashboards, and feed machine learning models that can react quickly to new trends within the data.



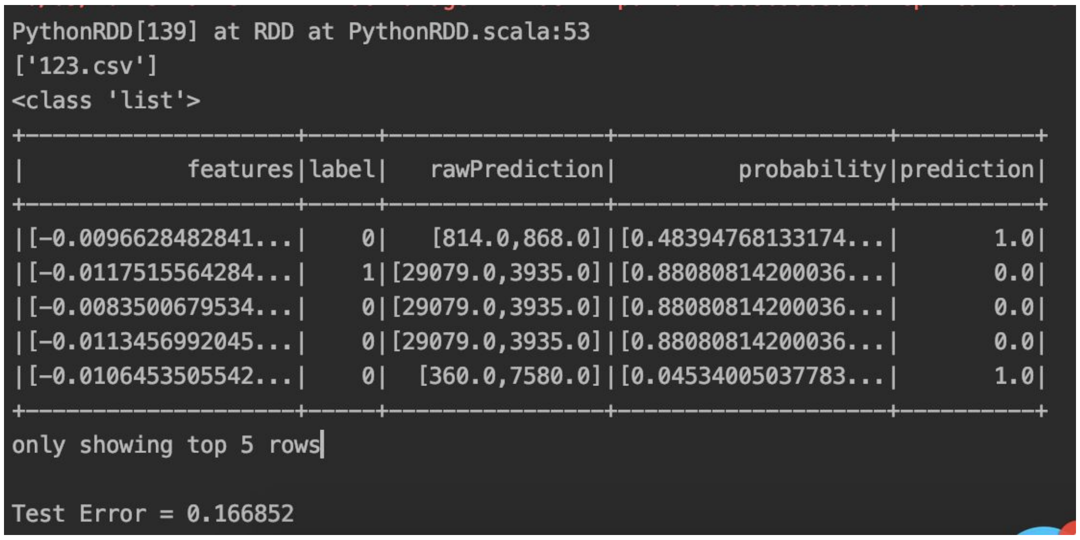
Spark streaming is based on the idea of discretized streams (DStreams). Each discretized stream is represented as a sequence of RDD. Since the alpha version of Spark (0.7.0) there have been many improvements undergoing because it's a very low-level API.



As a solution to the challenges, Spark Structured Streaming was introduced in Spark 2.0 (and became stable in 2.2) as an extension built on top of Spark SQL. Because of that, it takes advantage of Spark SQL code and memory optimizations. Structured Streaming also gives very powerful abstractions like Dataset/DataFrame APIs as well as SQL. No more dealing with RDD directly.

Both Structured Streaming and Streaming with DStreams use micro-batching(streaming's receivers accept data in parallel and buffer it in the memory of Spark's workers nodes). The biggest difference is latency and message delivery guarantees: Structured Streaming offers exactly-once delivery with 100+ milliseconds latency, whereas the Streaming with DStreams approach only guarantees at-least-once delivery, but can provide millisecond latencies. Unlike the traditional continuous operator model, where the computation is statically allocated to a node, Spark tasks are assigned dynamically to the workers based on the locality of the data and available resources.

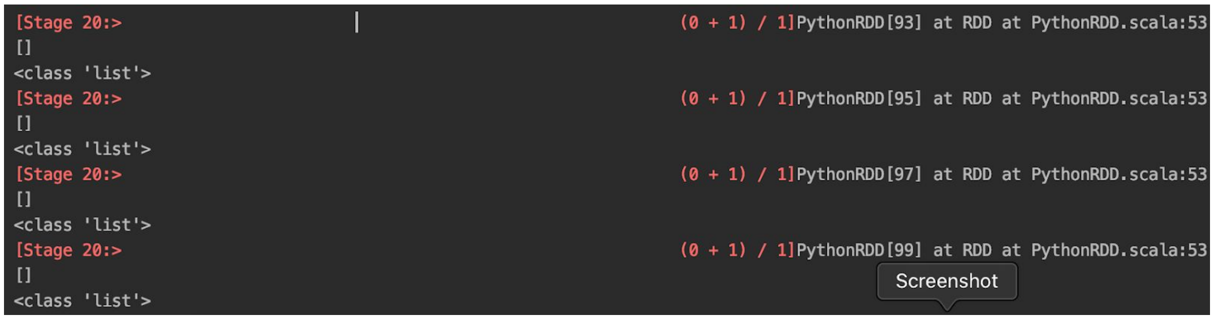
Basic streaming demo



```
PythonRDD[139] at RDD at PythonRDD.scala:53
['123.csv']
<class 'list'>
+--------------------+-----+----------------+--------------------+----------+
|            features|label|   rawPrediction|         probability|prediction|
+--------------------+-----+----------------+--------------------+----------+
|[-0.0096628482841...|    0|   [814.0,868.0]|[0.48394768133174...|       1.0|
|[-0.0117515564284...|    1|[29079.0,3935.0]|[0.88080814200036...|       0.0|
|[-0.0083500679534...|    0|[29079.0,3935.0]|[0.88080814200036...|       0.0|
|[-0.0113456992045...|    0|[29079.0,3935.0]|[0.88080814200036...|       0.0|
|[-0.0106453505542...|    0|  [360.0,7580.0]|[0.04534005037783...|       1.0|
+--------------------+-----+----------------+--------------------+----------+
only showing top 5 rows

Test Error = 0.166852
```

Data frame created after parsing the .csv file and making predictions using DecisionTreeClassifier which had been already trained.



```
[Stage 20:>                                    (0 + 1) / 1]PythonRDD[93] at RDD at PythonRDD.scala:53
[]
<class 'list'>
[Stage 20:>                                    (0 + 1) / 1]PythonRDD[95] at RDD at PythonRDD.scala:53
[]
<class 'list'>
[Stage 20:>                                    (0 + 1) / 1]PythonRDD[97] at RDD at PythonRDD.scala:53
[]
<class 'list'>
[Stage 20:>                                    (0 + 1) / 1]PythonRDD[99] at RDD at PythonRDD.scala:53
[]
<class 'list'>
```

Results comparison

Sklearn

We can say that we solved this binary classification problem using the sklearn framework for machine learning because we can see that most of the results obtained using the algorithms presented above are over 90%.

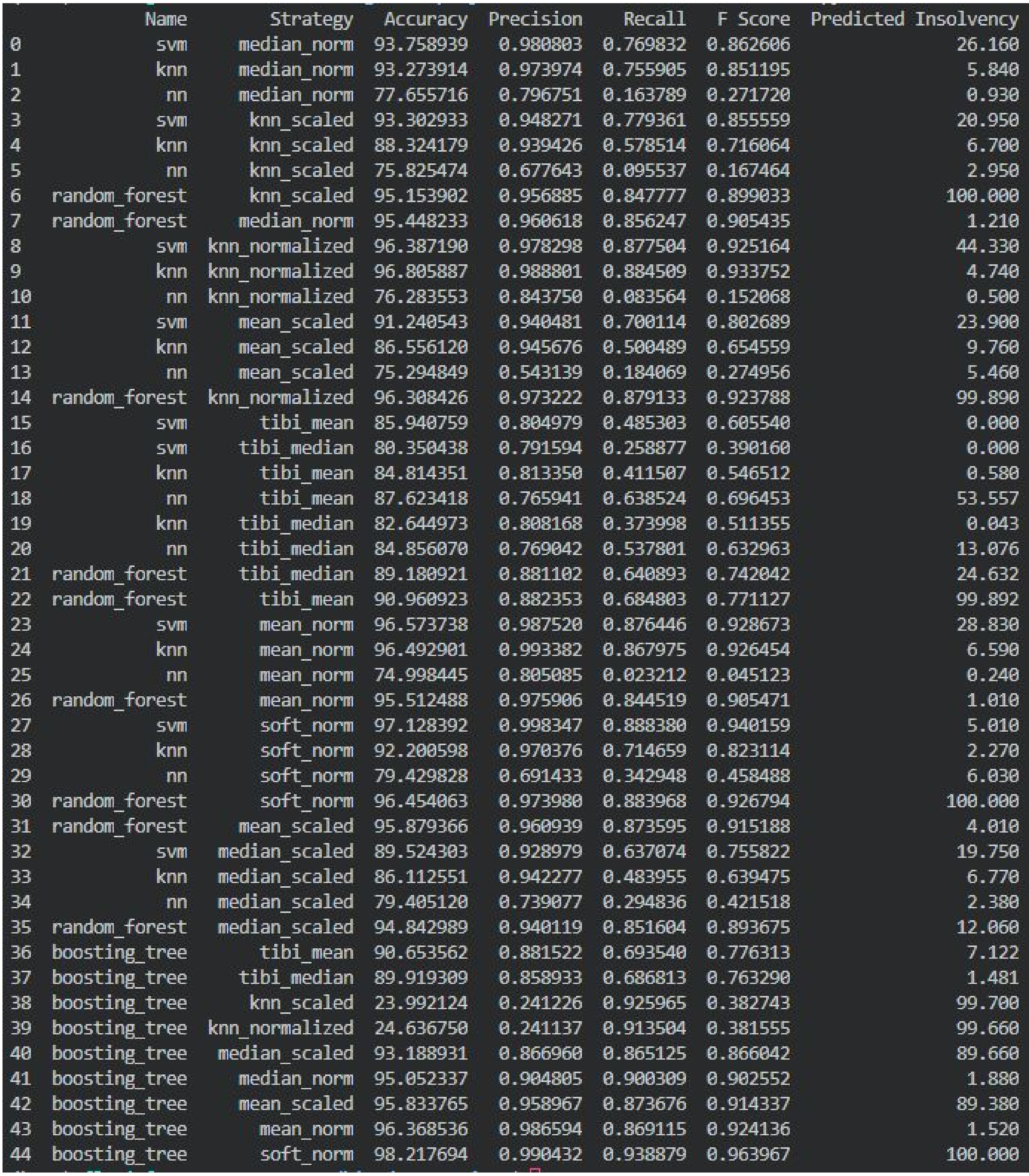
Another indicator that the problem is solved can be deducted from the confusion matrices in the annexes below and from the roc curves. In the majority of cases, the main diagonal of the confusion matrix (containing the true positives and the true negatives) has higher numbers than the secondary diagonal. This means that our model correctly predicts most of the examples. This fact is true for the ROC curves as well, where we can see that the curves are over the main diagonal of the plot in most of the cases.

There are 2 notable outliers, the boosting tree method on knn-based imputations, which have a very low accuracy (23% and 24%), which are way lower than the third lowest-scoring algorithm - neural network with mean_norm, which has an accuracy of 74,9%.

Despite the 2 outliers, the boosting tree method has the highest accuracy of all the tests when used with the soft-norm imputed dataset - **98.2%**.

We can see that the soft_norm imputation strategy works best on this dataset. As it can be seen in the chapter below, the KNN-based imputation strategies give the poorest results. This might be due to the fact that, given the computational demands of KNN imputation, neither our own machines, nor Google Colab could apply KNN imputation on the whole train test in a single run. We were thus forced to split the train set into approximately 30 000-sample subsets, run the imputation on those and then concatenate them.

As for the prediction algorithms, the neural network returns the worst results. This can be explained by the fact that we considered the neural network as a prototyping model, and as a starting point and benchmark for the other algorithms. The methods working with trees (XGBoost and Random Forest) give the best results.



```
             Name        Strategy   Accuracy  Precision    Recall   F Score  Predicted Insolvency
0             svm     median_norm  93.758939   0.980803  0.769832  0.862606                26.160
1             knn     median_norm  93.273914   0.973974  0.755905  0.851195                 5.840
2              nn     median_norm  77.655716   0.796751  0.163789  0.271720                 0.930
3             svm      knn_scaled  93.302933   0.948271  0.779361  0.855559                20.950
4             knn      knn_scaled  88.324179   0.939426  0.578514  0.716064                 6.700
5              nn      knn_scaled  75.825474   0.677643  0.095537  0.167464                 2.950
6   random_forest      knn_scaled  95.153902   0.956885  0.847777  0.899033               100.000
7   random_forest     median_norm  95.448233   0.960618  0.856247  0.905435                 1.210
8             svm  knn_normalized  96.387190   0.978298  0.877504  0.925164                44.330
9             knn  knn_normalized  96.805887   0.988801  0.884509  0.933752                 4.740
10             nn  knn_normalized  76.283553   0.843750  0.083564  0.152068                 0.500
11            svm     mean_scaled  91.240543   0.940481  0.700114  0.802689                23.900
12            knn     mean_scaled  86.556120   0.945676  0.500489  0.654559                 9.760
13             nn     mean_scaled  75.294849   0.543139  0.184069  0.274956                 5.460
14  random_forest  knn_normalized  96.308426   0.973222  0.879133  0.923788                99.890
15            svm       tibi_mean  85.940759   0.804979  0.485303  0.605540                 0.000
16            svm     tibi_median  80.350438   0.791594  0.258877  0.390160                 0.000
17            knn       tibi_mean  84.814351   0.813350  0.411507  0.546512                 0.580
18             nn       tibi_mean  87.623418   0.765941  0.638524  0.696453                53.557
19            knn     tibi_median  82.644973   0.808168  0.373998  0.511355                 0.043
20             nn     tibi_median  84.856070   0.769042  0.537801  0.632963                13.076
21  random_forest     tibi_median  89.180921   0.881102  0.640893  0.742042                24.632
22  random_forest       tibi_mean  90.960923   0.882353  0.684803  0.771127                99.892
23            svm       mean_norm  96.573738   0.987520  0.876446  0.928673                28.830
24            knn       mean_norm  96.492901   0.993382  0.867975  0.926454                 6.590
25             nn       mean_norm  74.998445   0.805085  0.023212  0.045123                 0.240
26  random_forest       mean_norm  95.512488   0.975906  0.844519  0.905471                 1.010
27            svm       soft_norm  97.128392   0.998347  0.888380  0.940159                 5.010
28            knn       soft_norm  92.200598   0.970376  0.714659  0.823114                 2.270
29             nn       soft_norm  79.429828   0.691433  0.342948  0.458488                 6.030
30  random_forest       soft_norm  96.454063   0.973980  0.883968  0.926794               100.000
31  random_forest     mean_scaled  95.879366   0.960939  0.873595  0.915188                 4.010
32            svm   median_scaled  89.524303   0.928979  0.637074  0.755822                19.750
33            knn   median_scaled  86.112551   0.942277  0.483955  0.639475                 6.770
34             nn   median_scaled  79.405120   0.739077  0.294836  0.421518                 2.380
35  random_forest   median_scaled  94.842989   0.940119  0.851604  0.893675                12.060
36  boosting_tree       tibi_mean  90.653562   0.881522  0.693540  0.776313                 7.122
37  boosting_tree     tibi_median  89.919309   0.858933  0.686813  0.763290                 1.481
38  boosting_tree      knn_scaled  23.992124   0.241226  0.925965  0.382743                99.700
39  boosting_tree  knn_normalized  24.636750   0.241137  0.913504  0.381555                99.660
40  boosting_tree   median_scaled  93.188931   0.866960  0.865125  0.866042                89.660
41  boosting_tree     median_norm  95.052337   0.904805  0.900309  0.902552                 1.880
42  boosting_tree     mean_scaled  95.833765   0.958967  0.873676  0.914337                89.380
43  boosting_tree       mean_norm  96.368536   0.986594  0.869115  0.924136                 1.520
44  boosting_tree       soft_norm  98.217694   0.990432  0.938879  0.963967               100.000
```

Spark

This binary classification problem has been successfully learned by machine learning models using big data tools like Spark.

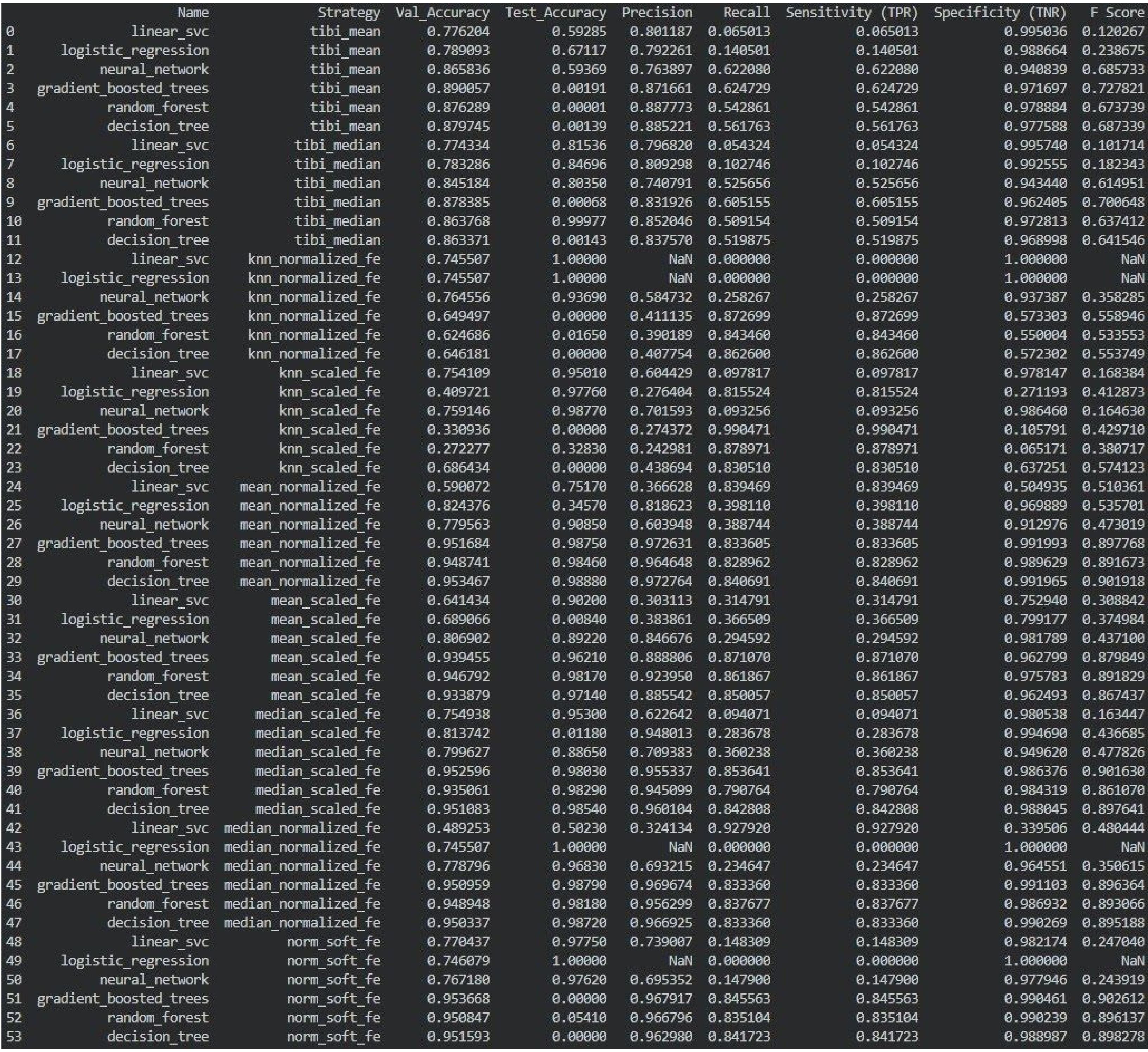
As it can be seen from the results tables and figures, the **best models** are the ones **based on trees**, with GBT and Decision trees obtaining the **highest** accuracies, followed up by Random forests. The largest validation accuracy is **95%** using **GBT** with **soft normalized** imputation. On the other hand, the **worst** performing models are **Linear Support Vector Classifier** and **Logistic Regression**, due to the high number of parameters involved. With the same imputation method, **soft normalized**, the Logistic regression model obtained an accuracy of **49%**.

As for the **imputation strategy**, the ones based on **mean and median** with **standardization and normalization**, along with **soft imputation** have obtained the **best** results, while those based on **KNN did not perform** as well as expected. A GBT model using soft normalized imputation obtained an accuracy o**f 95%**, while one using KNN imputation obtained only **33%** accuracy.

Ideally, the **confusion matrix** should have **sum on the principal diagonal equal to 1** (if normalized) and 0 on the secondary diagonal. The models used in the project achieved good results, obtaining **larger sums on the principal diagonal** for most of the experiments. Because our dataset is not balanced and there are more examples of firms which did not go into insolvency, most of the confusion matrices have a really large percentage of true negatives.

The **ROC curves** tell us how good the model is for discriminating positive and negative examples. If the **curve is above the diagonal of the plot**, then the model does a good job with this separation. In the figures obtained by our models we can see some examples of ROC curves which are high above the diagonal line, such as the ones obtained by Gradient Boosted Trees.

When it comes to differences compared to using plain Sklearn, one major one which is in favor of Spark approach is **scalability**, as the code would be able to run on separate machines and be faster to train. Another difference which comes as a drawback is the fact that PySpark API does not allow the same level of model configurations as the one from Sklearn. That’s why the best result achieved on Sklearn (98.22% validation accuracy on GBT with soft imputation) is higher than the best on Spark.



```
                      Name              Strategy  Val_Accuracy  Test_Accuracy  Precision    Recall  Sensitivity (TPR)  Specificity (TNR)   F Score
0               linear_svc             tibi_mean      0.776204        0.59285   0.801187  0.065013           0.065013           0.995036  0.120267
1      logistic_regression             tibi_mean      0.789093        0.67117   0.792261  0.140501           0.140501           0.988664  0.238675
2           neural_network             tibi_mean      0.865836        0.59369   0.763897  0.622080           0.622080           0.940839  0.685733
3   gradient_boosted_trees             tibi_mean      0.890057        0.00191   0.871661  0.624729           0.624729           0.971697  0.727821
4            random_forest             tibi_mean      0.876289        0.00001   0.887773  0.542861           0.542861           0.978884  0.673739
5            decision_tree             tibi_mean      0.879745        0.00139   0.885221  0.561763           0.561763           0.977588  0.687339
6               linear_svc           tibi_median      0.774334        0.81536   0.796820  0.054324           0.054324           0.995740  0.101714
7      logistic_regression           tibi_median      0.783286        0.84696   0.809298  0.102746           0.102746           0.992555  0.182343
8           neural_network           tibi_median      0.845184        0.80350   0.740791  0.525656           0.525656           0.943440  0.614951
9   gradient_boosted_trees           tibi_median      0.878385        0.00068   0.831926  0.605155           0.605155           0.962405  0.700648
10           random_forest           tibi_median      0.863768        0.99977   0.852046  0.509154           0.509154           0.972813  0.637412
11           decision_tree           tibi_median      0.863371        0.00143   0.837570  0.519875           0.519875           0.968998  0.641546
12              linear_svc     knn_normalized_fe      0.745507        1.00000        NaN  0.000000           0.000000           1.000000       NaN
13     logistic_regression     knn_normalized_fe      0.745507        1.00000        NaN  0.000000           0.000000           1.000000       NaN
14          neural_network     knn_normalized_fe      0.764556        0.93690   0.584732  0.258267           0.258267           0.937387  0.358285
15  gradient_boosted_trees     knn_normalized_fe      0.649497        0.00000   0.411135  0.872699           0.872699           0.573303  0.558946
16           random_forest     knn_normalized_fe      0.624686        0.01650   0.390189  0.843460           0.843460           0.550004  0.533553
17           decision_tree     knn_normalized_fe      0.646181        0.00000   0.407754  0.862600           0.862600           0.572302  0.553749
18              linear_svc         knn_scaled_fe      0.754109        0.95010   0.604429  0.097817           0.097817           0.978147  0.168384
19     logistic_regression         knn_scaled_fe      0.409721        0.97760   0.276404  0.815524           0.815524           0.271193  0.412873
20          neural_network         knn_scaled_fe      0.759146        0.98770   0.701593  0.093256           0.093256           0.986460  0.164630
21  gradient_boosted_trees         knn_scaled_fe      0.330936        0.00000   0.274372  0.990471           0.990471           0.105791  0.429710
22           random_forest         knn_scaled_fe      0.272277        0.32830   0.242981  0.878971           0.878971           0.065171  0.380717
23           decision_tree         knn_scaled_fe      0.686434        0.00000   0.438694  0.830510           0.830510           0.637251  0.574123
24              linear_svc    mean_normalized_fe      0.590072        0.75170   0.366628  0.839469           0.839469           0.504935  0.510361
25     logistic_regression    mean_normalized_fe      0.824376        0.34570   0.818623  0.398110           0.398110           0.969889  0.535701
26          neural_network    mean_normalized_fe      0.779563        0.90850   0.603948  0.388744           0.388744           0.912976  0.473019
27  gradient_boosted_trees    mean_normalized_fe      0.951684        0.98750   0.972631  0.833605           0.833605           0.991993  0.897768
28           random_forest    mean_normalized_fe      0.948741        0.98460   0.964648  0.828962           0.828962           0.989629  0.891673
29           decision_tree    mean_normalized_fe      0.953467        0.98880   0.972764  0.840691           0.840691           0.991965  0.901918
30              linear_svc        mean_scaled_fe      0.641434        0.90200   0.303113  0.314791           0.314791           0.752940  0.308842
31     logistic_regression        mean_scaled_fe      0.689066        0.00840   0.383861  0.366509           0.366509           0.799177  0.374984
32          neural_network        mean_scaled_fe      0.806902        0.89220   0.846676  0.294592           0.294592           0.981789  0.437100
33  gradient_boosted_trees        mean_scaled_fe      0.939455        0.96210   0.888806  0.871070           0.871070           0.962799  0.879849
34           random_forest        mean_scaled_fe      0.946792        0.98170   0.923950  0.861867           0.861867           0.975783  0.891829
35           decision_tree        mean_scaled_fe      0.933879        0.97140   0.885542  0.850057           0.850057           0.962493  0.867437
36              linear_svc      median_scaled_fe      0.754938        0.95300   0.622642  0.094071           0.094071           0.980538  0.163447
37     logistic_regression      median_scaled_fe      0.813742        0.01180   0.948013  0.283678           0.283678           0.994690  0.436685
38          neural_network      median_scaled_fe      0.799627        0.88650   0.709383  0.360238           0.360238           0.949620  0.477826
39  gradient_boosted_trees      median_scaled_fe      0.952596        0.98030   0.955337  0.853641           0.853641           0.986376  0.901630
40           random_forest      median_scaled_fe      0.935061        0.98290   0.945099  0.790764           0.790764           0.984319  0.861070
41           decision_tree      median_scaled_fe      0.951083        0.98540   0.960104  0.842808           0.842808           0.988045  0.897641
42              linear_svc  median_normalized_fe      0.489253        0.50230   0.324134  0.927920           0.927920           0.339506  0.480444
43     logistic_regression  median_normalized_fe      0.745507        1.00000        NaN  0.000000           0.000000           1.000000       NaN
44          neural_network  median_normalized_fe      0.778796        0.96830   0.693215  0.234647           0.234647           0.964551  0.350615
45  gradient_boosted_trees  median_normalized_fe      0.950959        0.98790   0.969674  0.833360           0.833360           0.991103  0.896364
46           random_forest  median_normalized_fe      0.948948        0.98180   0.956299  0.837677           0.837677           0.986932  0.893066
47           decision_tree  median_normalized_fe      0.950337        0.98720   0.966925  0.833360           0.833360           0.990269  0.895188
48              linear_svc          norm_soft_fe      0.770437        0.97750   0.739007  0.148309           0.148309           0.982174  0.247040
49     logistic_regression          norm_soft_fe      0.746079        1.00000        NaN  0.000000           0.000000           1.000000       NaN
50          neural_network          norm_soft_fe      0.767180        0.97620   0.695352  0.147900           0.147900           0.977946  0.243919
51  gradient_boosted_trees          norm_soft_fe      0.953668        0.00000   0.967917  0.845563           0.845563           0.990461  0.902612
52           random_forest          norm_soft_fe      0.950847        0.05410   0.966796  0.835104           0.835104           0.990239  0.896137
53           decision_tree          norm_soft_fe      0.951593        0.00000   0.962980  0.841723           0.841723           0.988987  0.898278
```

Conclusions

Modelling on microeconomic data has proven an interesting and insightful task, given the demand for feature engineering and the nature of the raw features. Even though the number of predictors was not high compared to other datasets, the variance in their nature, NaN count and type definitely put us in a pinch a couple of times.

Another problem was to find a way to correctly assess the performance of our models. Training on both labels and predicting on a set with an unique label would've given partially flawed results, as we had no idea how our models behaved on the other label. We bypassed this by applying the 60-30-10 train-validation-test split, where the validation set acted as the actual test set with both labels and the test set was used to test the prediction power on an insolvency-free.

From a computational time perspective, the SKLearn models took way longer to train and use. The Random Forest model took about 4 hours to train on a machine with an 8-core CPU. The SVM took about 6 hours. However, the XGBoost algorithm was really fast, with about 30 minutes per imputation method. The same thing can be said about the neural network, but that happened because it only had one layer with 30 neurons, since it was the first model we considered and we wanted to have a fast prototype in order to have a rough idea about the dataset and the results we should expect. The RAM usage wasn't high for the SKLearn models, but the CPU was under some stress.

For the PySpark module, training all the models for one imputation method took about 20-30 minutes on a 6 core (12 threads) CPU, so it offered a considerable improvement in time over Sklearn. However, there are some drawbacks when it comes to memory consumption, because training all the algorithms on all the imputation methods in one run caused the RAM to fill up, even dumping memory to disk and blocking the computer. So, in order to avoid this, for each imputation method we had to manually start the training process for all algorithms.

It comes as no surprise that the Sklearn approach yielded better results than the Spark MLlib. The main reason for this would be that Sklearn, as a machine learning-specialized library, has had its algorithms especially designed for prediction performance.

On the other hand, Spark was built in order to fit the role of a scalable big data engineering library. Thus, it's MLlib component has only later been added and has not experienced much development. This is why it lacks Sklearn's versatility when it comes to hyperparameter tuning and cross-validation methods.

However, Spark does bear one important quality and that is scalability. As we have seen, the KNN imputation method has been too computationally exhausting for both our machines and Google Colab. In this scenario, with enough commodity hardware to run on - an AWS, Azure or GCP hardware cluster is not expensive even for a small company - Spark, in combination with KNN imputation, would've offered much better results than our KNN approach.

An industry standard is to combine the two approaches that we have presented in this report. In other words, to distribute Sklearn models on a cloud hardware cluster using Spark. This way we would get the best out of both worlds: Sklearn's predictive performance and Sparks computational optimizations.

Another expected result on the validation set is the fact that the best performing model is the gradient boosted trees algorithm. However, as the Spark results figure has shown, the random forest and gradient boosted trees perform extremely poorly, up to the point in which the boosted trees have 0% accuracy. This is an interesting result that could be used to our advantage. Given that the boosted trees algorithm always marks a company that has not declared insolvency as one that has, all we must do is to invert the labels of the algorithm's output. This way we achieve a 100% score on the boosted trees classifier.

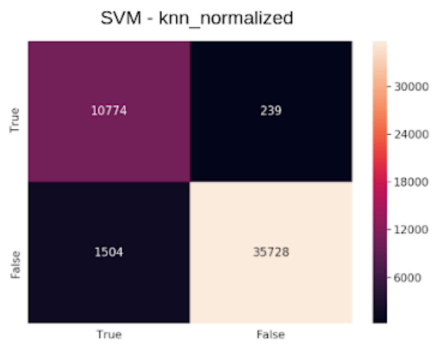
Further research

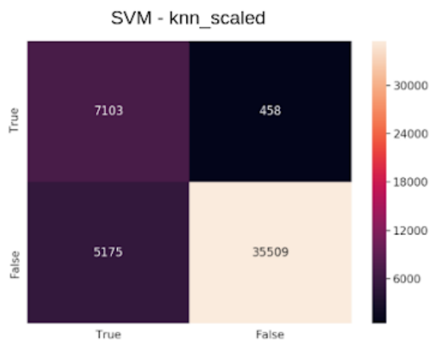
As we have discussed in the previous section, a worthwhile research direction regarding the problem of insolvency analysis would be a hybrid approach using Spark to leverage computational strain on a commodity cluster and Sklearn to assure a high performance in classification.

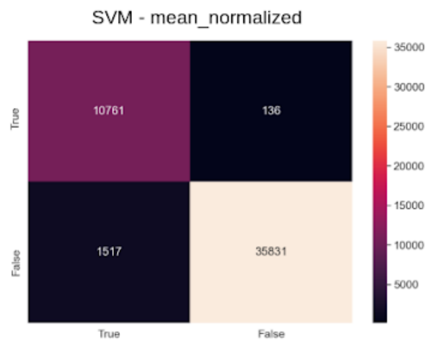
In this scenario, the Sklearn models would be distributed across the Spark cluster either via Spark streams or as .py files that would be run. After this, the results would be aggregated to the Spark master node and a few custom metrics would need to be designed and computed in order to obtain unflawed, correct feedback from the workers.

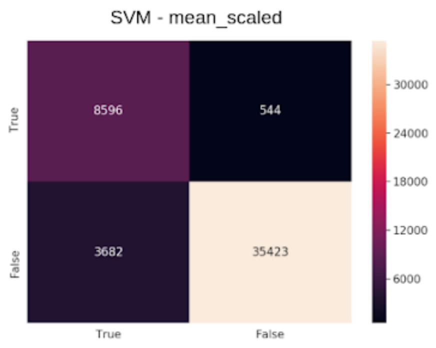
The big data technologies play a bigger role than just distributing datasets across a cluster. HDFS would be a suitable candidate here for data storage, as its size grows larger with each imputed dataset. Moreover, Spark and HDFS are optimized to work well together, as part of the same technology stack, resulting in faster computations, retrieval and storage.

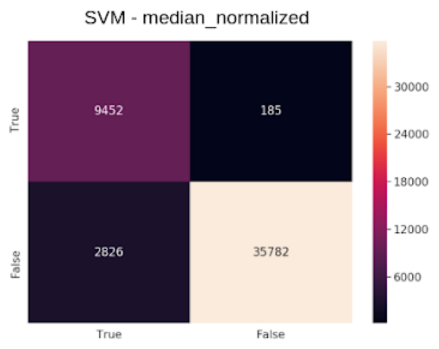
Another possible improvement would be receiving feedback from an industry expert, such a financial analyst or an economist, on how to best treat our raw KPIs in the preprocessing stage, what new features to compute based on the raw ones and how to interpret them.

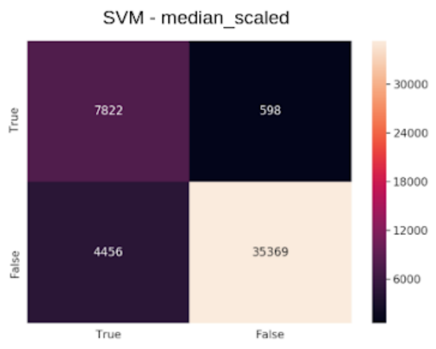

SVM - knn_normalized
True
False
10774
239
1504
35728
True
False
30000
24000
18000
12000
6000

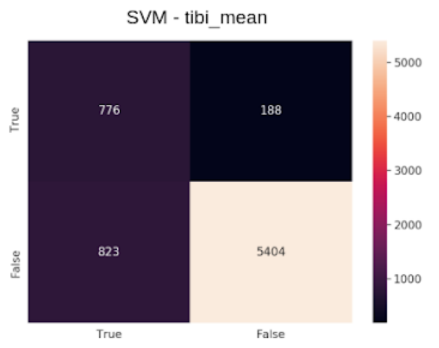

SVM - knn_scaled
True
False
7103
458
5175
35509
True
False
30000
24000
18000
12000
6000

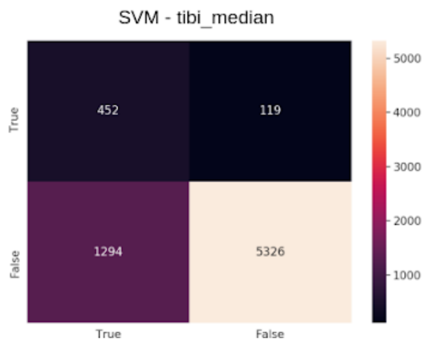

SVM - mean_normalized
True
False
10761
136
1517
35831
True
False
35000
30000
25000
20000
15000
10000
5000

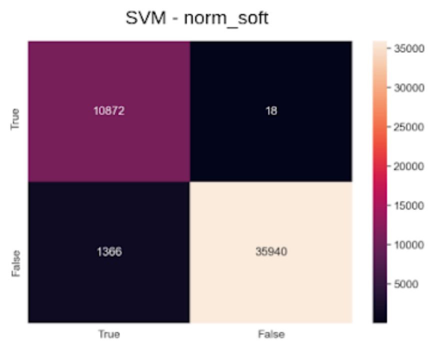

SVM - mean_scaled
True
False
8596
544
3682
35423
True
False
30000
24000
18000
12000
6000


SVM - median_normalized
True
False
9452
185
2826
35782
True
False
30000
24000
18000
12000
6000

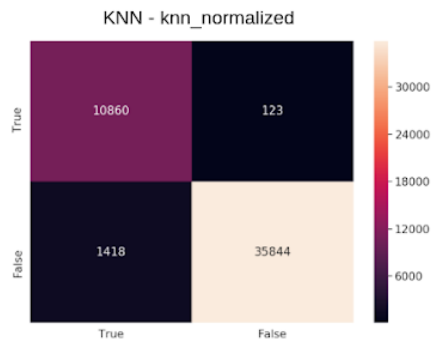

SVM - median_scaled
True
False
7822
598
4456
35369
True
False
30000
24000
18000
12000
6000

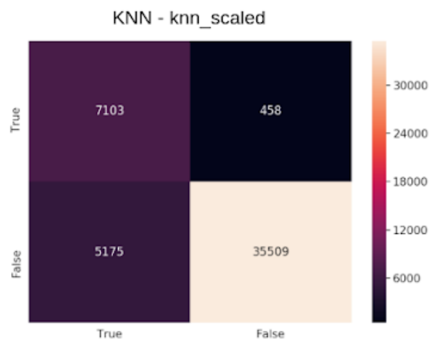

SVM - tibi_mean
True
False
776
188
823
5404
True
False
5000
4000
3000
2000
1000

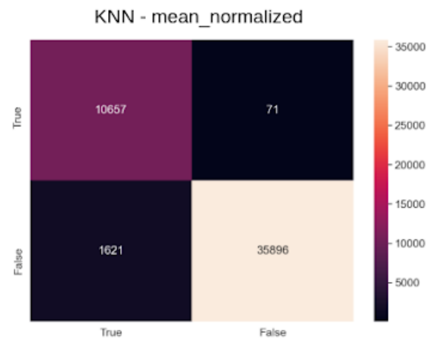

SVM - tibi_median
True
False
452
119
1294
5326
True
False
5000
4000
3000
2000
1000

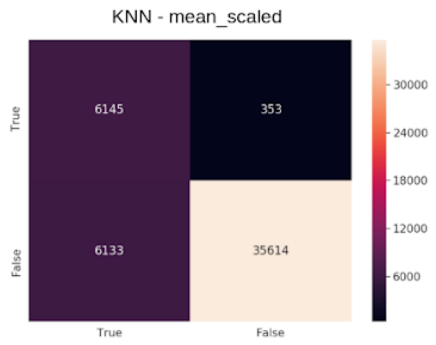

SVM - norm_soft
True
False
10872
18
1366
35940
True
False
35000
30000
25000
20000
15000
10000
5000

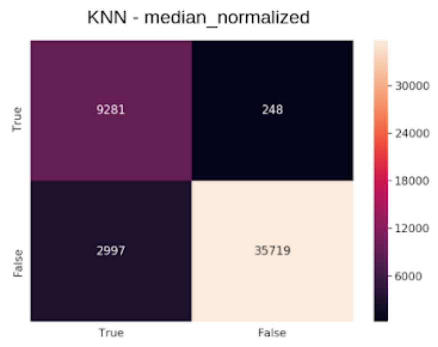
1

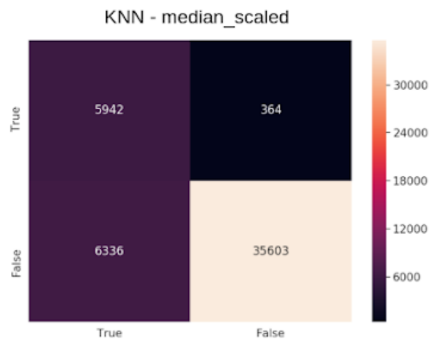

KNN - knn_normalized
True
False
10860
123
1418
35844
True
False
30000
24000
18000
12000
6000

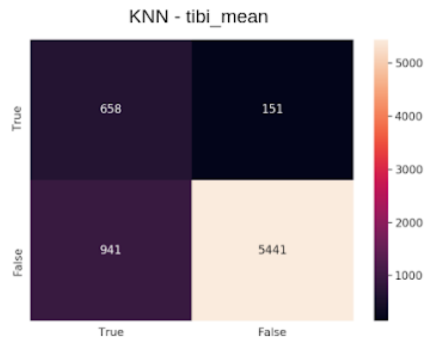

KNN - knn_scaled
True
False
7103
458
5175
35509
True
False
30000
24000
18000
12000
6000

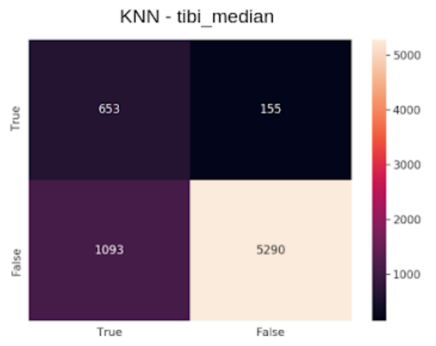

KNN - mean_normalized
True
False
10657
71
1621
35896
True
False
35000
30000
25000
20000
15000
10000
5000

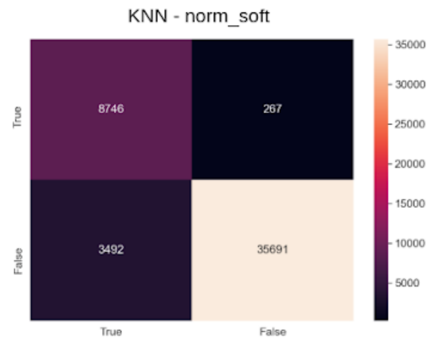

KNN - mean_scaled
True
False
6145
353
6133
35614
True
False
30000
24000
18000
12000
6000

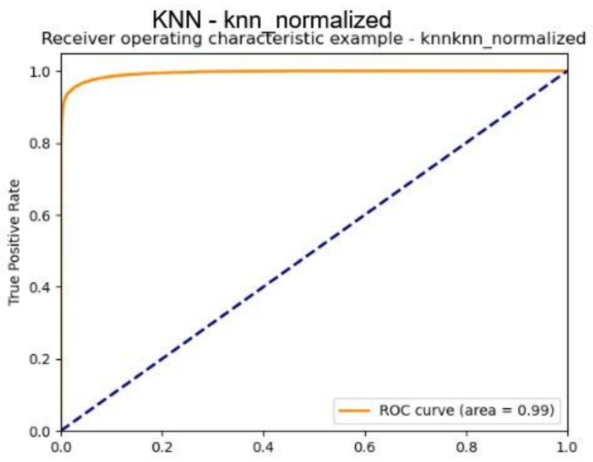

KNN - median_normalized
True
False
9281
248
2997
35719
True
False
30000
24000
18000
12000
6000

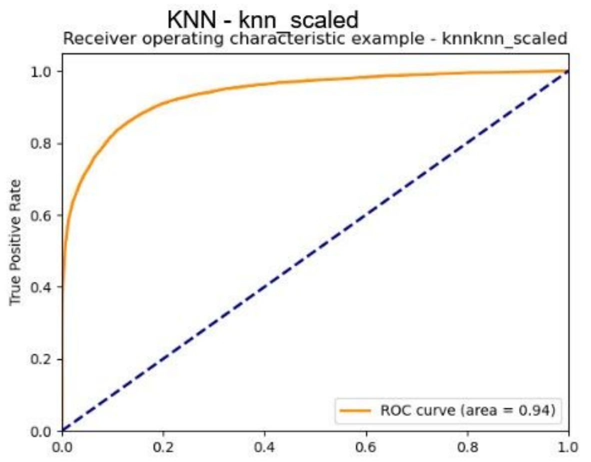

KNN - median_scaled
True
False
5942
364
6336
35603
True
False
30000
24000
18000
12000
6000

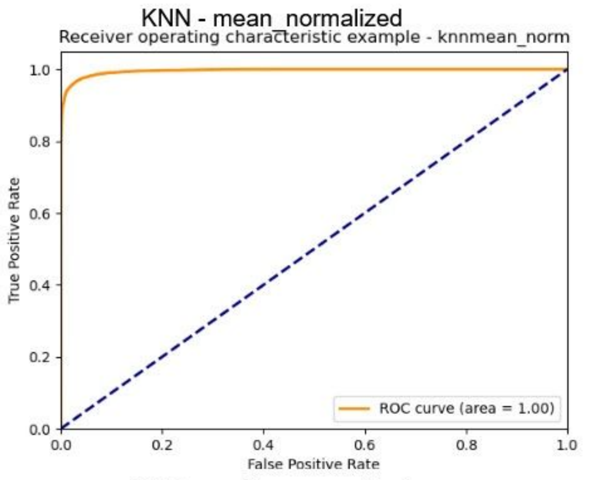

KNN - tibi_mean
True
False
658
151
941
5441
True
False
5000
4000
3000
2000
1000

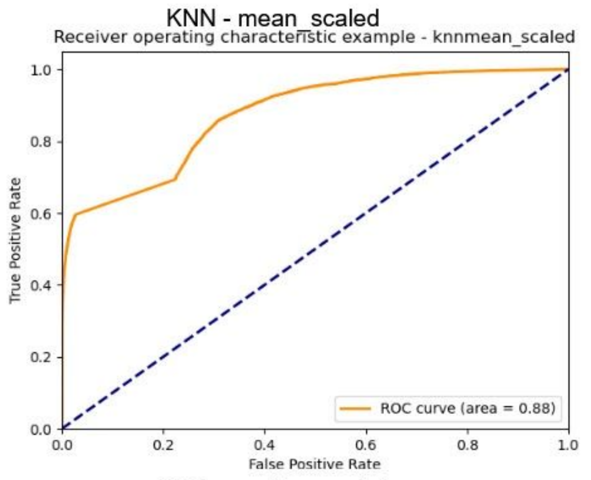

KNN - tibi_median
True
False
653
155
1093
5290
True
False
5000
4000
3000
2000
1000

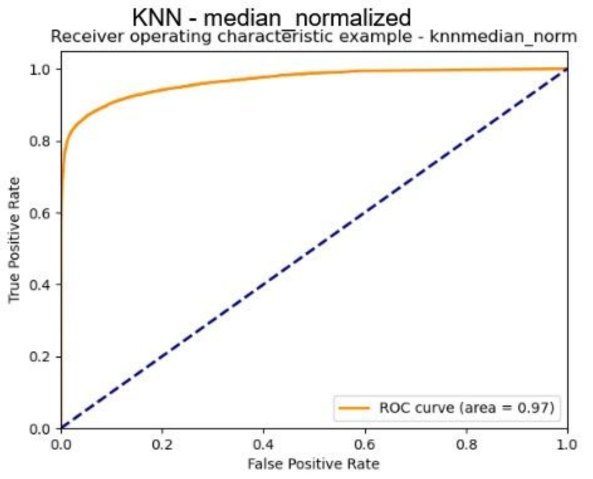

KNN - norm_soft
True
False
8746
267
3492
35691
True
False
35000
30000
25000
20000
15000
10000
5000

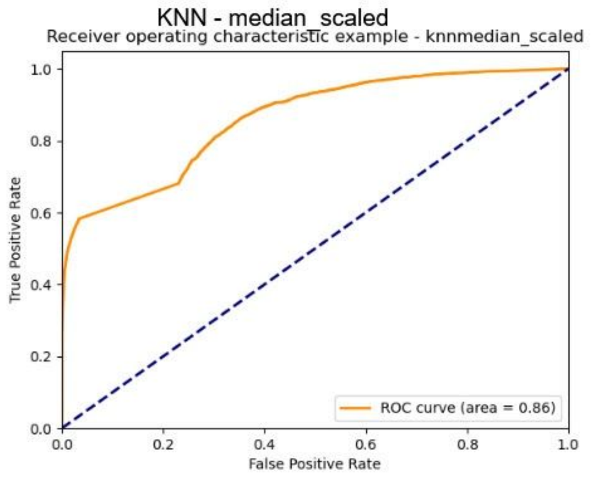

KNN - knn_normalized
Receiver operating characteristic example - knnknn_normalized
True Positive Rate
ROC curve (area = 0.99)

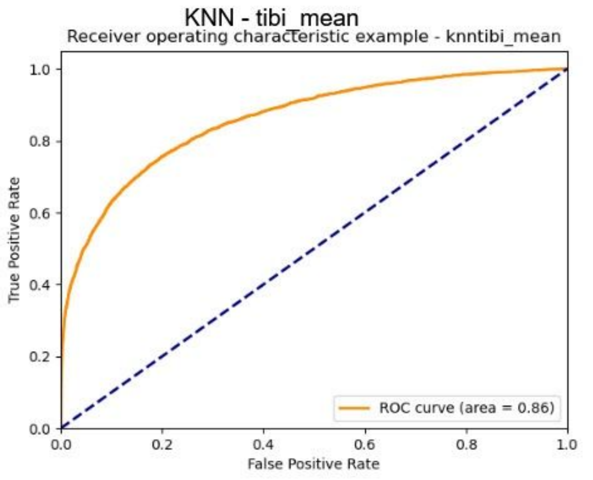

KNN - knn_scaled
Receiver operating characteristic example - knnknn_scaled
True Positive Rate
ROC curve (area = 0.94)


KNN - mean_normalized
Receiver operating characteristic example - knnmean_norm
True Positive Rate
False Positive Rate
ROC curve (area = 1.00)

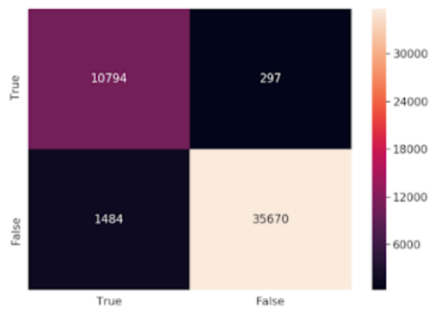

KNN - mean_scaled
Receiver operating characteristic example - knnmean_scaled
True Positive Rate
False Positive Rate
ROC curve (area = 0.88)


KNN - median_normalized
Receiver operating characteristic example - knnmedian_norm
True Positive Rate
False Positive Rate
ROC curve (area = 0.97)

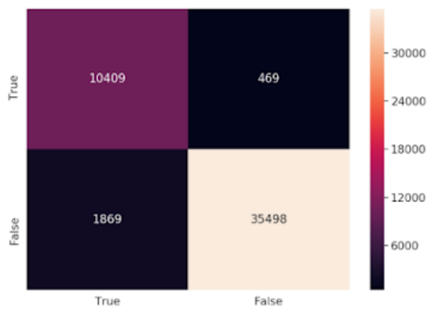

KNN - median_scaled
Receiver operating characteristic example - knnmedian_scaled
True Positive Rate
False Positive Rate
ROC curve (area = 0.86)


KNN - tibi_mean
Receiver operating characteristic example - knntibi_mean
True Positive Rate
False Positive Rate
ROC curve (area = 0.86)

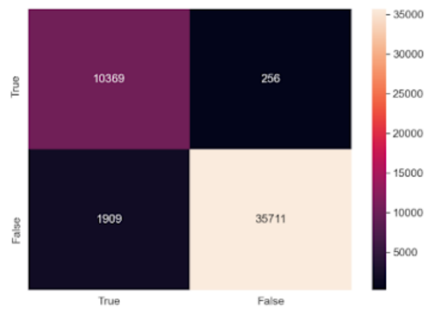
random_forest - knn_normalized



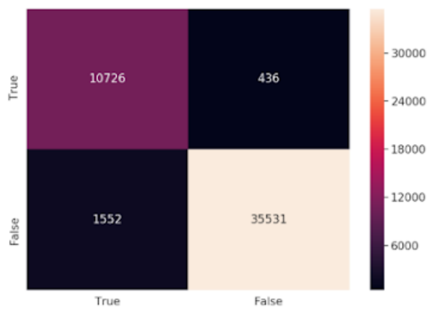
random_forest - knn_scaled



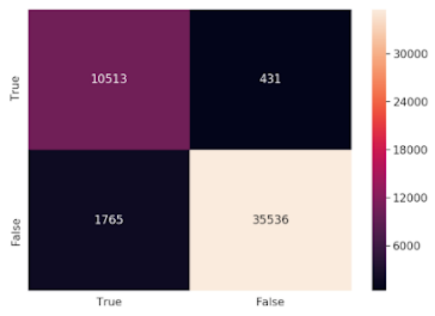
random_forest - mean_normalized



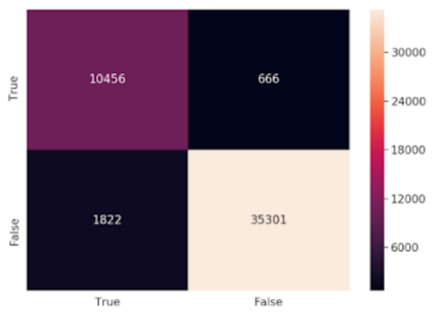
random_forest - mean_scaled



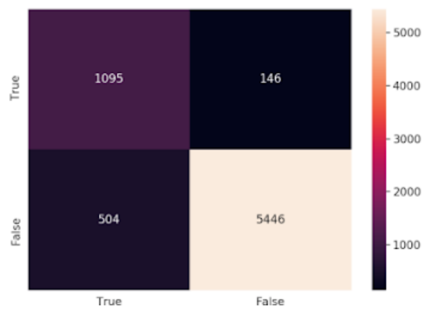
random_forest - median_normalized



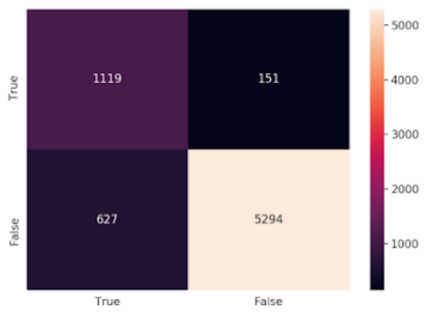
random_forest - median_scaled



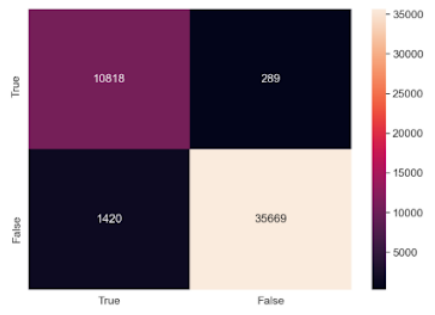
random_forest - tibi_mean

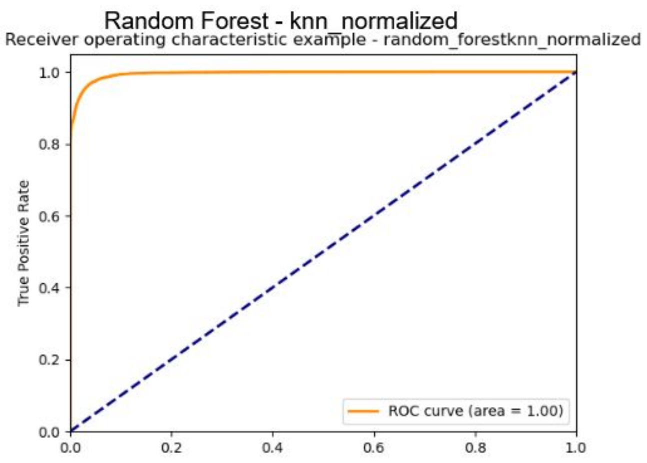


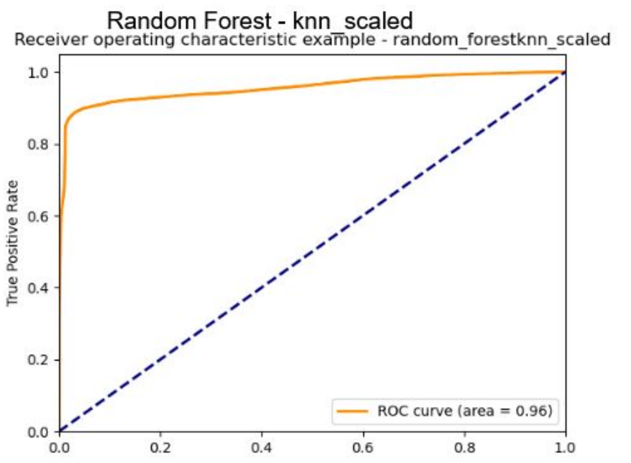
random_forest - tibi_median

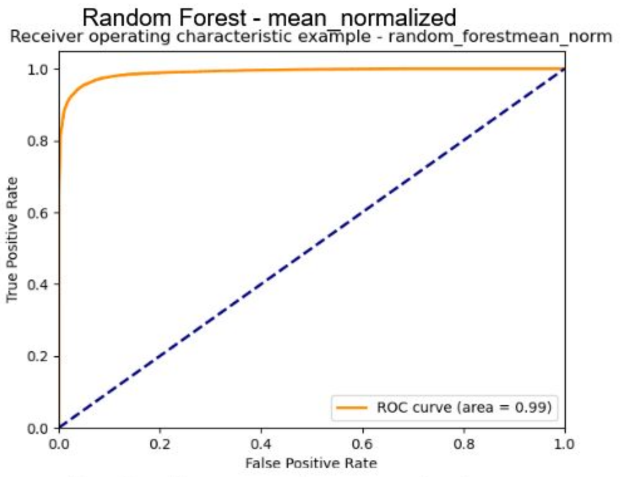


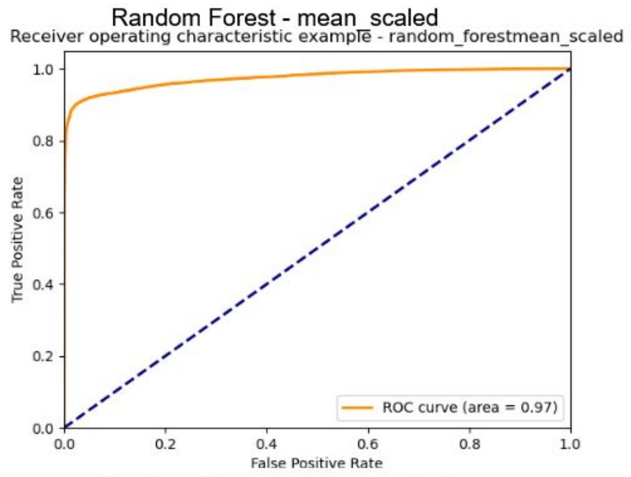
random_forest - norm_soft

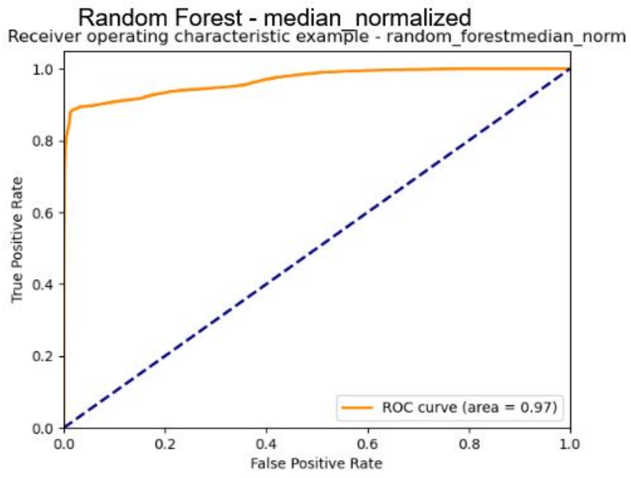


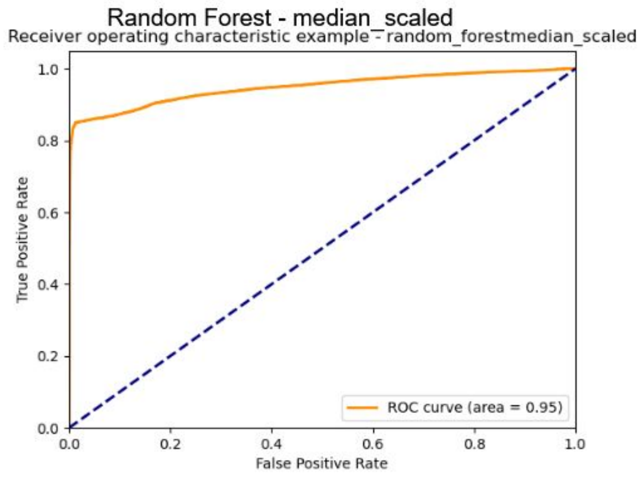

Random Forest - knn_normalized
Receiver operating characteristic example - random_forestknn_normalized
True Positive Rate
ROC curve (area = 1.00)

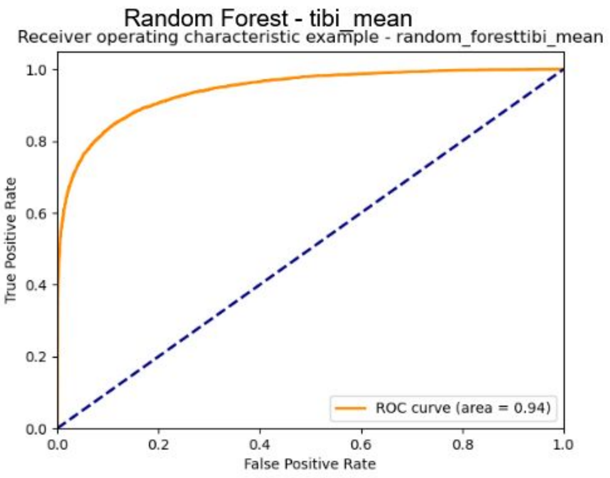

Random Forest - knn_scaled
Receiver operating characteristic example - random_forestknn_scaled
True Positive Rate
ROC curve (area = 0.96)

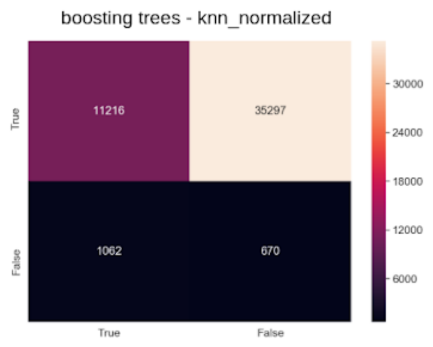

Random Forest - mean_normalized
Receiver operating characteristic example - random_forestmean_norm
True Positive Rate
False Positive Rate
ROC curve (area = 0.99)

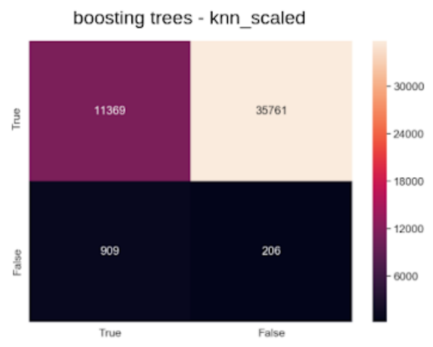

Random Forest - mean_scaled
Receiver operating characteristic example - random_forestmean_scaled
True Positive Rate
False Positive Rate
ROC curve (area = 0.97)

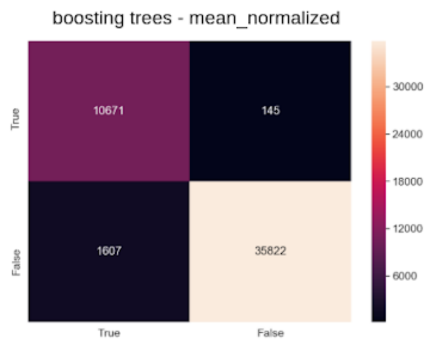

Random Forest - median_normalized
Receiver operating characteristic example - random_forestmedian_norm
True Positive Rate
False Positive Rate
ROC curve (area = 0.97)

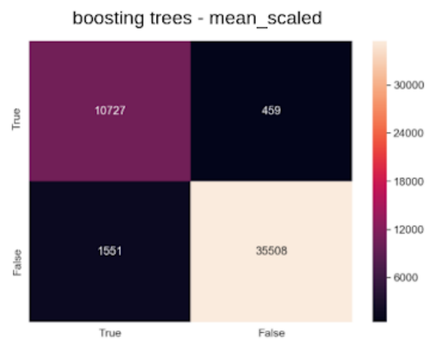

Random Forest - median_scaled
Receiver operating characteristic example - random_forestmedian_scaled
True Positive Rate
False Positive Rate
ROC curve (area = 0.95)

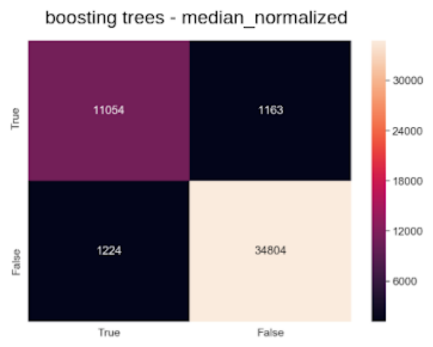

Random Forest - tibi_mean
Receiver operating characteristic example - random_foresttibi_mean
True Positive Rate
False Positive Rate
ROC curve (area = 0.94)

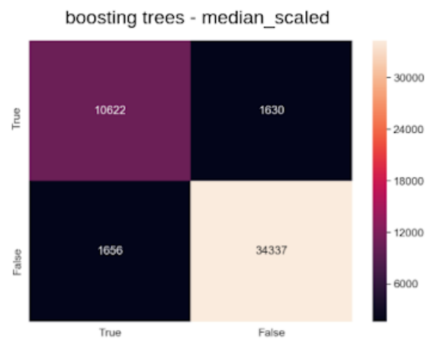

boosting trees - knn_normalized
True
False
11216
35297
1062
670
True
False
30000
24000
18000
12000
6000

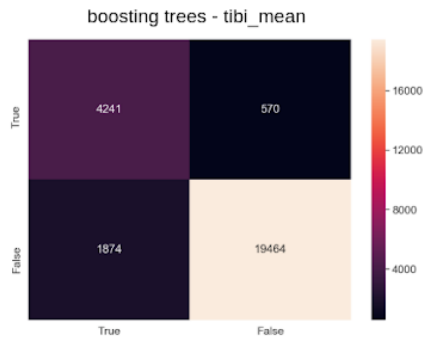

boosting trees - knn_scaled
True
False
11369
35761
909
206
True
False
30000
24000
18000
12000
6000

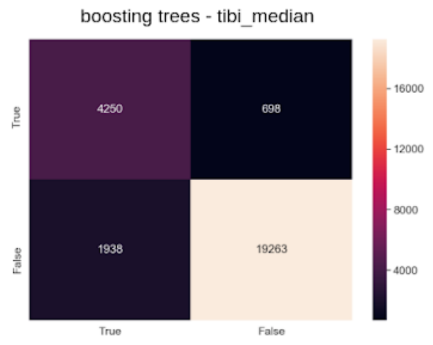

boosting trees - mean_normalized
True
False
10671
145
1607
35822
True
False
30000
24000
18000
12000
6000

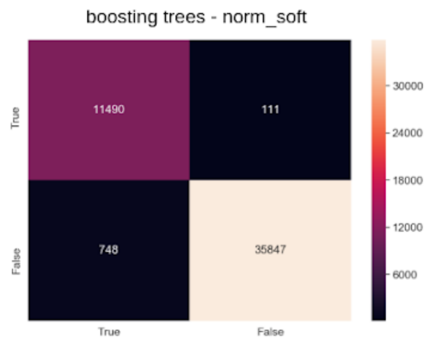

boosting trees - mean_scaled
True
False
10727
459
1551
35508
True
False
30000
24000
18000
12000
6000

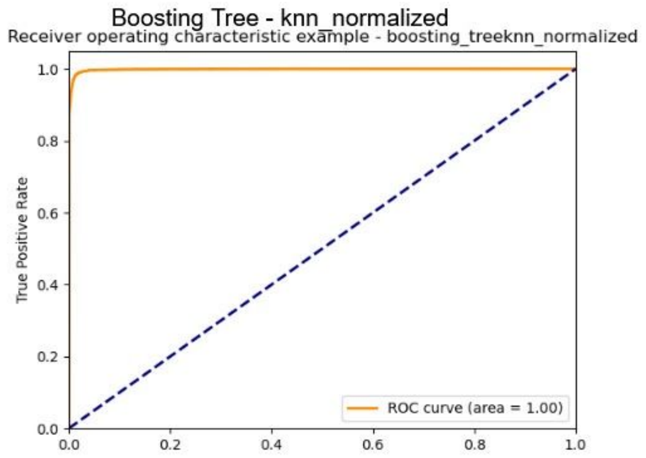

boosting trees - median_normalized
True
False
11054
1163
1224
34804
True
False
30000
24000
18000
12000
6000

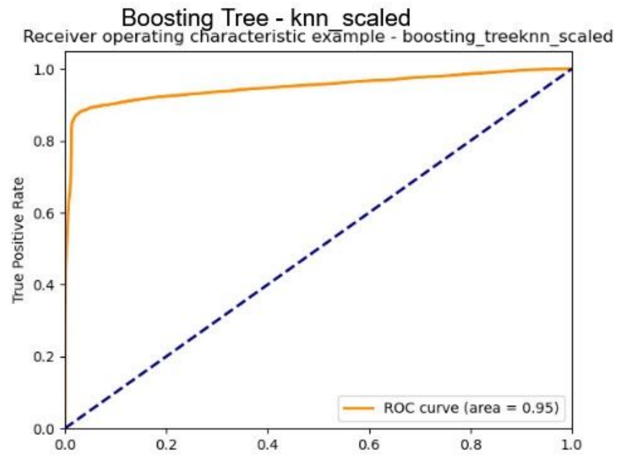

boosting trees - median_scaled
True
False
10622
1630
1656
34337
True
False
30000
24000
18000
12000
6000

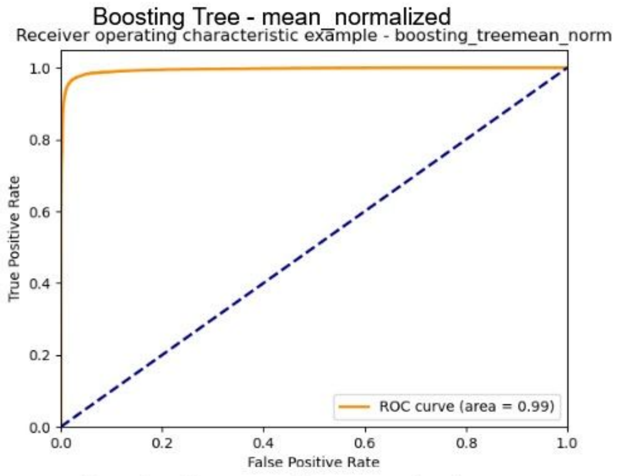

boosting trees - tibi_mean
True
False
4241
570
1874
19464
True
False
16000
12000
8000
4000

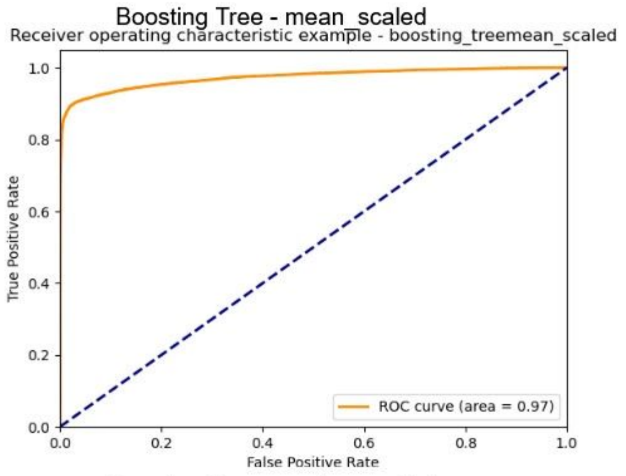

boosting trees - tibi_median
True
False
4250
698
1938
19263
True
False
16000
12000
8000
4000

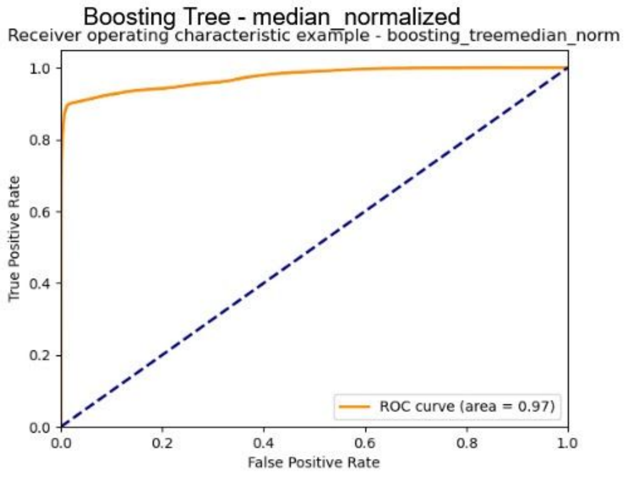

boosting trees - norm_soft
True
False
11490
111
748
35847
True
False
30000
24000
18000
12000
6000

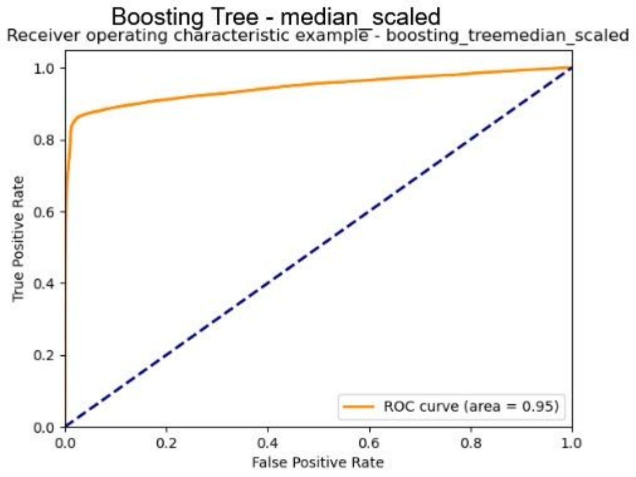

Boosting Tree - knn_normalized
Receiver operating characteristic example - boosting_treeknn_normalized
True Positive Rate
ROC curve (area = 1.00)

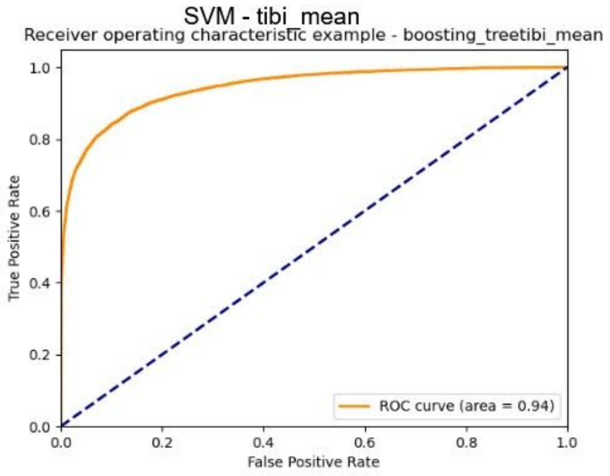

Boosting Tree - knn_scaled
Receiver operating characteristic example - boosting_treeknn_scaled
True Positive Rate
ROC curve (area = 0.95)

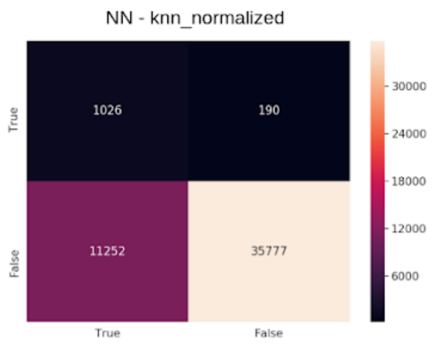

Boosting Tree - mean_normalized
Receiver operating characteristic example - boosting_treemean_norm
True Positive Rate
False Positive Rate
ROC curve (area = 0.99)

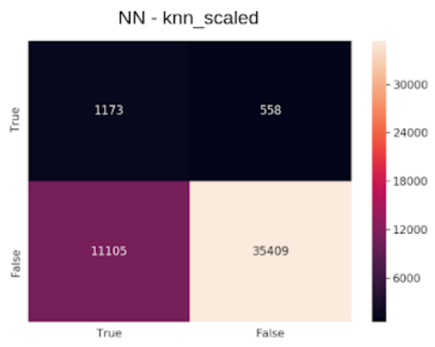

Boosting Tree - mean_scaled
Receiver operating characteristic example - boosting_treemean_scaled
True Positive Rate
False Positive Rate
ROC curve (area = 0.97)

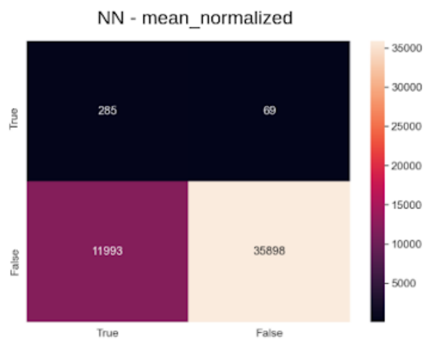

Boosting Tree - median_normalized
Receiver operating characteristic example - boosting_treemedian_norm
True Positive Rate
False Positive Rate
ROC curve (area = 0.97)

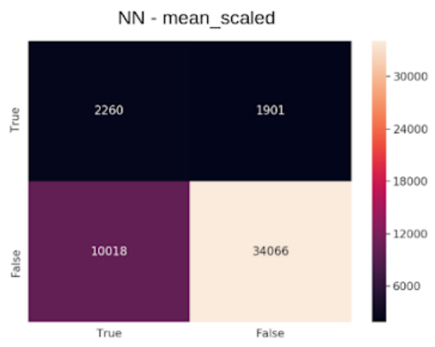

Boosting Tree - median_scaled
Receiver operating characteristic example - boosting_treemedian_scaled
True Positive Rate
False Positive Rate
ROC curve (area = 0.95)

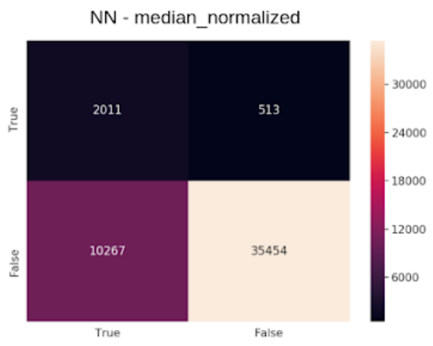

SVM - tibi_mean
Receiver operating characteristic example - boosting_treetibi_mean
True Positive Rate
False Positive Rate
ROC curve (area = 0.94)

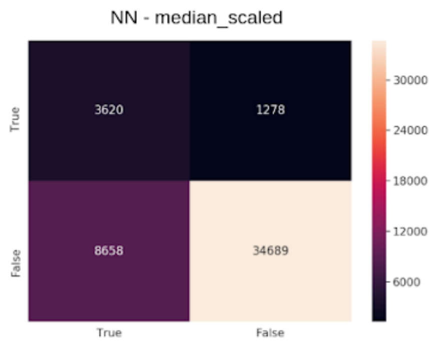

NN - knn_normalized
True
False
1026
190
11252
35777
True
False
30000
24000
18000
12000
6000

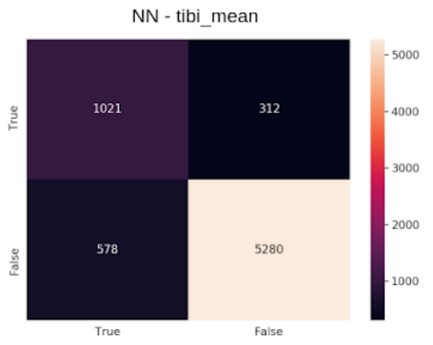

NN - knn_scaled
True
False
1173
558
11105
35409
True
False
30000
24000
18000
12000
6000

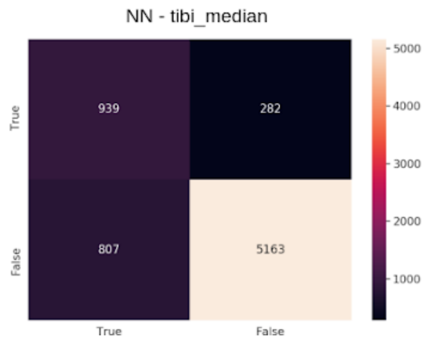

NN - mean_normalized
True
False
285
69
11993
35898
True
False
35000
30000
25000
20000
15000
10000
5000

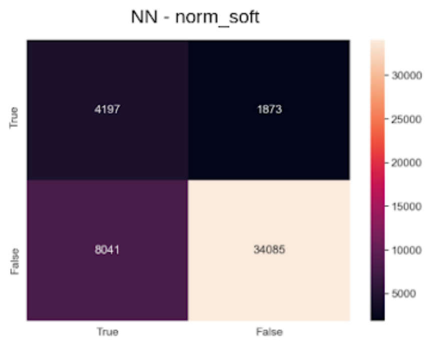

NN - mean_scaled
True
False
2260
1901
10018
34066
True
False
30000
24000
18000
12000
6000

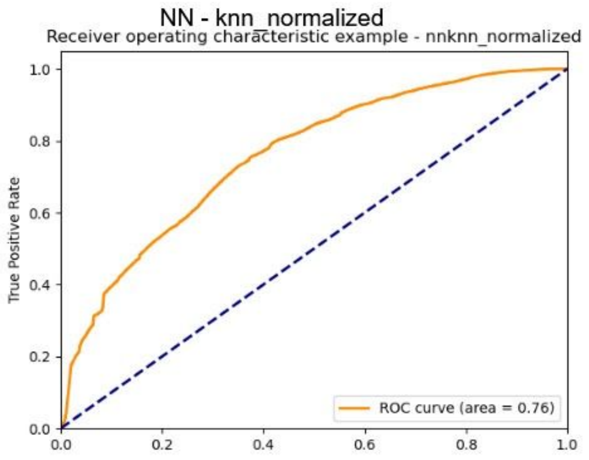

NN - median_normalized
True
False
2011
513
10267
35454
True
False
30000
24000
18000
12000
6000

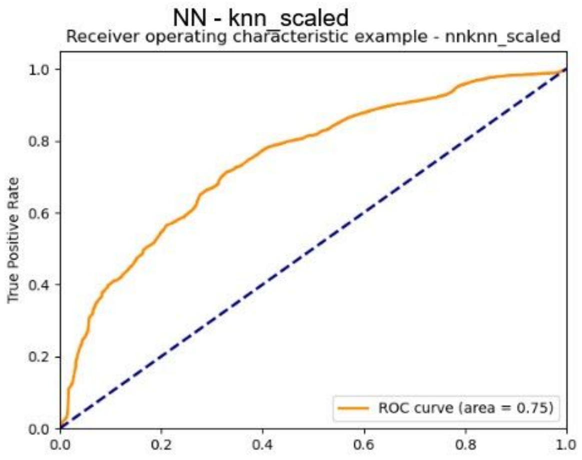

NN - median_scaled
True
False
3620
1278
8658
34689
True
False
30000
24000
18000
12000
6000

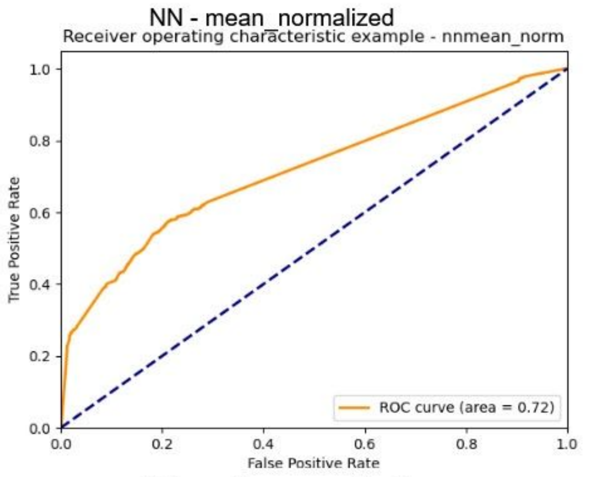

NN - tibi_mean
True
False
1021
312
578
5280
True
False
5000
4000
3000
2000
1000

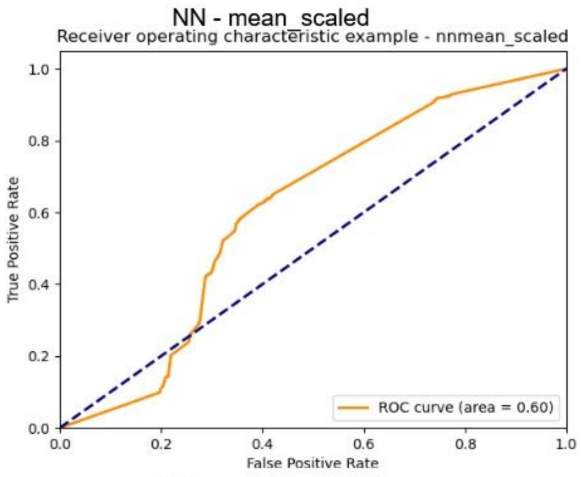

NN - tibi_median
True
False
939
282
807
5163
True
False
5000
4000
3000
2000
1000

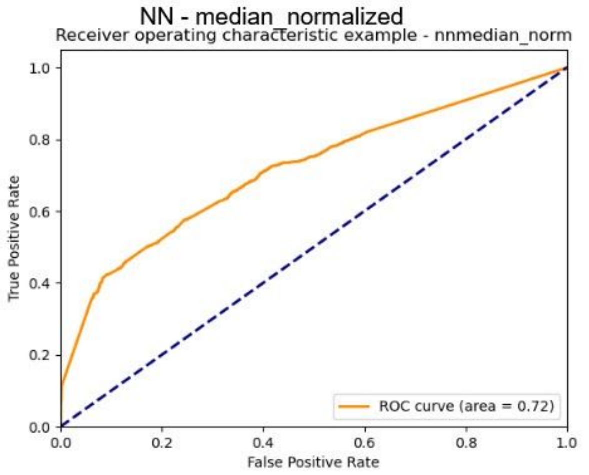

NN - norm_soft
True
False
4197
1873
8041
34085
True
False
30000
25000
20000
15000
10000
5000

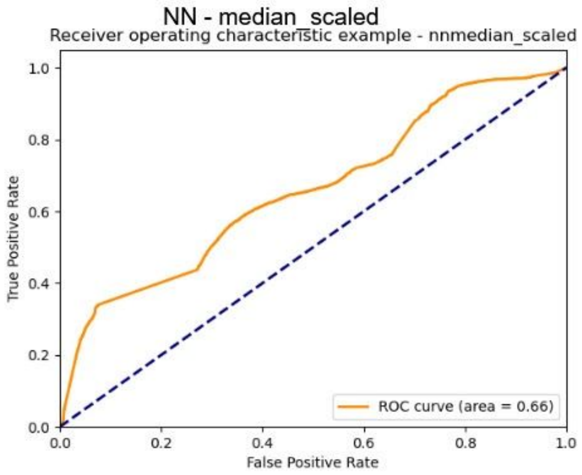

NN - knn_normalized
Receiver operating characteristic example - nnknn_normalized
True Positive Rate
ROC curve (area = 0.76)

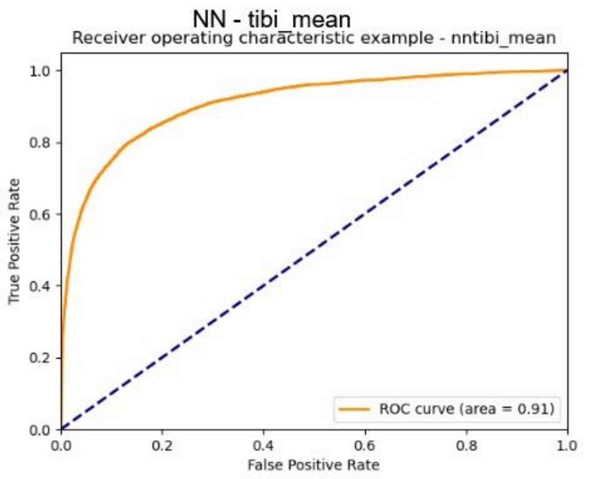

NN - knn_scaled
Receiver operating characteristic example - nnknn_scaled
True Positive Rate
ROC curve (area = 0.75)

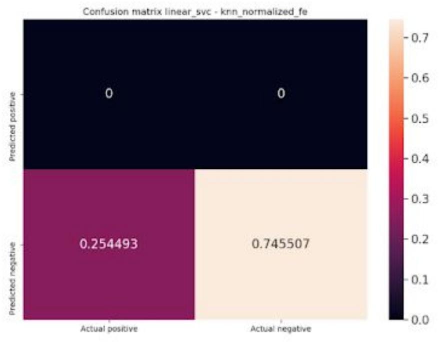

NN - mean_normalized
Receiver operating characteristic example - nnmean_norm
True Positive Rate
False Positive Rate
ROC curve (area = 0.72)

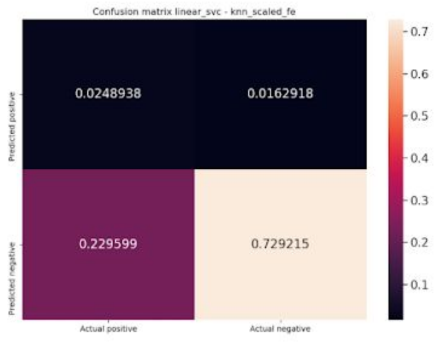

NN - mean_scaled
Receiver operating characteristic example - nnmean_scaled
True Positive Rate
False Positive Rate
ROC curve (area = 0.60)

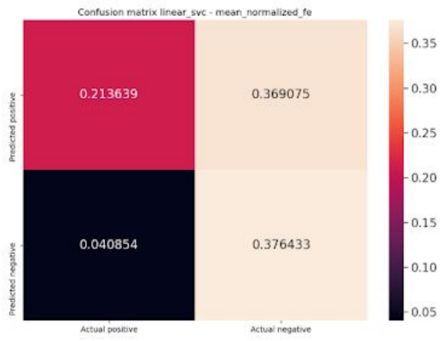

NN - median_normalized
Receiver operating characteristic example - nnmedian_norm
True Positive Rate
False Positive Rate
ROC curve (area = 0.72)

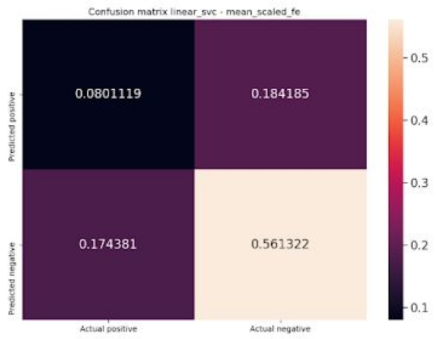

NN - median_scaled
Receiver operating characteristic example - nnmedian_scaled
True Positive Rate
False Positive Rate
ROC curve (area = 0.66)

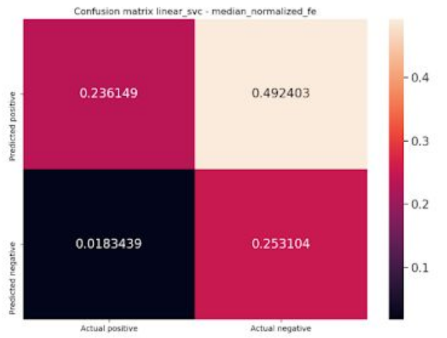

NN - tibi_mean
Receiver operating characteristic example - nntibi_mean
True Positive Rate
False Positive Rate
ROC curve (area = 0.91)

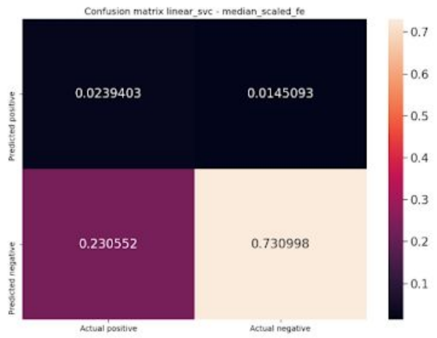

Confusion matrix linear_svc - knn_normalized_fe
Predicted positive
Predicted negative
0
0
0.254493
0.745507
Actual positive
Actual negative
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0

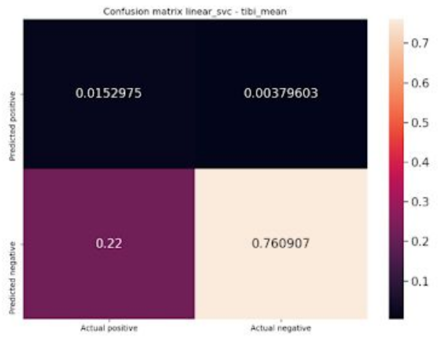

Confusion matrix linear_svc - knn_scaled_fe
Predicted positive
Predicted negative
0.0248938
0.0162918
0.229599
0.729215
Actual positive
Actual negative
0.7
0.6
0.5
0.4
0.3
0.2
0.1

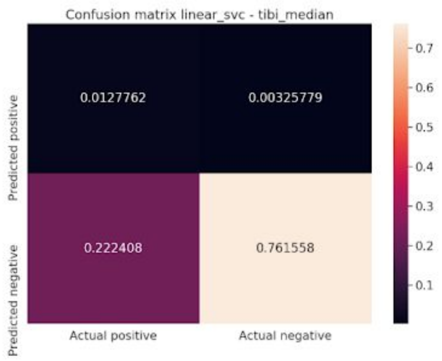

Confusion matrix linear_svc - mean_normalized_fe
Predicted positive
Predicted negative
0.213639
0.369075
0.040854
0.376433
Actual positive
Actual negative
0.35
0.30
0.25
0.20
0.15
0.10
0.05

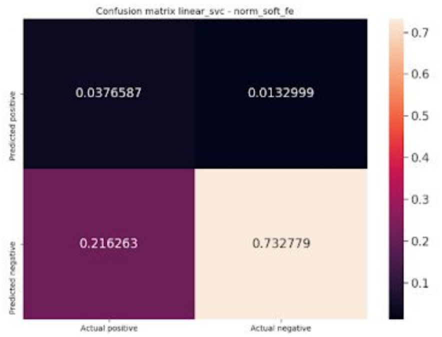

Confusion matrix linear_svc - mean_scaled_fe
Predicted positive
Predicted negative
0.0801119
0.184185
0.174381
0.561322
Actual positive
Actual negative
0.5
0.4
0.3
0.2
0.1

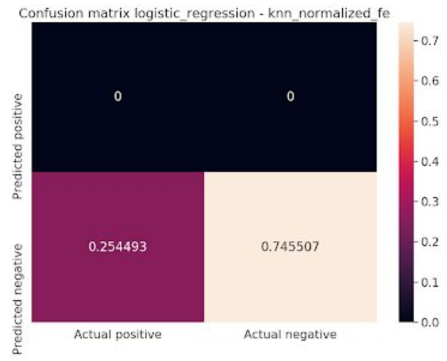

Confusion matrix linear_svc - median_normalized_fe
Predicted positive
Predicted negative
0.236149
0.492403
0.0183439
0.253104
Actual positive
Actual negative
0.4
0.3
0.2
0.1

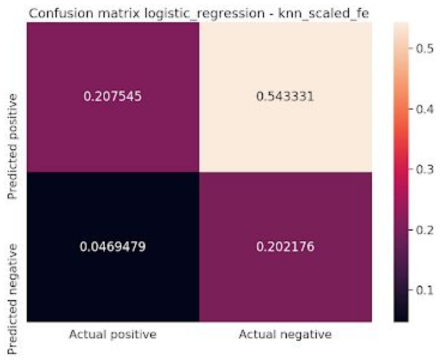

Confusion matrix linear_svc - median_scaled_fe
Predicted positive
Predicted negative
0.0239403
0.0145093
0.230552
0.730998
Actual positive
Actual negative
0.7
0.6
0.5
0.4
0.3
0.2
0.1

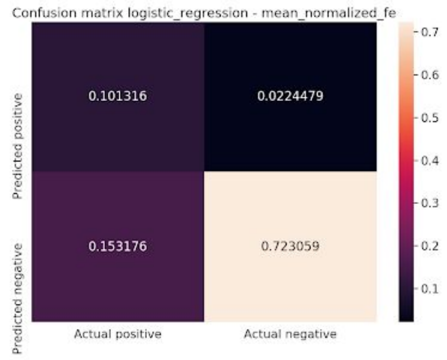

Confusion matrix linear_svc - tibi_mean
Predicted positive
Predicted negative
0.0152975
0.00379603
0.22
0.760907
Actual positive
Actual negative
0.7
0.6
0.5
0.4
0.3
0.2
0.1

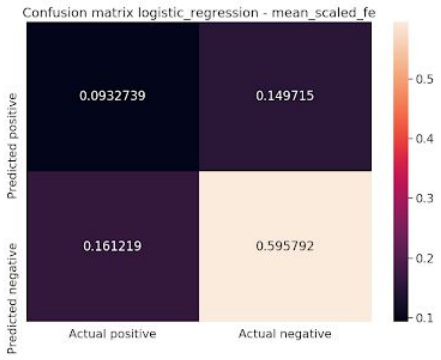

Confusion matrix linear_svc - tibi_median
Predicted positive
Predicted negative
0.0127762
0.00325779
0.222408
0.761558
Actual positive
Actual negative
0.7
0.6
0.5
0.4
0.3
0.2
0.1

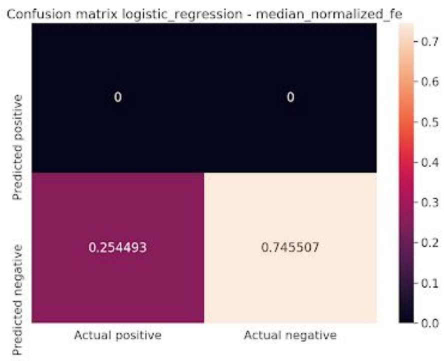

Confusion matrix linear_svc - norm_soft_fe
Predicted positive
Predicted negative
0.0376587
0.0132999
0.216263
0.732779
Actual positive
Actual negative
0.7
0.6
0.5
0.4
0.3
0.2
0.1

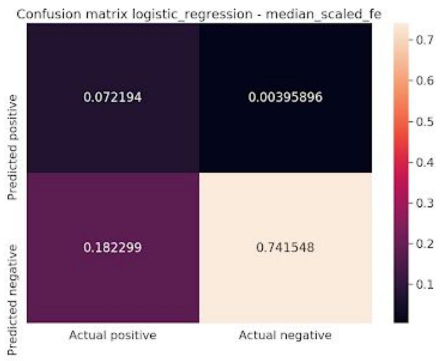

Confusion matrix logistic_regression - knn_normalized_fe
Predicted positive
Predicted negative
0
0
0.254493
0.745507
Actual positive
Actual negative
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0

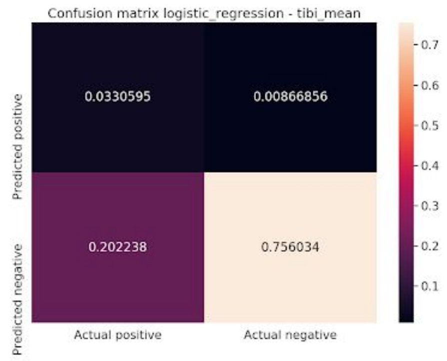

Confusion matrix logistic_regression - knn_scaled_fe
Predicted positive
Predicted negative
0.207545
0.543331
0.0469479
0.202176
Actual positive
Actual negative
0.5
0.4
0.3
0.2
0.1

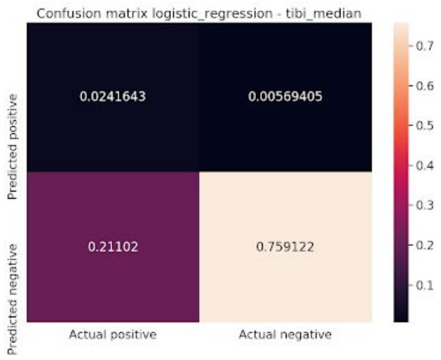

Confusion matrix logistic_regression - mean_normalized_fe
Predicted positive
Predicted negative
0.101316
0.0224479
0.153176
0.723059
Actual positive
Actual negative
0.7
0.6
0.5
0.4
0.3
0.2
0.1

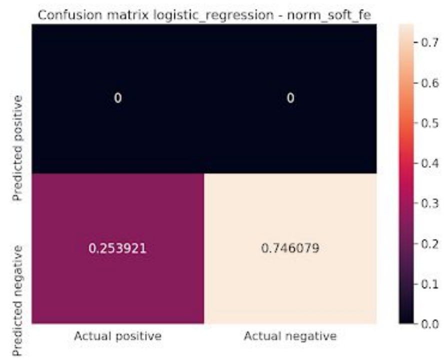

Confusion matrix logistic_regression - mean_scaled_fe
Predicted positive
Predicted negative
0.0932739
0.149715
0.161219
0.595792
Actual positive
Actual negative
0.5
0.4
0.3
0.2
0.1

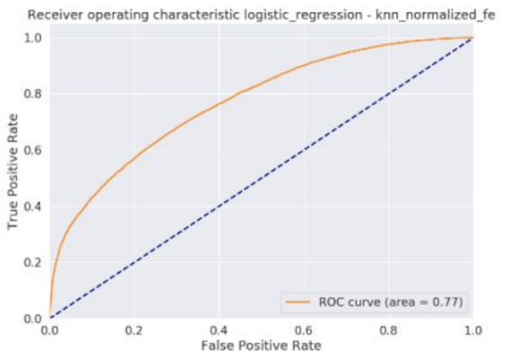

Confusion matrix logistic_regression - median_normalized_fe
Predicted positive
Predicted negative
0
0
0.254493
0.745507
Actual positive
Actual negative
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0

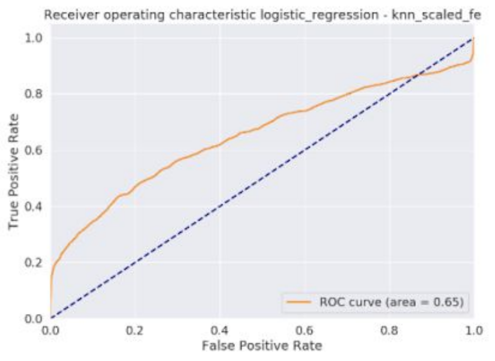

Confusion matrix logistic_regression - median_scaled_fe
Predicted positive
Predicted negative
0.072194
0.00395896
0.182299
0.741548
Actual positive
Actual negative
0.7
0.6
0.5
0.4
0.3
0.2
0.1

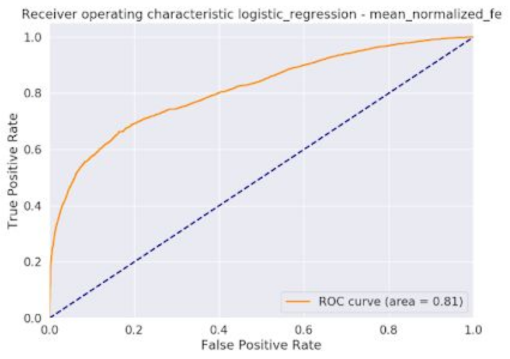

Confusion matrix logistic_regression - tibi_mean
Predicted positive
Predicted negative
0.0330595
0.00866856
0.202238
0.756034
Actual positive
Actual negative
0.7
0.6
0.5
0.4
0.3
0.2
0.1

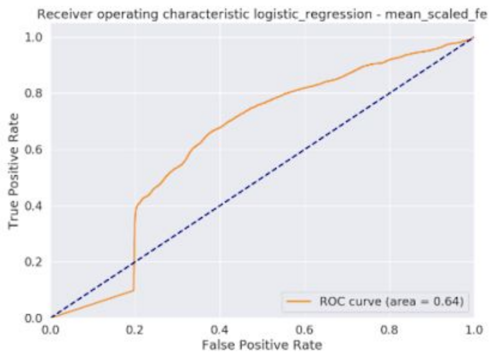

Confusion matrix logistic_regression - tibi_median
Predicted positive
Predicted negative
0.0241643
0.00569405
0.21102
0.759122
Actual positive
Actual negative
0.7
0.6
0.5
0.4
0.3
0.2
0.1

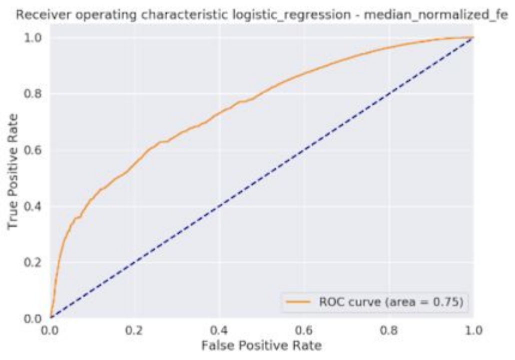

Confusion matrix logistic_regression - norm_soft_fe
Predicted positive
Predicted negative
0
0
0.253921
0.746079
Actual positive
Actual negative
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0

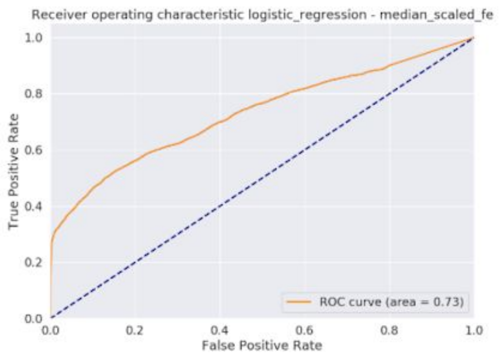

Receiver operating characteristic logistic_regression - knn_normalized_fe
True Positive Rate
False Positive Rate
ROC curve (area = 0.77)

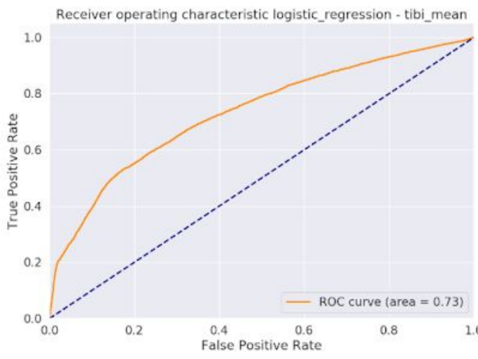

Receiver operating characteristic logistic_regression - knn_scaled_fe
True Positive Rate
False Positive Rate
ROC curve (area = 0.65)

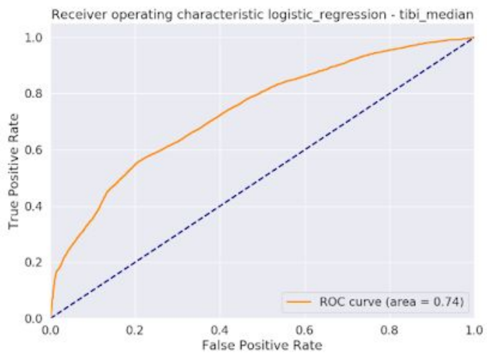

Receiver operating characteristic logistic_regression - mean_normalized_fe
True Positive Rate
False Positive Rate
ROC curve (area = 0.81)

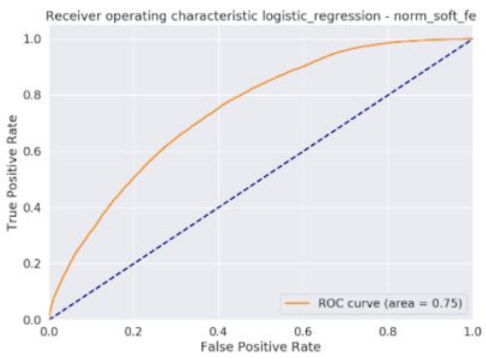

Receiver operating characteristic logistic_regression - mean_scaled_fe
True Positive Rate
False Positive Rate
ROC curve (area = 0.64)

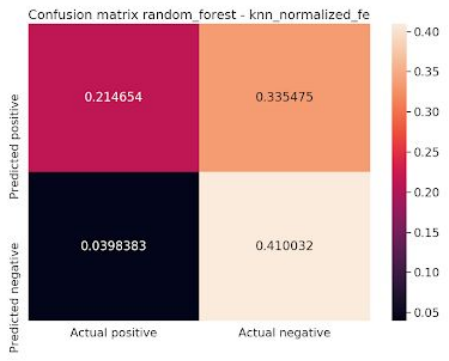

Receiver operating characteristic logistic_regression - median_normalized_fe
True Positive Rate
False Positive Rate
ROC curve (area = 0.75)

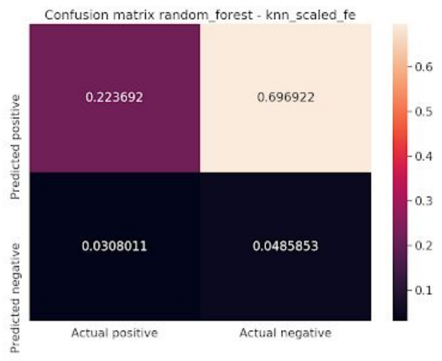

Receiver operating characteristic logistic_regression - median_scaled_fe
True Positive Rate
False Positive Rate
ROC curve (area = 0.73)

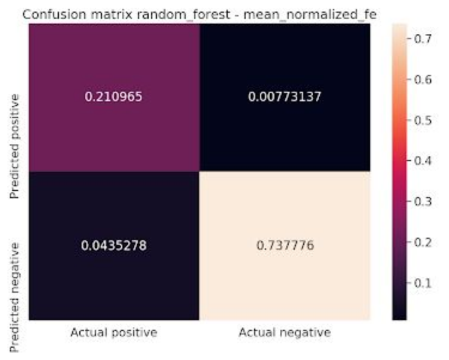

Receiver operating characteristic logistic_regression - tibi_mean
True Positive Rate
False Positive Rate
ROC curve (area = 0.73)

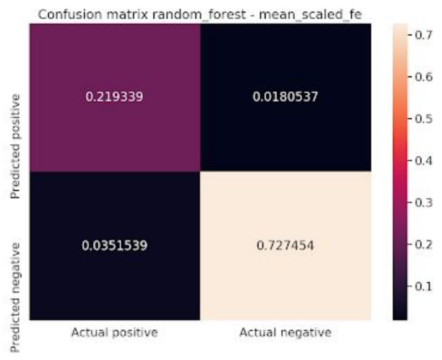

Receiver operating characteristic logistic_regression - tibi_median
True Positive Rate
False Positive Rate
ROC curve (area = 0.74)

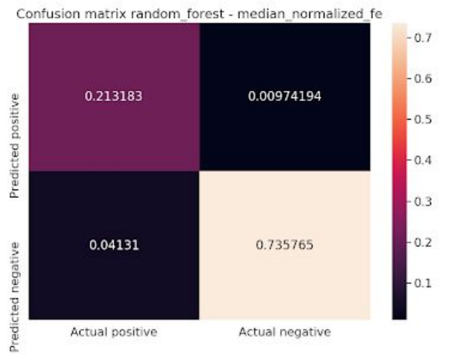

Receiver operating characteristic logistic_regression - norm_soft_fe
True Positive Rate
False Positive Rate
ROC curve (area = 0.75)

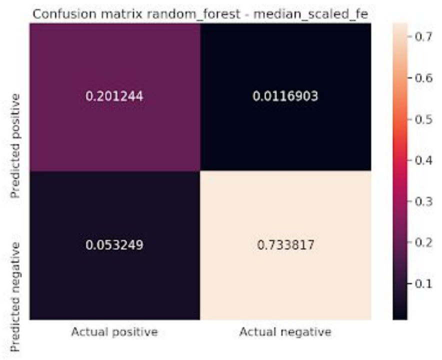

Confusion matrix random_forest - knn_normalized_fe
Predicted positive
Predicted negative
Actual positive
Actual negative
0.214654
0.335475
0.0398383
0.410032
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05

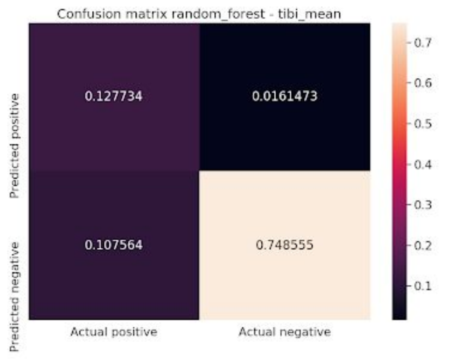

Confusion matrix random_forest - knn_scaled_fe
Predicted positive
Predicted negative
Actual positive
Actual negative
0.223692
0.696922
0.0308011
0.0485853
0.6
0.5
0.4
0.3
0.2
0.1

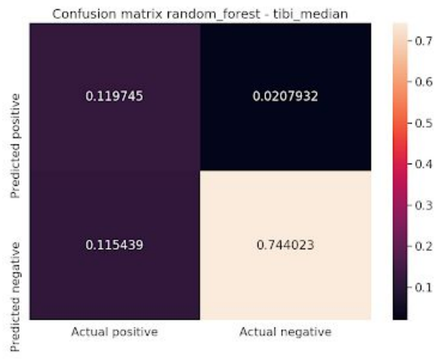

Confusion matrix random_forest - mean_normalized_fe
Predicted positive
Predicted negative
Actual positive
Actual negative
0.210965
0.00773137
0.0435278
0.737776
0.7
0.6
0.5
0.4
0.3
0.2
0.1

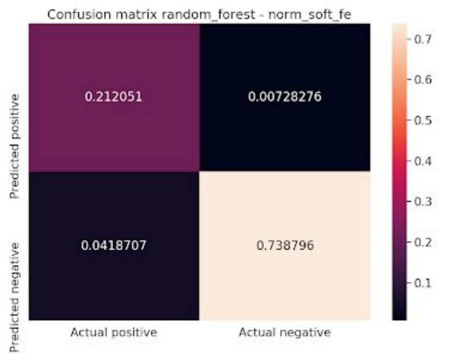

Confusion matrix random_forest - mean_scaled_fe
Predicted positive
Predicted negative
Actual positive
Actual negative
0.219339
0.0180537
0.0351539
0.727454
0.7
0.6
0.5
0.4
0.3
0.2
0.1

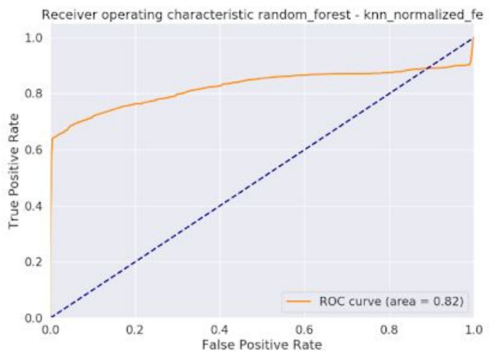

Confusion matrix random_forest - median_normalized_fe
Predicted positive
Predicted negative
Actual positive
Actual negative
0.213183
0.00974194
0.04131
0.735765
0.7
0.6
0.5
0.4
0.3
0.2
0.1

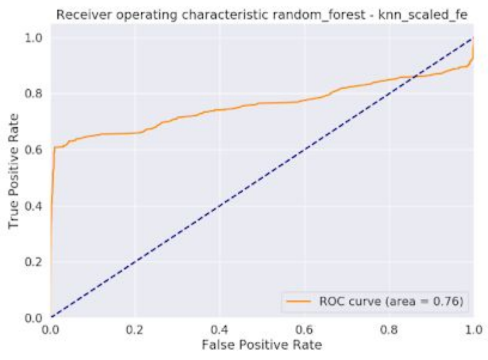

Confusion matrix random_forest - median_scaled_fe
Predicted positive
Predicted negative
Actual positive
Actual negative
0.201244
0.0116903
0.053249
0.733817
0.7
0.6
0.5
0.4
0.3
0.2
0.1

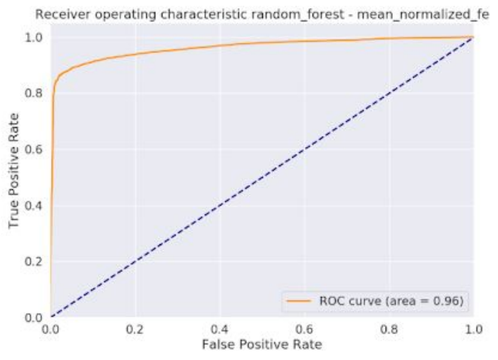

Confusion matrix random_forest - tibi_mean
Predicted positive
Predicted negative
Actual positive
Actual negative
0.127734
0.0161473
0.107564
0.748555
0.7
0.6
0.5
0.4
0.3
0.2
0.1

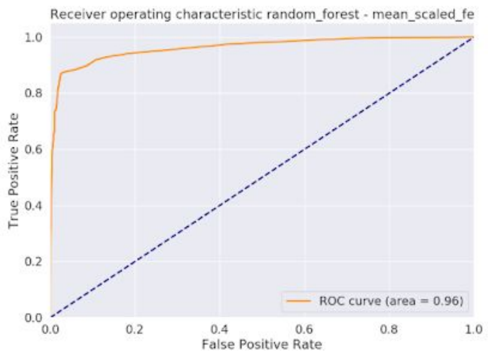

Confusion matrix random_forest - tibi_median
Predicted positive
Predicted negative
Actual positive
Actual negative
0.119745
0.0207932
0.115439
0.744023
0.7
0.6
0.5
0.4
0.3
0.2
0.1

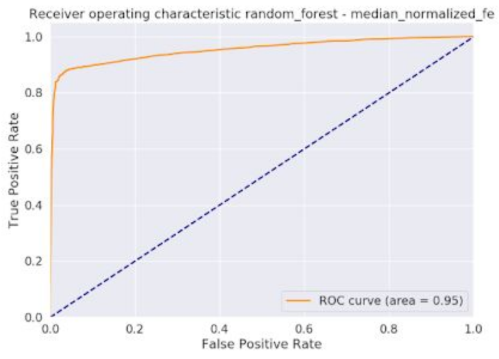

Confusion matrix random_forest - norm_soft_fe
Predicted positive
Predicted negative
Actual positive
Actual negative
0.212051
0.00728276
0.0418707
0.738796
0.7
0.6
0.5
0.4
0.3
0.2
0.1

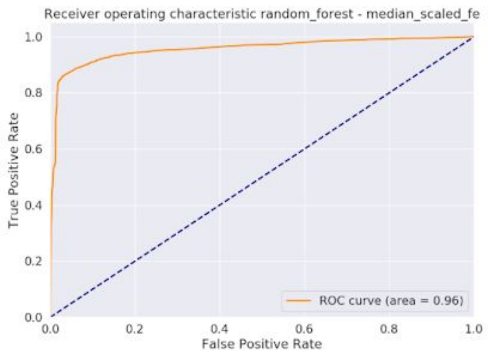

Receiver operating characteristic random_forest - knn_normalized_fe
True Positive Rate
False Positive Rate
ROC curve (area = 0.82)

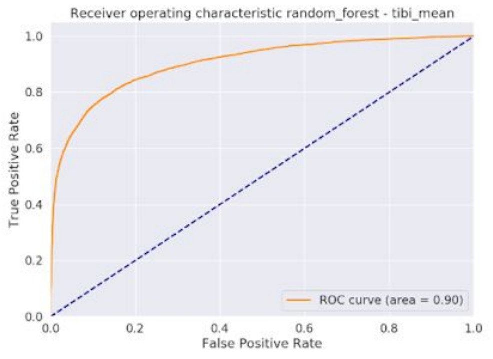

Receiver operating characteristic random_forest - knn_scaled_fe
True Positive Rate
False Positive Rate
ROC curve (area = 0.76)

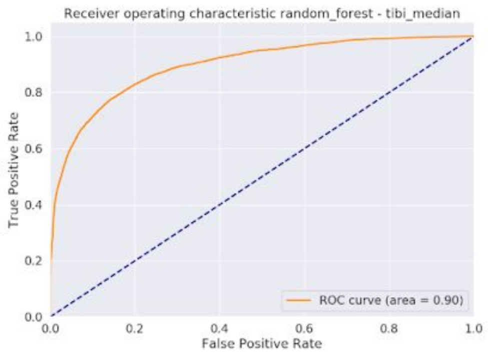

Receiver operating characteristic random_forest - mean_normalized_fe
True Positive Rate
False Positive Rate
ROC curve (area = 0.96)

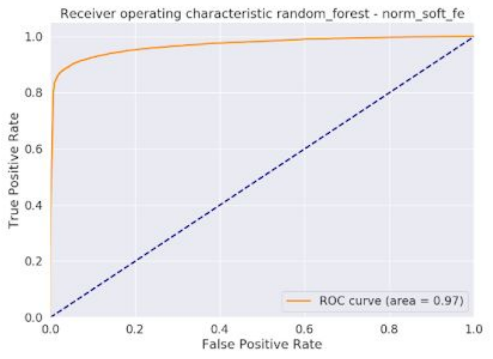

Receiver operating characteristic random_forest - mean_scaled_fe
True Positive Rate
False Positive Rate
ROC curve (area = 0.96)

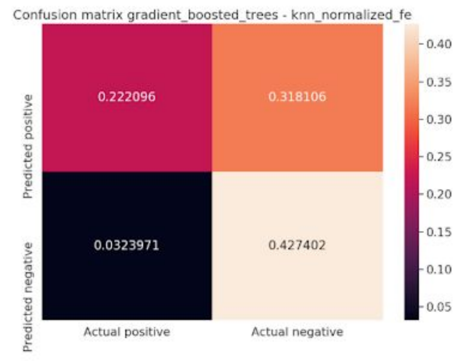

Receiver operating characteristic random_forest - median_normalized_fe
True Positive Rate
False Positive Rate
ROC curve (area = 0.95)

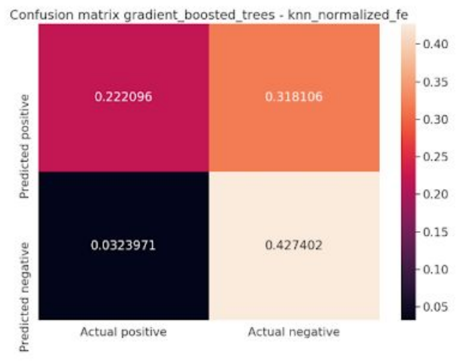

Receiver operating characteristic random_forest - median_scaled_fe
True Positive Rate
False Positive Rate
ROC curve (area = 0.96)

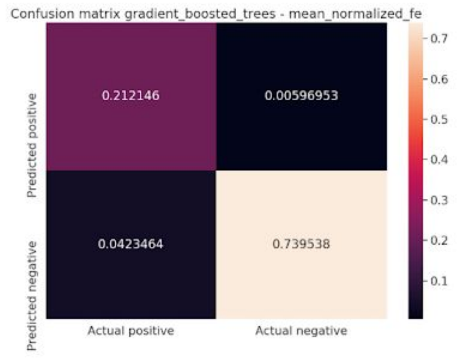

Receiver operating characteristic random_forest - tibi_mean
True Positive Rate
False Positive Rate
ROC curve (area = 0.90)

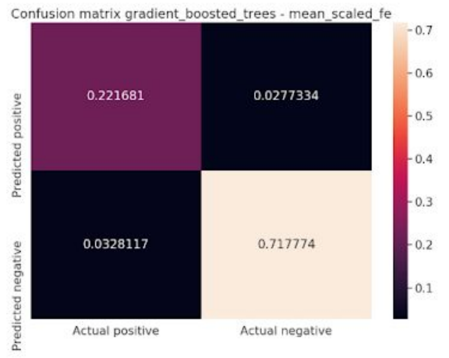

Receiver operating characteristic random_forest - tibi_median
True Positive Rate
False Positive Rate
ROC curve (area = 0.90)

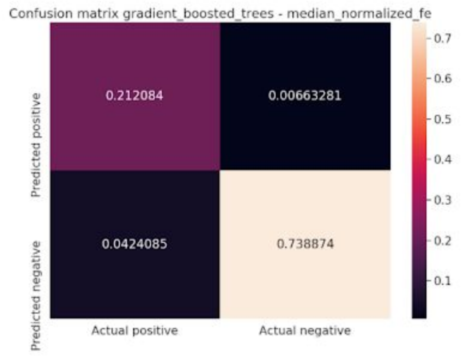

Receiver operating characteristic random_forest - norm_soft_fe
True Positive Rate
False Positive Rate
ROC curve (area = 0.97)

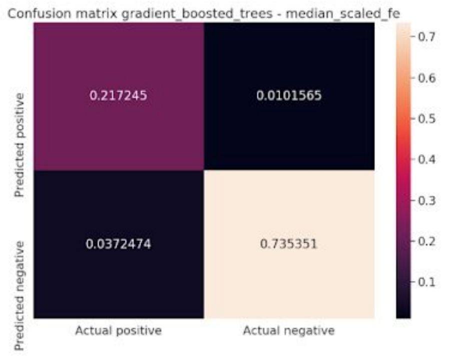

Confusion matrix gradient_boosted_trees - knn_normalized_fe
0.222096
0.318106
0.0323971
0.427402
Predicted positive
Predicted negative
Actual positive
Actual negative
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05

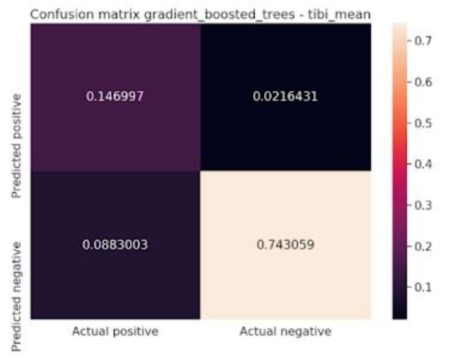

Confusion matrix gradient_boosted_trees - knn_normalized_fe
0.222096
0.318106
0.0323971
0.427402
Predicted positive
Predicted negative
Actual positive
Actual negative
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05

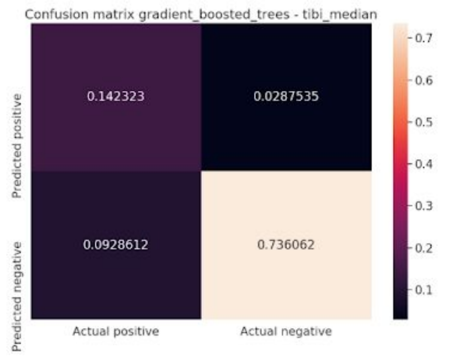

Confusion matrix gradient_boosted_trees - mean_normalized_fe
0.212146
0.00596953
0.0423464
0.739538
Predicted positive
Predicted negative
Actual positive
Actual negative
0.7
0.6
0.5
0.4
0.3
0.2
0.1

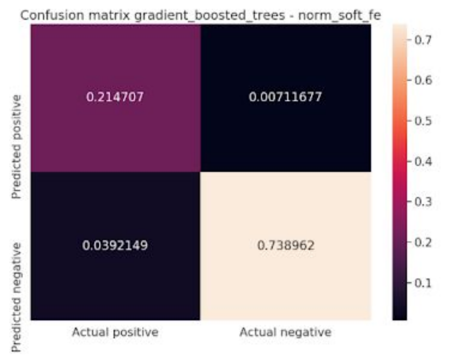

Confusion matrix gradient_boosted_trees - mean_scaled_fe
0.221681
0.0277334
0.0328117
0.717774
Predicted positive
Predicted negative
Actual positive
Actual negative
0.7
0.6
0.5
0.4
0.3
0.2
0.1

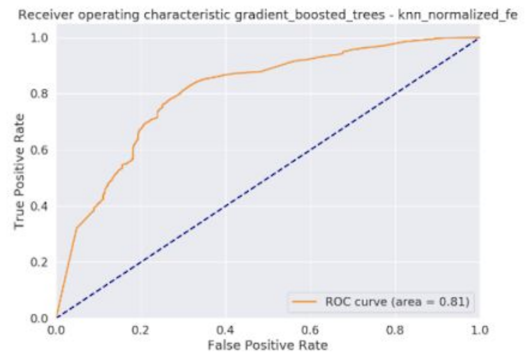

Confusion matrix gradient_boosted_trees - median_normalized_fe
0.212084
0.00663281
0.0424085
0.738874
Predicted positive
Predicted negative
Actual positive
Actual negative
0.7
0.6
0.5
0.4
0.3
0.2
0.1

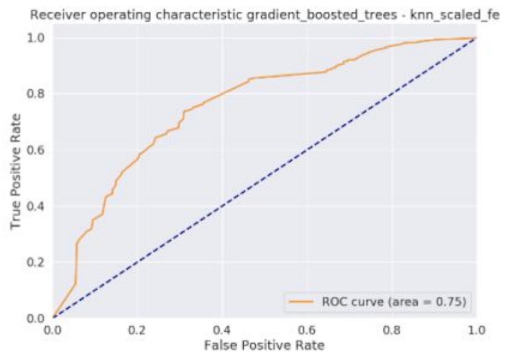

Confusion matrix gradient_boosted_trees - median_scaled_fe
0.217245
0.0101565
0.0372474
0.735351
Predicted positive
Predicted negative
Actual positive
Actual negative
0.7
0.6
0.5
0.4
0.3
0.2
0.1

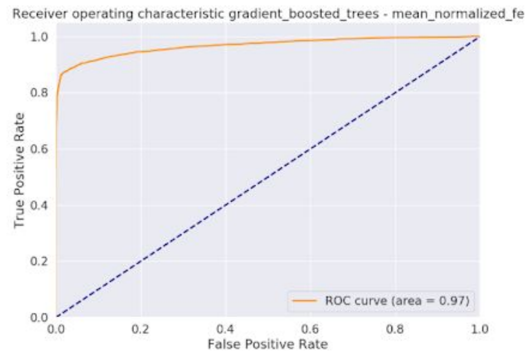

Confusion matrix gradient_boosted_trees - tibi_mean
0.146997
0.0216431
0.0883003
0.743059
Predicted positive
Predicted negative
Actual positive
Actual negative
0.7
0.6
0.5
0.4
0.3
0.2
0.1

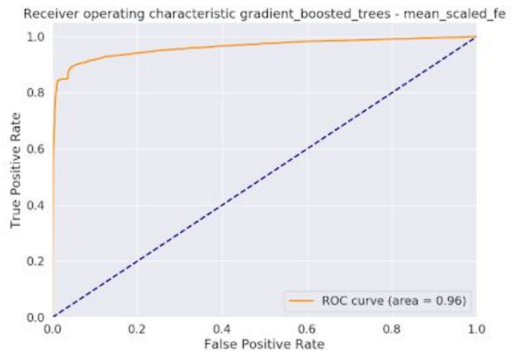

Confusion matrix gradient_boosted_trees - tibi_median
0.142323
0.0287535
0.0928612
0.736062
Predicted positive
Predicted negative
Actual positive
Actual negative
0.7
0.6
0.5
0.4
0.3
0.2
0.1

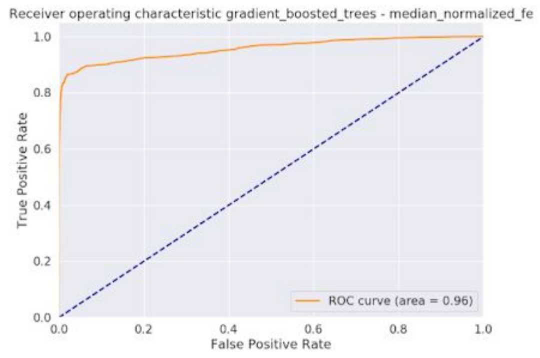

Confusion matrix gradient_boosted_trees - norm_soft_fe
0.214707
0.00711677
0.0392149
0.738962
Predicted positive
Predicted negative
Actual positive
Actual negative
0.7
0.6
0.5
0.4
0.3
0.2
0.1

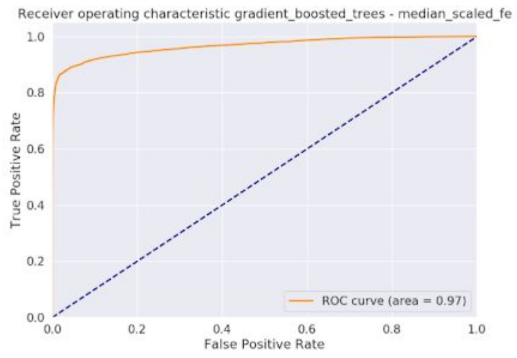

Receiver operating characteristic gradient_boosted_trees - knn_normalized_fe
True Positive Rate
False Positive Rate
ROC curve (area = 0.81)

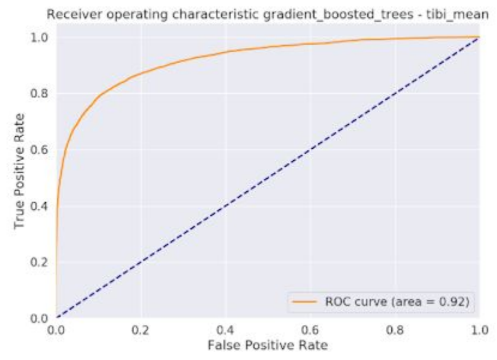

Receiver operating characteristic gradient_boosted_trees - knn_scaled_fe
True Positive Rate
False Positive Rate
ROC curve (area = 0.75)

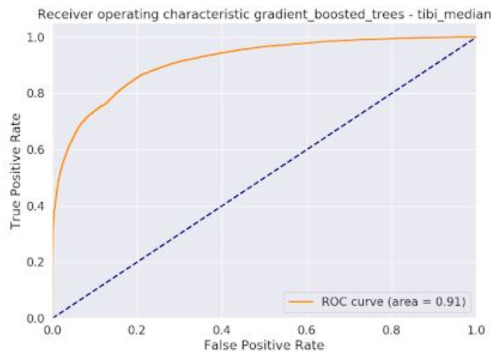

Receiver operating characteristic gradient_boosted_trees - mean_normalized_fe
True Positive Rate
False Positive Rate
ROC curve (area = 0.97)

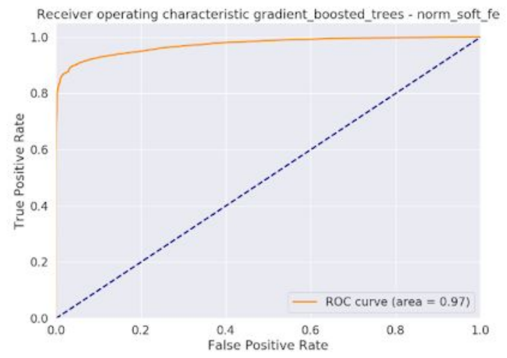

Receiver operating characteristic gradient_boosted_trees - mean_scaled_fe
True Positive Rate
False Positive Rate
ROC curve (area = 0.96)

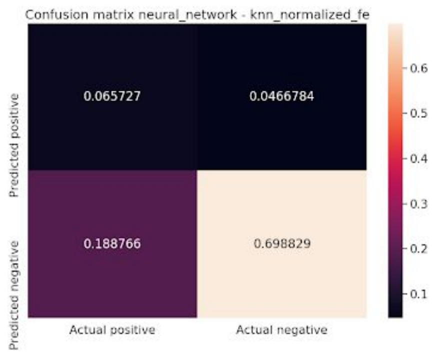

Receiver operating characteristic gradient_boosted_trees - median_normalized_fe
True Positive Rate
False Positive Rate
ROC curve (area = 0.96)

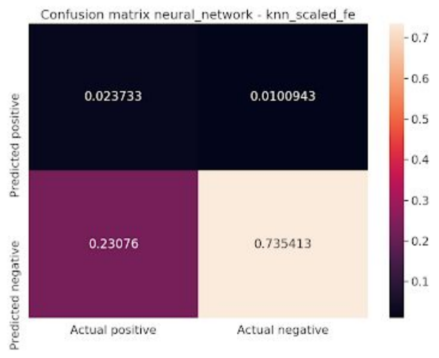

Receiver operating characteristic gradient_boosted_trees - median_scaled_fe
True Positive Rate
False Positive Rate
ROC curve (area = 0.97)

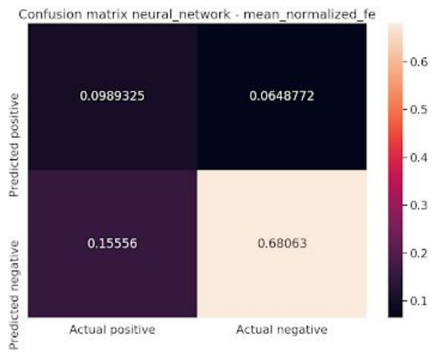

Receiver operating characteristic gradient_boosted_trees - tibi_mean
True Positive Rate
False Positive Rate
ROC curve (area = 0.92)

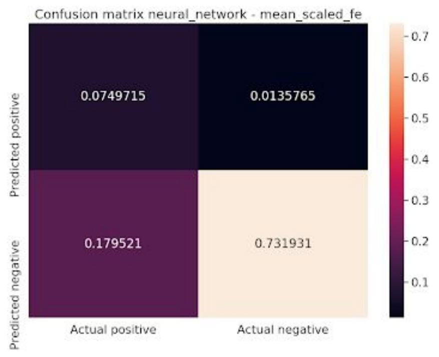

Receiver operating characteristic gradient_boosted_trees - tibi_median
True Positive Rate
False Positive Rate
ROC curve (area = 0.91)

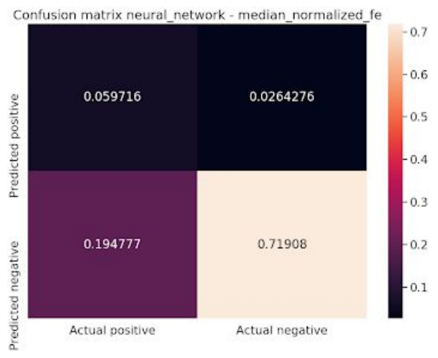

Receiver operating characteristic gradient_boosted_trees - norm_soft_fe
True Positive Rate
False Positive Rate
ROC curve (area = 0.97)

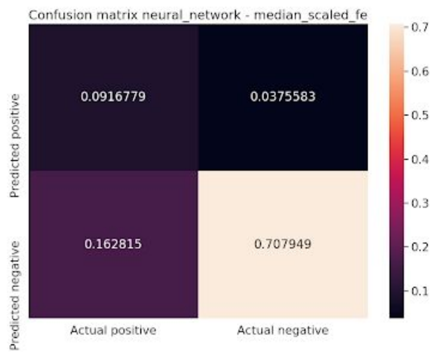

Confusion matrix neural_network - knn_normalized_fe
Predicted positive
Predicted negative
Actual positive
Actual negative
0.065727
0.0466784
0.188766
0.698829
0.6
0.5
0.4
0.3
0.2
0.1

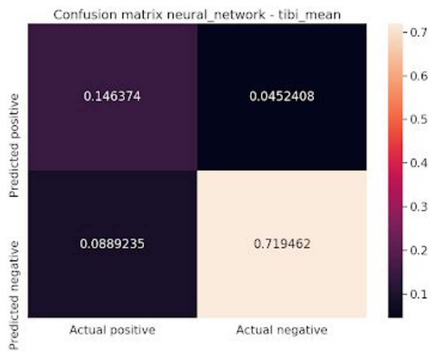

Confusion matrix neural_network - knn_scaled_fe
Predicted positive
Predicted negative
Actual positive
Actual negative
0.023733
0.0100943
0.23076
0.735413
0.7
0.6
0.5
0.4
0.3
0.2
0.1

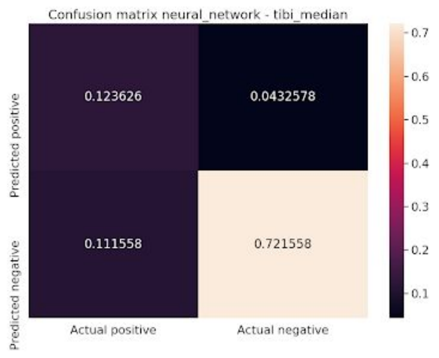

Confusion matrix neural_network - mean_normalized_fe
Predicted positive
Predicted negative
Actual positive
Actual negative
0.0989325
0.0648772
0.15556
0.68063
0.6
0.5
0.4
0.3
0.2
0.1

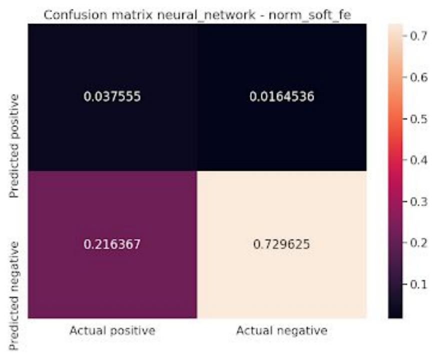

Confusion matrix neural_network - mean_scaled_fe
Predicted positive
Predicted negative
Actual positive
Actual negative
0.0749715
0.0135765
0.179521
0.731931
0.7
0.6
0.5
0.4
0.3
0.2
0.1

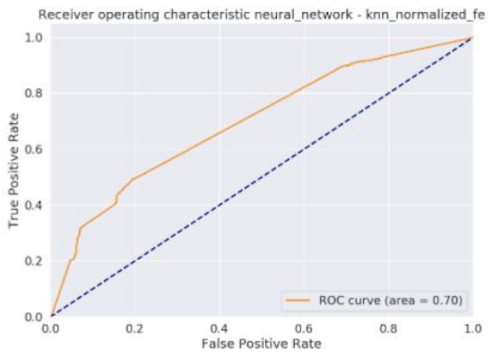
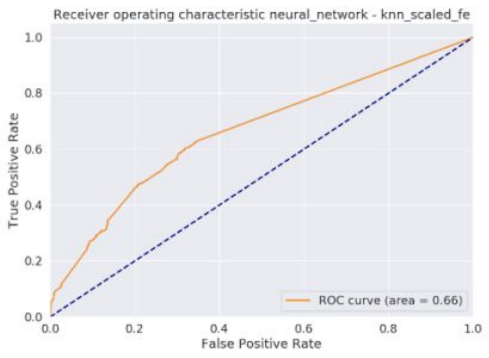
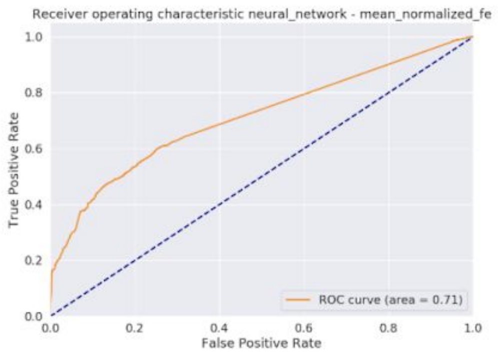
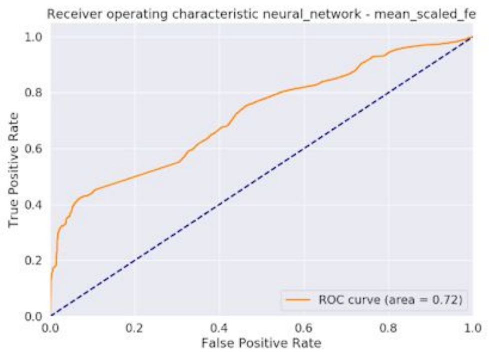
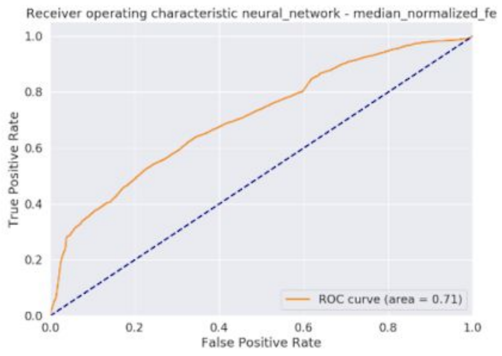
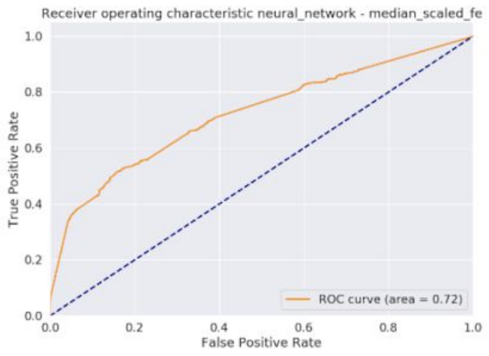
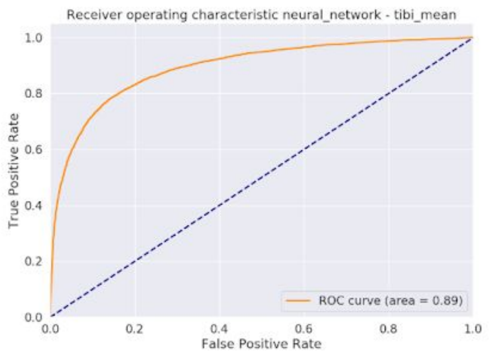
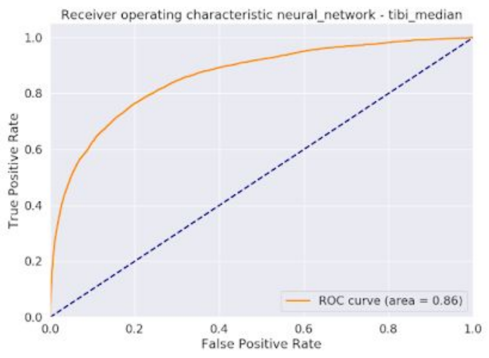
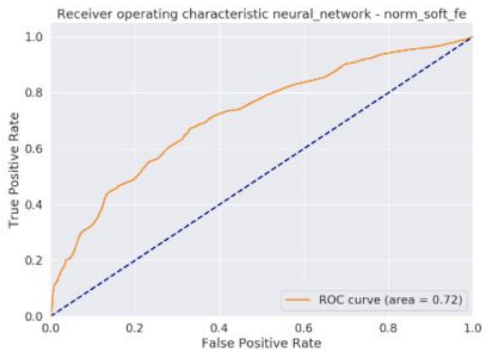

Confusion matrix neural_network - median_normalized_fe
Predicted positive
Predicted negative
Actual positive
Actual negative
0.059716
0.0264276
0.194777
0.71908
0.7
0.6
0.5
0.4
0.3
0.2
0.1

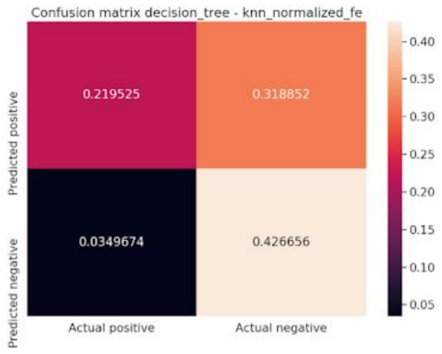

Confusion matrix neural_network - median_scaled_fe
Predicted positive
Predicted negative
Actual positive
Actual negative
0.0916779
0.0375583
0.162815
0.707949
0.7
0.6
0.5
0.4
0.3
0.2
0.1

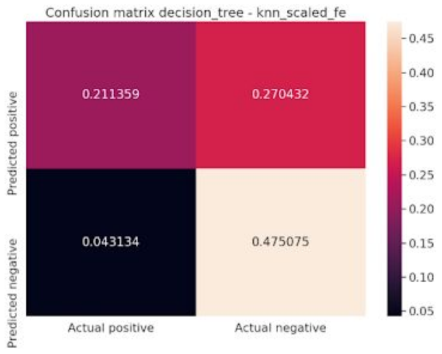

Confusion matrix neural_network - tibi_mean
Predicted positive
Predicted negative
Actual positive
Actual negative
0.146374
0.0452408
0.0889235
0.719462
0.7
0.6
0.5
0.4
0.3
0.2
0.1

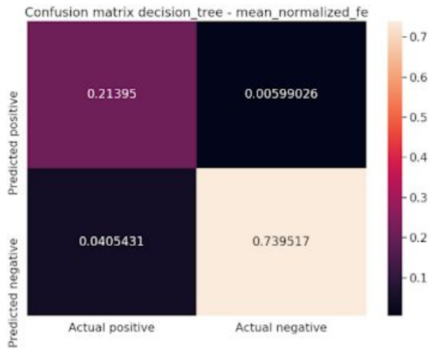

Confusion matrix neural_network - tibi_median
Predicted positive
Predicted negative
Actual positive
Actual negative
0.123626
0.0432578
0.111558
0.721558
0.7
0.6
0.5
0.4
0.3
0.2
0.1

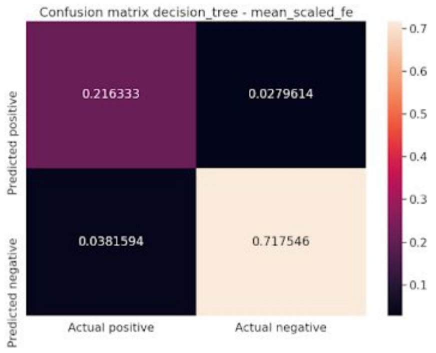

Confusion matrix neural_network - norm_soft_fe
Predicted positive
Predicted negative
Actual positive
Actual negative
0.037555
0.0164536
0.216367
0.729625
0.7
0.6
0.5
0.4
0.3
0.2
0.1

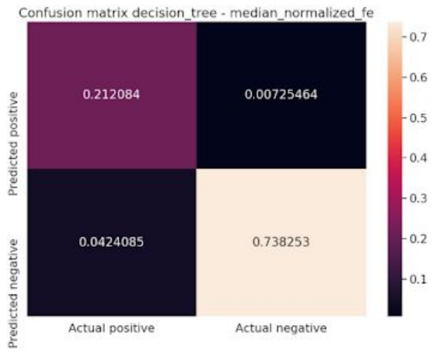

Receiver operating characteristic neural_network - knn_normalized_fe
ROC curve (area = 0.70)
True Positive Rate
False Positive Rate

Receiver operating characteristic neural_network - knn_scaled_fe
ROC curve (area = 0.66)
True Positive Rate
False Positive Rate

Receiver operating characteristic neural_network - mean_normalized_fe
ROC curve (area = 0.71)
True Positive Rate
False Positive Rate

Receiver operating characteristic neural_network - mean_scaled_fe
ROC curve (area = 0.72)
True Positive Rate
False Positive Rate

Receiver operating characteristic neural_network - median_normalized_fe
ROC curve (area = 0.71)
True Positive Rate
False Positive Rate

Receiver operating characteristic neural_network - median_scaled_fe
ROC curve (area = 0.72)
True Positive Rate
False Positive Rate

Receiver operating characteristic neural_network - tibi_mean
ROC curve (area = 0.89)
True Positive Rate
False Positive Rate

Receiver operating characteristic neural_network - tibi_median
ROC curve (area = 0.86)
True Positive Rate
False Positive Rate

Receiver operating characteristic neural_network - norm_soft_fe
ROC curve (area = 0.72)
True Positive Rate
False Positive Rate

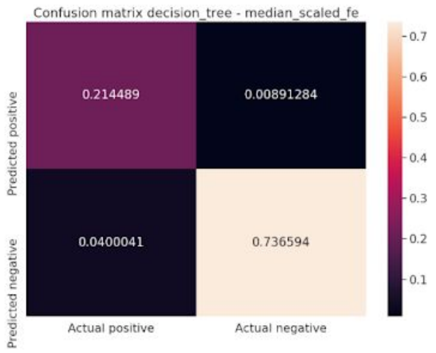

Confusion matrix decision_tree - knn_normalized_fe
Predicted positive
Predicted negative
Actual positive
Actual negative
0.219525
0.318852
0.0349674
0.426656

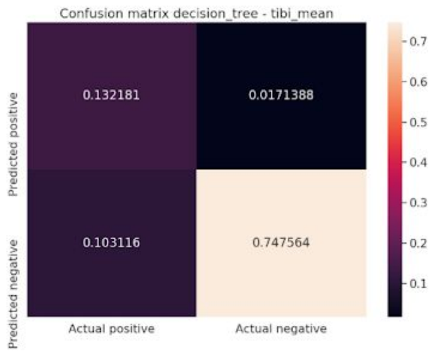

Confusion matrix decision_tree - knn_scaled_fe
Predicted positive
Predicted negative
Actual positive
Actual negative
0.211359
0.270432
0.043134
0.475075

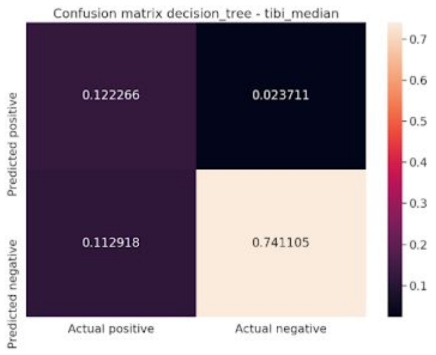

Confusion matrix decision_tree - mean_normalized_fe
Predicted positive
Predicted negative
Actual positive
Actual negative
0.21395
0.00599026
0.0405431
0.739517

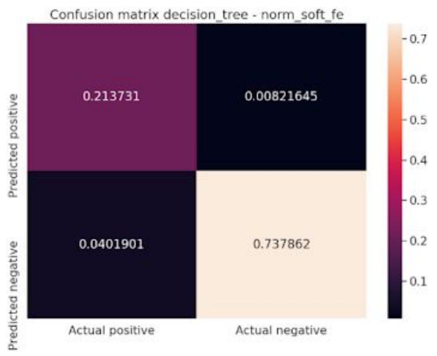

Confusion matrix decision_tree - mean_scaled_fe
Predicted positive
Predicted negative
Actual positive
Actual negative
0.216333
0.0279614
0.0381594
0.717546

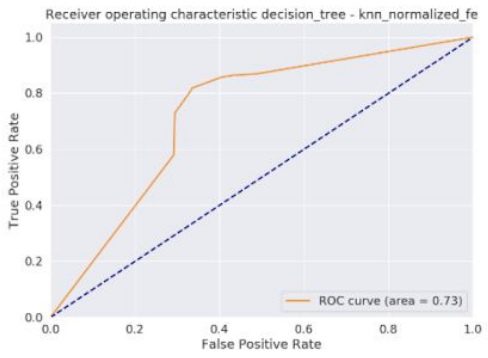

Confusion matrix decision_tree - median_normalized_fe
Predicted positive
Predicted negative
Actual positive
Actual negative
0.212084
0.00725464
0.0424085
0.738253

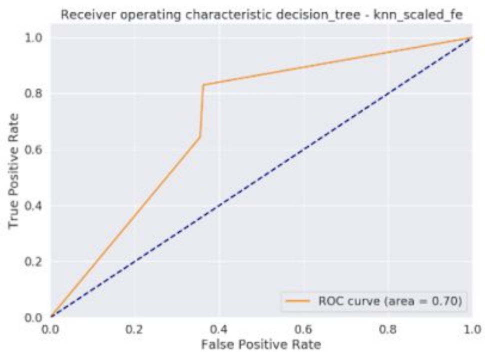

Confusion matrix decision_tree - median_scaled_fe
Predicted positive
Predicted negative
Actual positive
Actual negative
0.214489
0.00891284
0.0400041
0.736594

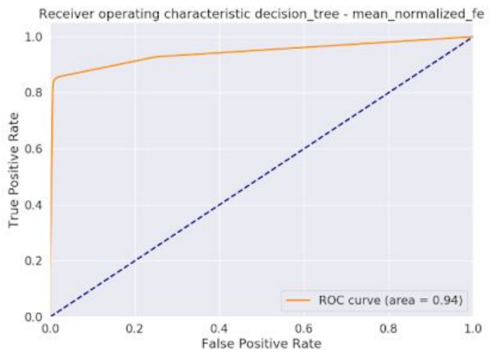

Confusion matrix decision_tree - tibi_mean
Predicted positive
Predicted negative
Actual positive
Actual negative
0.132181
0.0171388
0.103116
0.747564

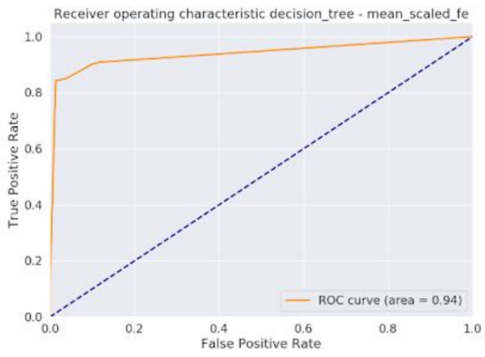

Confusion matrix decision_tree - tibi_median
Predicted positive
Predicted negative
Actual positive
Actual negative
0.122266
0.023711
0.112918
0.741105

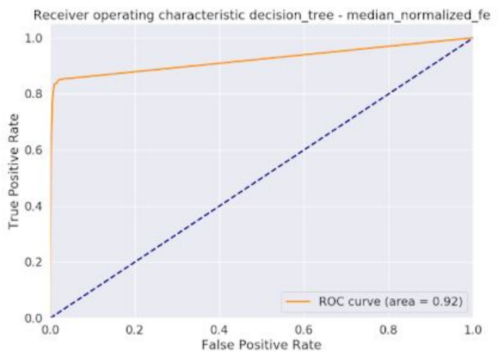

Confusion matrix decision_tree - norm_soft_fe
Predicted positive
Predicted negative
Actual positive
Actual negative
0.213731
0.00821645
0.0401901
0.737862

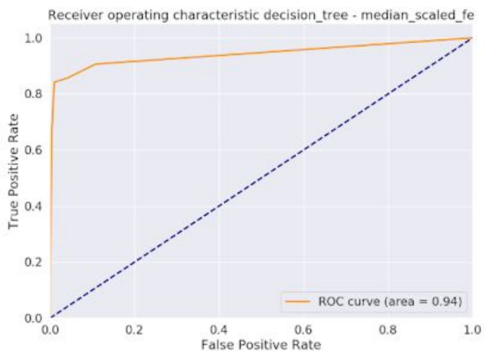

Receiver operating characteristic decision_tree - knn_normalized_fe
True Positive Rate
False Positive Rate
ROC curve (area = 0.73)

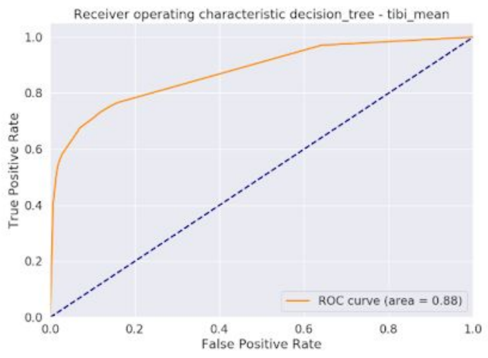

Receiver operating characteristic decision_tree - knn_scaled_fe
True Positive Rate
False Positive Rate
ROC curve (area = 0.70)

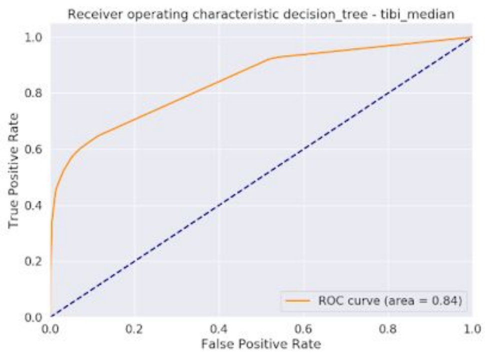

Receiver operating characteristic decision_tree - mean_normalized_fe
True Positive Rate
False Positive Rate
ROC curve (area = 0.94)

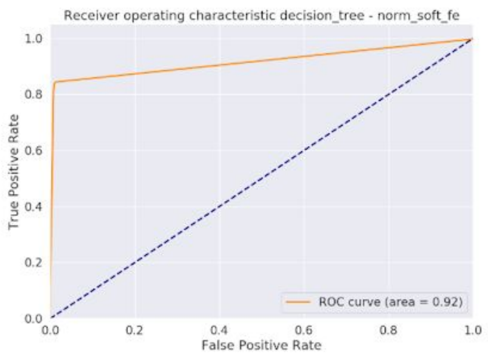

Receiver operating characteristic decision_tree - mean_scaled_fe
True Positive Rate
False Positive Rate
ROC curve (area = 0.94)


Receiver operating characteristic decision_tree - median_normalized_fe
True Positive Rate
False Positive Rate
ROC curve (area = 0.92)


Receiver operating characteristic decision_tree - median_scaled_fe
True Positive Rate
False Positive Rate
ROC curve (area = 0.94)


Receiver operating characteristic decision_tree - tibi_mean
True Positive Rate
False Positive Rate
ROC curve (area = 0.88)


Receiver operating characteristic decision_tree - tibi_median
True Positive Rate
False Positive Rate
ROC curve (area = 0.84)


Receiver operating characteristic decision_tree - norm_soft_fe
True Positive Rate
False Positive Rate
ROC curve (area = 0.92)