**Applied Machine Learning**

**Group Practical Project**

**IoT Intrusion Detection Competition using Machine Learning**

**Students**

**Alberto Matuozzo (13174490)**

**David Kirby (13167401)**

**Adam Elshimi (12825564)**

**Ali Sahin (13175649)**

**Andrew Robinson (13122132)**

**Introduction**

The Internet of Things (IoT) is a paradigm that involves networked physical objects with embedded technologies to collect, communicate, sense and interact with the external environment through wireless or wired connections. With rapid advancements in IoT technology, the number of IoT devices is expected to surpass 50 billion by 2020, which has also drawn the attention of attackers who seek to exploit the merits of this new technology for their own benefits [1].

**Aim**

The aim of this project is to address the issue of security threats in IoT devices by building a predictive machine learning model to distinguish between “intrusive” traffic, called intrusions or attacks, and “good” normal traffic.

**Objectives**

To explore the data and identify any pre-processing and feature selection required.

To experiment with alternative machine learning algorithms and identify the most suitable for intrusion detection.

To refine the chosen algorithm by tuning the hyperparameters.

To evaluate the algorithm performance and set a direction for future work.

**Data**

The predictive machine learning model will focus on detecting impersonation attacks in a reduced CLS portion of the Aegean Wi-Fi Intrusion Dataset (AWID) which was originally prepared and managed by George Mason University and the University of the Aegean.

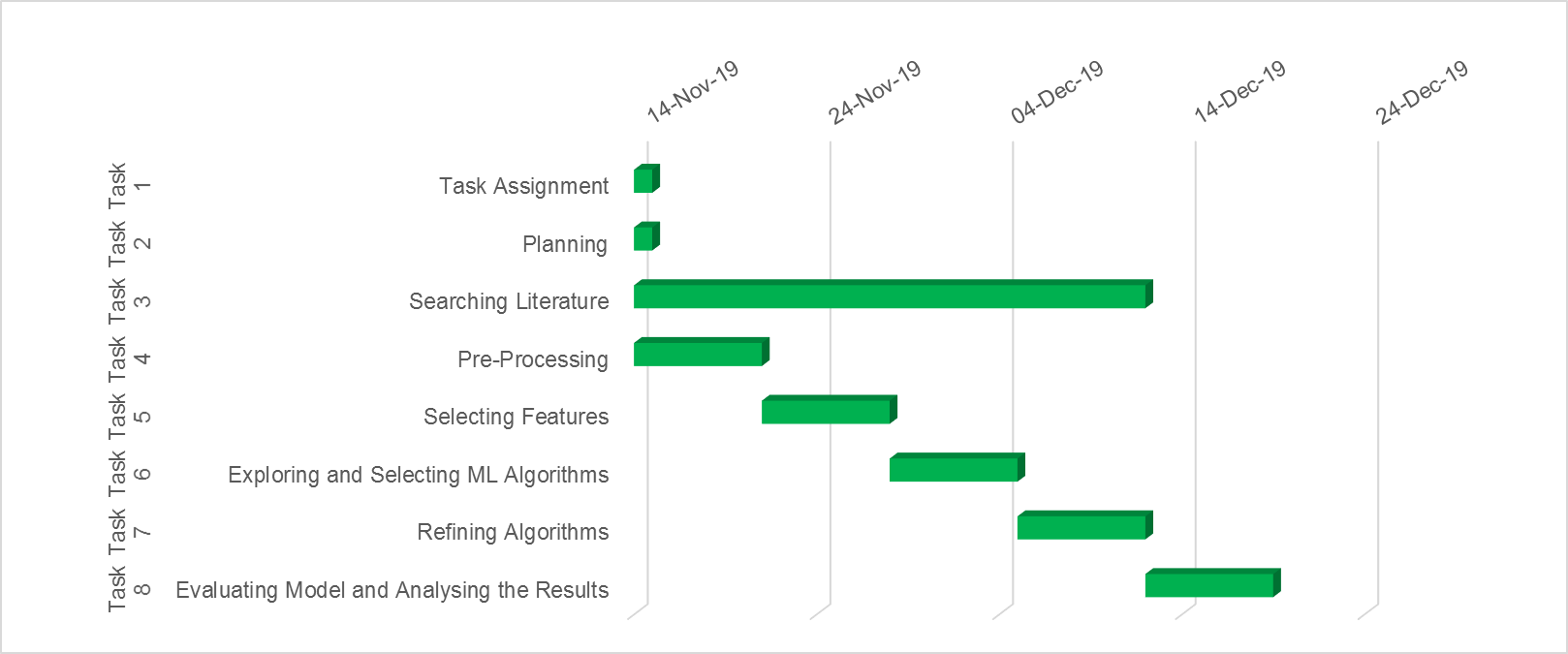
**Planning**

A total of 10 tasks were defined and assigned to the members of the team as follows:

* Task Assignment – Group Discussion
* Planning – Group Discussion
* Searching Literature – Each group member to read around their assigned task and record references.
* Pre-Processing – Alberto Matuozzo
* Selecting Features – David Kirby
* Exploring and Selecting Machine Learning (ML) Algorithms – Adam Elshimi
* Refining Algorithms – Ali Sahin
* Evaluating and Analysisng the Results - Andrew Robinson
* Potential for future work – Group Discussion
* Write up – Each team member to develop their own phase, and group contribution to the other sections.

**Schedule**

A deadline of 12th December was established for all the different phases involving literature research and practical programming up to hyperparameters tuning. The evaluation task was to be completed by the 19th of December. Given the iterative and experimental nature of any machine learning project, a further two weeks were set aside for additional experiments and writing up each task before the project deadline.

Furthermore, the below Gantt Chart was put togther to provide an overview of the project timeline:

**Resources**

Each team members use their own computers. All code to be written using Juypter notebooks.

GitHub would be used to develop and store our project notebooks and other related documents .

University library and computer lab to facilitate literature research and group tasks.

**Progress Monitoring**

A weekly meeting is organised to discuss the progress made during a given week, brainstorm ideas and decide action points for the following week.

One person would take and circulate the minutes of the meeting.

**Pre-Processing - Alberto Matuozzo**

The aim of the pre-processing stage is to analyse the raw dataset and make the necessary modifications to put it in the most effective conditions to perform the actual machine learning task.

Pre-processing steps can be grouped into data cleaning, data reduction and data transformation.

Data Cleaning

As shown by Little and Rubin[2], It is important to identify the presence of missing data and implement an appropriate policy to solve the issue. In this case the training set is complete.

The presence of duplicate cases can create a bias in the dataset causing the algorithm to learn improperly the persistence of a pattern. McKinney [3] highlights two options to deal with duplicates: keep them, or make the assumption that they are redundant, and discard them. The training set has 547 duplicate observations. Our assumption is that intrusion attacks are independent therefore duplication in this case is considered as persistence of certain patterns. Therefore, we decided to keep the duplicates.

Data Reduction

This step aims to simplify the dataset eliminating features that are irrelevant at best and constitute noise at worst. Effective data reduction results in greater learning speed and potentially better accuracy.

Inspecting the standard deviation of the features matrix highlighted a minimum of zero (figure 1). Looking at the value counts for each level of standard deviation (figure 2) revealed that 74 predictors assume the exact identical value across every training instance. They are redundant, therefore have been eliminated using the *Scikit Learn VarianceThreshold* selector. The resulting dataset has been named *Xselected.*

**figure 1**. Training set. Features Matrix descriptive statistics.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Min** | **Max** | **Std\_Dev** | **Skew** |
| Min | 0.000000 | 0.000000 | 0.000000 | -67.95797 |
| Mean | 0.016447 | 0.438865 | 0.055441 | 13.28332 |
| Max | 1.000000 | 1.000000 | 0.473930 | 311.5188 |

**figure 2**. Training set. Features Standard deviation: top 5 value counts.

|  |  |
| --- | --- |
| **Std\_Dev** | **Nr Features** |
| 0.000000 | 74 |
| 0.014709 | 11 |
| 0.354444 | 2 |
| 0.003210 | 2 |
| 0.062765 | 2 |

Data Transformation

Different algorithms make certain assumptions about the probability distribution of data and or work best when it comes to optimizing parameters if input features are within a certain range.

Given that there is not an algorithm that performs better than others a priori on a given task, the output of the data transformation phase consists of several datasets to experiment on.

The predictors are heterogeneous: some variables appear discrete, others binary, others continuous. As suggested in Kolias *et al.* [4] and in Aminanto *et al*. [5], a normalization step has been performed so that every predictors value range is between zero and one. *Scikit Learn* *MinMaxScaler* has been applied to the *Xselected* dataset.

Using skew as a measure of symmetry around the mean of each feature, highlights that the respective frequency distributions deviate substantially from the Gaussian (figure1). Kolias et al. [4] showed that the Naive Bayes Algorithm was the most accurate (albeit not to a satisfactory level) in detecting impersonation attack. This algorithm, when dealing with continuous features, assumes a Gaussian distribution of the predictor given a class.

Yeo and Johnson [6] proposed a power transformation to reduce skew that does not require specific underlying assumptions on the variable to be transformed. This power transformation has been implemented on the *Xselected* dataset using *Scikit Learn PowerTransformer.* As a result, the parameters of each feature distribution are closer to a Gaussian.

A final and alternative transformation has been performed on the *Xselected* dataset. Given the heterogeneity of features, it is useful to scale each instance to have a norm of one in case at a later stage of the project an algorithm like Neural Network that weights input values, or that uses kernels (for example an SVM) to quantify the similarity of different examples will be deployed. This transformation has been implemented using *Scikit Learn Normalizer*.

At the end of the pre-processing phase, four datasets are available to experiment with different feature and model selections:

1. X.selected: features with variance greater than zero
2. NX3df: Xselected with features value in range [0,1]
3. PX2df: Xselected with a power Gaussian tranformation
4. NX2df: Xselected with each training instance to have unit norm.

The best results have been achieved just removing the features with zero variance.

**Selecting Features - David Kirby**

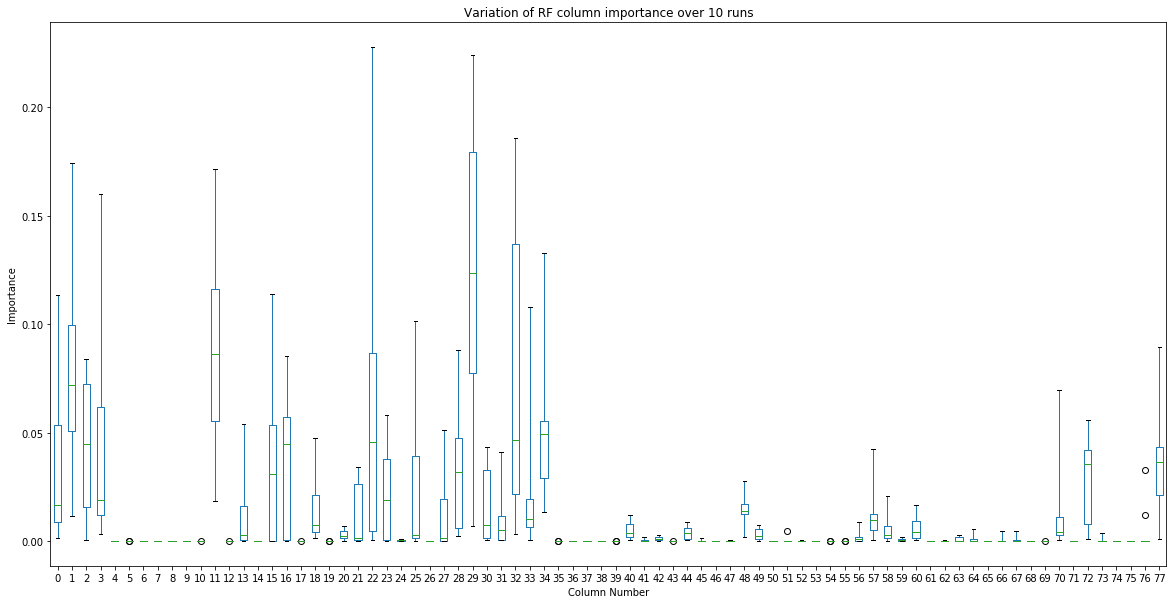
**Feature Selection**

The purpose of feature selection is to reduce the dimensionality of the problem space by eliminating features from the model that are redundant or unnecessary for prediction. This can improve accuracy if the eliminated features are irrelevant to the model or are strongly correlated to other features. Feature selection can also speed up training and prediction times and make the model more interpretable.

**Embedded Feature Selection Via Tree Based Algorithms**

The initial method chosen for feature selection was to use tree based supervised learning algorithms to identify the most relevant features. Three algorithms were selected for comparison: Random Forest [7], XGBoost [8] and CatBoost [9]. As explained in the previous section columns with zero variance had been removed leaving 78 features as a starting point for further elimination.

Each model was fit to the training data and their feature importance ratings compared to see which columns they had in common. However it was observed that the ratings given by Random Forest were not stable - repeatedly fitting to the same data set gave substantially different results each time. Figure 3 shows a boxplot for each feature with the variation in feature importance assigned by Random Forest over 10 runs. Both XGBoost and CatBoost were consistent in their importance measures. Due to this variability Random Forest was dropped as a means of selecting features.

  
**figure 3**

The top twenty features selected by the remaining algorithms were (going from lowest importance to highest):

xgboost: [31, 25, 28, 52, 32, 49, 13, 44, 20, 56, 18, 0, 72, 2, 48, 11, 34, 57, 29, 22]  
catboost: [77, 49, 31, 20, 70, 30, 1, 0, 33, 32, 72, 18, 48, 2, 3, 34, 57, 29, 22, 11]

N.B. These are the column indexes after the zero-variance columns have been removed. These both accounted for over 99% of the importance values assigned by the respective algorithms.

The intersection of these gave a set of 14 features that both XGBoost and CatBoost considered important:

XGBoost and CatBoost Intersection: {0, 32, 2, 34, 72, 11, 48, 49, 18, 20, 22, 57, 29, 31}

To validate the usefulness of these features 5-fold cross validation on the training set was performed with both XGBoost and CatBoost using the set of 14 features. The results were promising, with accuracies upwards of 99.96% and for some folds 100%. The number of features was further reduced to six using the RFECV (Recursive Feature Elimination with Cross Validation [10]) and the CatBoost algorithm. However although this still gave a cross-validation accuracy in excess of 99.98%, when the model was used to predict the test data set the accuracy dropped to 53-54%, barely better than chance.

Investigation into the cause of the poor accuracy against the test data set found that some features in the test data were radically different from the same features in the training data. The data had been pre-processed by a third party before we received it to scale all the features into the range 0 to 1. It appears that for some columns this had been done incorrectly so the distributions in the test data set did not match the distributions in the training data set. Any models that depended on these erroneous features would behave poorly against the test data. See appendix A for further information.

**Simple feature selection using SelectKBest, Chi-squared and ANOVA F-value metrics**

It was decided that selectively filtering out the mismatched columns between the training and test sets would be liable to overfitting the model to the test data, so an alternative approach was taken of using a simpler method for feature selection and using a wider range of features. It was found that selecting 20-40 features by doing a Chi-squared or ANOVA comparison with the target data and selecting the K best gave good results for some algorithms. Chi-squared was tried initially but does not work with the negative values created by the power transformation, so ANOVA F-value was added as an alternative.

The exact number of features to select and which metric to use was left as part of the model tuning stage of the process.

**Exploring and Selecting Machine Learning Algorithms - Adam Elshimi**

**Introduction**

After having pre-processed the dataset, used feature selection to reduce our dataset, categorised the problem and understood the data, we identified the algorithms that are applicable and practical to implement. Furthermore, before having set out to select the correct algorithm we chose the following 3 measurements to be the bases of our evaluation and point of comparison of other algorithms:

1. The accuracy of the model.
2. Time taken to make predictions using the model
3. Time taken to build, train, and test the model

**ML Algorithms**

Having set the groundwork with the phase1 and 2 the algorithm we decided to use as a baseline for our machine learning algorithm phase, was Xgboost, which in the paper [11] outline the benefits over many other machine learning models.

Xgboost is a greedy gradient boosting algorithm, which is a decision-tree ensemble method that performs well on structured data. A common issue that arises with tree learning is finding the best split that leads to optimal results indicated by equation in [12]. Xgboost tackle this problem by using a depth-first approach, which is a tree pruning method that has shown to significantly increase performance whilst reducing the time complexity of the algorithm [12].

The following algorithms Catboost, logistic regression, naïve Bayes, Random Forest and decision tree classifier (Adaboost) are the five extra algorithms we decided test.

Catboost is a unbiased gradient boosting algorithm having shown to outperform many of the latest and best algorithms [13], this was a step up from Xgboost, however, it took a significant amount longer for making predictions. Logistic regression, Random Forest, and naïve Bayes have previously been proven to work well with AWID dataset, as shown in the paper by Dr Paul and his counterparts [14] where tree learning and simple algorithms like logistic regression proved to be both interpretable and predict with high accuracy

**Classifier**

Boosting is generally a technique that derives from Probably Approximately Correct (PAC) models – a theoretical learning framework with the aim to show that an algorithm achieves low empirical risk (ER) error with high probability. However, for this model we are adopting a more general learning framework where we use Adaboost – an ensemble boosting algorithm created by Yoav Freund and Robert Schapire in 1996 [15] to tackle the problem of overfitting, increase accuracy through combination of weak learners and an algorithm which is simple to implement.

The formula for classifying with Adaboost is the following:

A picture containing object

Description automatically generated

Figure 4

Referencing figure4, f(x) is the function representing the mth classifier and theta is tha mth corresponding weight. Thus, this formula is the combination of all the weights of all mth classifiers (ensemble method).

After thorough investigation the Adaboost classifier was chosen as the classifier for our model, which is evaluated in the evaluation section.

**Refining Algorithms - Ali Sahin**

Model hyperparameters are parameters that cannot be derived from the data and therefore must be specified before utilising the model. Default parameters are initially used at the beginning of the machine learning process and are adjusted after model selection. The aim of optimisation is to find a set of parameters which minimise a predefined loss function; exponential loss in adaboost and log loss in logistic regression [16].

Figure 5: Formularisation of hyperparameter search. Where represents the hyperparameters, X (te) represents the test data, represents the training data [1].

Figure 5 shows the mathematical representation of hyperparameter search. There are many algorithms which automate this process. The most commonly used are grid search and Bayesian search. These algorithms are preferential to the alternative of manually adjusting the parameters, as research shows this results in models with greater predictive performance [17]. Bayesian search was not used due to the time constraints. The sklearn library at the time of conducting the project, did not have a dedicated class to utilise the algorithm and therefore would have required significant time to code. Hyperparameters were evaluated using a tenfold cross validation (cv). Initially kbest was included in the grid search but then removed. Including kbest resulted in poor predictive performance on the test data for the adaboost models, accuracy decreased from 99.07% to 50%. This was the result of the adaboost models overfitting the training data.

For the final three pipelines, the parameters which were tuned are as follows: number of estimators, learning rate, ‘C’ and solver. The number of estimators (N) and learning rate (L) are a parameter of adaboost. There is a trade-off between N and L, the optimal number of estimators required increases as L decreases [18]. The number of estimators is the number of models that are iteratively trained: the values searched were 10,50,100,150. The learning rate is the contribution each model has on the weights of the algorithm. The values 0.001,0.01,0.1 were searched, as past studies suggest values below 0.1 are optimal [18]. The lowest L values should have the greatest cross validation score as a result of reduced overfitting however, the opposite was true. The highest L achieved the best cv score. The parameters Solver and ‘C’ are used to optimise logistic regression. The inverse of regularisation strength is referred to as ‘C’. The lower the ‘C’ the greater the regularisation strength. Regularization can be interpreted as a penalty term that discourages complex weights by penalising large weights. This improves the generalisability of the model by reducing the variance and therefore should reduce overfitting. However, the highest C values (1000) resulted in the greatest cv score [19].

**Evaluating Model and Analysing the Results – Andrew Robinson**

After selecting and refining numerous algorithms three final models were selected as the focus of the evaluation. Pipelines were set up for each model before final refinements were made. Each model was trained using a version of the, previously mentioned, AWID dataset which was balanced to show equal numbers of attack and non-attack instances. The models were then evaluated using a testing dataset which was subjected to the same modifications and was made up of 40,158 observations (20,079 normal traffic (non-attacks) and 20,079 impersonation attacks).

In order to give a full overview of the performance of each model a number of performance measures have been calculated. The majority of these measures involve calculating the: True Positive (TP) values which show intrusions correctly identified as an attacks; True Negative (TN) values which show normal instances correctly identified as non-attack; False Positive (FP) values or normal instances incorrectly identified as an attack; and False Negative (FN) values showing Intrusions incorrectly identified as non-attack. These values were obtained from the classification report run on each pipeline.

A brief summary of each performance measure is outlined below:

In addition to the above measures Time to Build (TTB) in seconds will also be calculated for each model. In order to maintain a level playing field, all models will be run in the same environment using a mac OS with 1.4 GHz Intel i5 CPU and 8 Gb of RAM. Furthermore additional information on each performance measure above can be found in the DEMISe Techniques for IoT Intrusion Detection paper [14] should it be needed.

**Results**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Pipeline 1 | Pipeline 2 | Pipeline 3 |
| Acc (%) | 99.08 | 99.08 | 98.63 |
| DR (Recall) (%) | 98.15 | 98.15 | 97.96 |
| Precision (%) | 100 | 100 | 99.29 |
| FAR (%) | 0 | 0 | 0.01 |
| FNR (%) | 1.85 | 1.85 | 2.04 |
| F1 (%) | 99.07 | 99.07 | 98.62 |
| MCC (%) | 98.17 | 98.17 | 97.26 |
| TTB (seconds) | 7.24 | 7.4 | 5.57 |

**Evaluation – Pipeline 1**

Pipeline 1 is made up of the following:

* Variance Threshold is used to remove variables with zero variance.
* Fclassif for feature selection.
* Adaboost as the algorithm.

The parameters of each element of the model were tuned using grid search.

As can be seen from the above results table Adaboost performed much better than logistic regression on all measures except time to build.

**Evaluation – Pipeline 2**

Pipeline 2 is made up of the same elements as Pipeline 1 with one addition, a Min/Max scaler is added to scale every variable between 0 and 1.

There is no impact of the scaler on performance other than to increase the time to build the model.

**Evaluation – Pipeline 3**

Pipeline 3 is made up of:

* Variance Threshold is used to remove variables with zero variance.
* Min/Max Scaler to scale variables between 0 and 1.
* Chi Square for feature selection.
* Logistic regression for the algorithm.

Whilst not performing at the same level as the Adaboost model. Logistic Regression still performed well on our test data. In addition, the time to build for a Logistic Regression was over 1.8 seconds faster the Adaboost model.

**Conclusion**

Following experimenting with multiple different algorithms it was clear that the Adaboost model performed significantly better than the other models that were trained on the AWID dataset. This was backed up when the final three models were used on the test data. Therefore, the pipeline 1 model is chosen as our best performing model for this project.

**Future Work**

Further work could be done to test the change in accuracy of our model combining existing selected features with predictors extracted using autoenconders. Our analysis shows that there’s a strong tendency of complex models to exhibit high variance and overfit. Ensemble based learning techniques have been tried to mitigate this issue with Adaboost showing the best results. More research could be done exploring state of the art complex classification algorithms in the area of Deep Learning in conjunction with dropout or regularization techniques to improve generalization.

This research focused on impersonation attacks. It would be interesting to extend our work to different kinds of intrusion techniques analysing to what extent the features and the algorithms selected for impersonation attacks remain effective. The analysis of the test set revealed substantial differences with the training set. This should be investigated further so that the cause of the differences can be identified and fixed. Furthermore, different search algorithms could have been explored like random search. Random search uses a random combination of hyperparameters, the algorithm takes a fraction of the time to execute than grid search as it is less computationally expensive. However, the algorithm has high variance.

Further research could be done to look into the robustness of our methodology in the face of different datasets.

**References**

[1] X. Du, H. Chen, L. Zhu, J. Li and Z. Chang, "Security and Privacy in Wireless IoT," in IEEE Wireless Communications, vol. 25, no. 6, pp. 10-11, December 2018.

[2] R.J. Little, D.B. Rubin. Statistical Analysis with Missing Data. John Wiley & Sons Inc., 2019.

[3] W. McKinney. Python for Data Analysis. O’Reilly Media Inc., 2017.

[4] C. Kolias, G. Kambrourakis, A. Stavrou, S. Gritzalis. Intrusion Detection in 802.11 Networks: Empirical Evaluation of Threats and a Public Dataset. IEEE Communication Surveys and Tutorials, vol.18, no. 1, pp 184 -186, 1st Quarter, 2016.

[5] M.E. Aminanto, R. Choi, T.C. Tanuwidjaja, P.D. Yoo, K. Kim. Deep Abstraction and Weighted Feature Selection for Wi-Fi- Impersonation Detection. IEEE Transactions on Information Forensics and Security, vol. 13, no. 3, pp 621 – 636, March 2018.

[6] IK. Yeo, R. Johnson. A new family of power transformations to improve normality or symmetry. Biometrika,  no 87, pp 954 – 959. 2000.

[7] L. Breiman, “Random Forests”, Machine Learning, 45(1), 5-32, 2001.  
<https://link.springer.com/content/pdf/10.1023/A:1010933404324.pdf>

[8] T. Chen &C. Guestrin "XGBoost: A Scalable Tree Boosting System" KDD '16: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining August 2016 Pages 785–794 <https://doi.org/10.1145/2939672.2939785>.

[9] A.V. Dorogush, V. Ershov & A. Gulin. "CatBoost: gradient boosting with categorical features support" Workshop on ML Systems at NIPS 2017.  
<http://learningsys.org/nips17/assets/papers/paper_11.pdf>

[10] I. Guyon, J. Weston, S. Barnhill, & V. Vapnik (2002). "Gene selection for cancer classification using Support Vector Machines". Machine Learning, 46(1-3), 389–422. <https://doi.org/10.1023/A:1012487302797>

[11]Liu L, [Yu Y](https://www.ncbi.nlm.nih.gov/pubmed/?term=Yu%20Y%5BAuthor%5D&cauthor=true&cauthor_uid=30463545), [Fei Z](https://www.ncbi.nlm.nih.gov/pubmed/?term=Fei%20Z%5BAuthor%5D&cauthor=true&cauthor_uid=30463545), [Li M](https://www.ncbi.nlm.nih.gov/pubmed/?term=Li%20M%5BAuthor%5D&cauthor=true&cauthor_uid=30463545), [Wu FX](https://www.ncbi.nlm.nih.gov/pubmed/?term=Wu%20FX%5BAuthor%5D&cauthor=true&cauthor_uid=30463545), [Li HD](https://www.ncbi.nlm.nih.gov/pubmed/?term=Li%20HD%5BAuthor%5D&cauthor=true&cauthor_uid=30463545), Pan Y, [Wang J](https://www.ncbi.nlm.nih.gov/pubmed/?term=Wang%20J%5BAuthor%5D&cauthor=true&cauthor_uid=30463545). An interpretable boosting model to predict side effects of analgesics for osteoarthritis. IEEE International Conference on Bioinformatics and Biomedicine 2017, pp 5-6.

[12]Tianqi Chen, Carlos Guestr XGBoost: A Scalable Tree Boosting System. University of Washington, pp 3

[13] Liudmila Prokhorenkova, Gleb Gusev, Aleksandr Vorobev, Anna Veronika Dorogush, Andrey Gulin Yandex, CatBoost: unbiased boosting with categorical features. Moscow Institute of Physics and Technology, Dolgoprudny, Russia, p

[14]L.R.Parker, Paul D Yoo, Taufiq A Asyhari, Lounis Chermak, Yoonchan Jhi, Kamal Taha. (2019) DEMISe: Interpretable Deep Extraction and Mutual Information Selection Techniques for IoT Intrusion Detection. University of Birkbeck, pp 8

[15]Yoav Freund and Robert E. Schapire. (1996) A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting. AT6T Labs, 180 Park Avenue, Florham Park, New Jersey 07932

[16] M. Claesen and B. De Moor, “Hyperparameter Search in Machine Learning,” in *MIC 2015: The XI Metaheuristics International Conference in Agadir, Morocco*, 2015.

[17] J. Bergstra and Y. Bengio, “Random Search for Hyper-Parameter Optimization Yoshua Bengio,” *Journal of Machine Learning Research*, vol. 13, pp. 281–305, Feb. 2012.

‌

[18] S. Kim, M. Son, K. Kwon, and D. Lee, “Electricity Demand Forecasting Algorithm Development Based on the Gradient Boosting Machine by Selecting the Optimal Combination of Weather Data and Stations,” *The transactions of The Korean Institute of Electrical Engineers*, vol. 68, no. 7, pp. 834–843, Jul. 2019.

‌[19]G. James, D. Witten, T. Hastie, and R. Tibshirani, *An introduction to statistical learning : with applications in R*. Springer, 2013.

[20] J.L. Hodges Jr., “The Significance Probability of the Smirnov Two-Sample Test,” Arkiv fiur Matematik, 3, No. 43 (1958), 469-86.

[21] P.Virtanen et al (2019) "SciPy 1.0–Fundamental Algorithms for Scientific Computing in Python". preprint arXiv:1907.10121 <https://arxiv.org/abs/1907.10121>

**Appendix A: Investigation of Training / Test Data Sets Mismatch – Dave Kirby**

**Introduction**

It was suspicious that models with a high accuracy (over 98%) when run with cross validation on the training set should score barely better than chance (52-54%) on the test set. Prediction is only possible if both the training set and test set are drawn from the same underlying population and have features that have been identically preprocessed so it was hypothesised that there were some difference between the training data and test data that was causing this.

A comparison of the provided training set and test set was done and it appears that for some features the values in the tests set are considerably different from the values in the training set. The data sets provided had already been scaled so that all values lay in the interval 0 to 1 so a possible cause of the discrepancies was from errors introduced by this pre-processing.

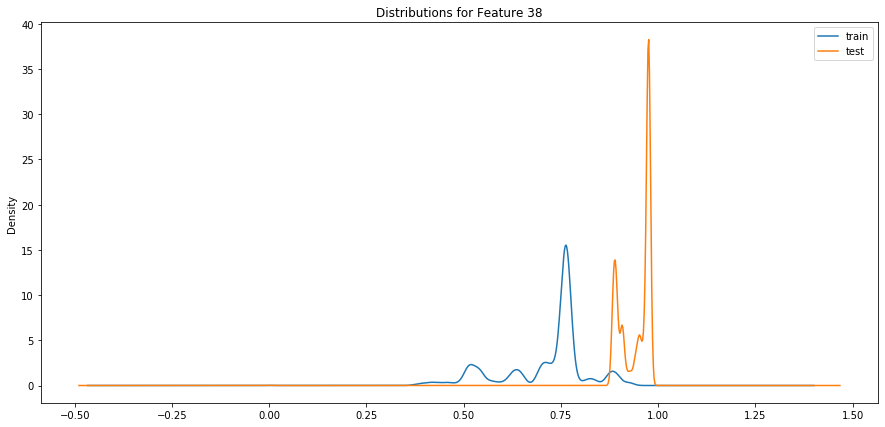
**Initial investigation - Kolmogorov-Smirnov test**

To get an overview of the problem the Kolmogorv-Smirnov test[20] was used to compare each column in the training data with the same column in the test data. The k-s statistic tests the null hypothesis that two sample sets come from the same underlying distribution. First all columns with zero variance in either the test set or training set were removed, leaving 69 columns. The ks\_2samp function from the scipy package[21] was used to compare the training data and test data for each of the remaining features. Of the 69 features 40 had a p-value at or close to zero (rejecting the null hypothesis) and 18 had a p-value at or close to 1. The remaining 11 features were somewhere in between. This gives an indication that there may be a problem but is not conclusive, particularly since the k-s test assumes that the underlying distribution is continuous.

**Direct Comparison of Single Features**

To investigate further individual features were selected so that they could be compared directly.

***Feature 38***

First to be compared was feature “38” (where “38” is the column header in the original CSV file). The distributions were compared by a kde plot for both the training and test data for the feature.  **Figure A.1**

From this plot it appears that the overall shapes are similar but shifted up on the test set. One possible reason for this could be that the data sets were wrongly scaled. You could get this effect if it the training set and test set had been scaled independently instead of the scaler being fit to the training data and applied to the test data - an outlier in the training set could push all the other values down after scaling. Unfortunately we do not have access to the original data to test this hypothesis.

**Categorical Features**

Several features were found that appeared to be categorical (they only took 2 or 3 values in both data sets) but the values for the categories were completely different. For example feature “129” has this distribution in the test set:

|  |  |
| --- | --- |
| value | Count |
| 0.0 | 38897 |
| 1.0 | 1261 |

Which seems reasonable for a 2-value categorical feature. However the training data has

|  |  |
| --- | --- |
| Value | Count |
| 0.0 | 94954 |
| 0.0000158 | 2090 |

The count ratios are similar, but the values are different by four orders magnitude.

Similar results have been found for other categorical features. For example feature “128” appears to have three categories and the training set is similarly mis-scaled:

|  |  |  |  |
| --- | --- | --- | --- |
| test set values | test set counts | training set values | training set counts |
| 0.0 | 38897 | 0.0 | 94954 |
| 0.5 | 141 | 0.0000155 | 147 |
| 1.0 | 1120 | 0.0000309 | 1943 |

Several other features were found to have similar problems, mostly in the range “108” - “129”.

**Conclusion**

It appears from the above that some of the features have been corrupted by the scaling process.  
For this reason the data sets should be considered unfit for purpose and regenerated from the original raw data if available.

**Appendix A.1: Notebook for Test Data Sets Mismatch – Dave Kirby**

test data investigation

# Rationale

It is suspicious that models with a high (>98%) accuracy on cross validation with the training set should score barely better than chance (52-54%) on the test set. From an initial comparison of the two data sets it appears that for some features the values in the tests set are considerably different from the values in the training set. Prediction is only possible if both the training set and test set are drawn from the same underlying population and have features that have been identically processed.

This notebook investigates the hypothesis that there is some systematic bias or difference between the two data sets that makes accurate prediction impossible.

In [1]:

%matplotlib inline  
  
import numpy as np  
import pandas as pd  
from scipy import stats  
from matplotlib import pyplot as plt

In [2]:

train\_data = pd.read\_csv('../datasets/train\_imperson\_without4n7\_balanced\_data.csv')  
X\_train, Y\_train = train\_data.loc[:, train\_data.columns != '155'], train\_data['155']  
  
test\_data = pd.read\_csv('../datasets/test\_imperson\_without4n7\_balanced\_data.csv')  
X\_test, Y\_test = test\_data.loc[:, test\_data.columns != '155'], test\_data['155']

Get the columns with no variance and check they are the same for both data sets.

In [3]:

train\_no\_var = X\_train.var() == 0.0  
test\_no\_var = X\_test.var() == 0.0  
all(train\_no\_var == test\_no\_var)

Out[3]:

False

OK, so the columns in the training set with no variance are different from the columns in the test set with no variance. Let investigate them further...

Lets see what the values are in these columns:

In [6]:

{  
 col: list(X\_train[col].value\_counts().items())   
 for col in X\_train[X\_train.columns[train\_no\_var != test\_no\_var]].columns  
}

Out[6]:

{'69': [(0, 97044)],  
 '81': [(0, 97044)],  
 '83': [(0, 97042), (1, 2)],  
 '84': [(0, 97038), (1, 6)],  
 '86': [(0, 97036), (1, 8)],  
 '88': [(0.0, 97038), (0.03125, 3), (0.46875, 2), (0.375, 1)],  
 '97': [(0, 97043), (1, 1)],  
 '113': [(0.0, 97041), (0.42857, 3)],  
 '117': [(0.0, 97036),  
 (0.0034351, 3),  
 (0.15611, 2),  
 (0.11412, 1),  
 (0.097328, 1),  
 (0.096183, 1)],  
 '133': [(0, 97043), (1, 1)],  
 '138': [(0.5, 97043), (0.88, 1)],  
 '148': [(0, 97044)]}

In [7]:

{col: list(X\_test[col].value\_counts().items())   
 for col in X\_test[X\_test.columns[train\_no\_var != test\_no\_var]].columns  
}

Out[7]:

{'69': [(0, 21552), (1, 18606)],  
 '81': [(0.0, 27711),  
 (0.090909, 6243),  
 (0.18182, 6181),  
 (0.27273000000000003, 23)],  
 '83': [(0, 40158)],  
 '84': [(0, 40158)],  
 '86': [(0, 40158)],  
 '88': [(0, 40158)],  
 '97': [(0, 40158)],  
 '113': [(0, 40158)],  
 '117': [(0, 40158)],  
 '133': [(0, 40158)],  
 '138': [(0, 40158)],  
 '148': [(0, 40157), (1, 1)]}

From the above we can see that there are 9 features that have variance in the training set but no variance in test set, and three features the other way round. For most of the cases the columns with variance only have a single value or handful of values that differ, so are unlikely to have much effect on any model trained on the data.

We shall exclude all these features from further investigation, along with the features that have no variance for either data set.

In [8]:

X\_train\_with\_var = X\_train[X\_train.columns[~(train\_no\_var | test\_no\_var)]]  
X\_test\_with\_var = X\_test[X\_test.columns[~(train\_no\_var | test\_no\_var)]]  
  
X\_train\_with\_var.shape, X\_test\_with\_var.shape

Out[8]:

((97044, 69), (40158, 69))

By eliminating features that have no variance in either the training or test set (or both) we are left with 69 features.

We now want to test for each feature in the remaining data sets whether the feature in the test set comes from the same distribution as the same feature in the training set. We shall use the ks2\_samp function to determine this.

In [9]:

ks\_pvalues = {col: stats.ks\_2samp(X\_train[col], X\_test[col]).pvalue   
 for col in X\_train\_with\_var.columns  
}  
ks\_pvalues

Out[9]:

{'5': 0.0,  
 '6': 0.0,  
 '8': 0.0,  
 '9': 0.0,  
 '14': 1.0,  
 '15': 1.0,  
 '16': 1.0,  
 '18': 1.0,  
 '20': 1.0,  
 '26': 1.0,  
 '29': 1.0,  
 '38': 0.0,  
 '43': 1.0,  
 '47': 3.392828255901444e-109,  
 '48': 1.0,  
 '50': 1.5897695358746193e-89,  
 '51': 2.7909260802022505e-89,  
 '52': 1.0,  
 '61': 0.0,  
 '62': 1.0,  
 '64': 6.6900579035861e-132,  
 '66': 5.749325310483437e-91,  
 '67': 2.839482114752711e-64,  
 '68': 3.849391965991644e-49,  
 '70': 6.39437325347434e-41,  
 '71': 0.0,  
 '72': 0.9954440728272881,  
 '73': 2.3606093166691013e-53,  
 '75': 0.0,  
 '76': 0.0,  
 '77': 0.0,  
 '78': 0.0,  
 '79': 0.0,  
 '80': 0.0,  
 '82': 0.0,  
 '89': 1.0,  
 '90': 0.34532006221886347,  
 '93': 2.2110364904425522e-20,  
 '94': 0.9999999999999997,  
 '98': 0.4136946783683769,  
 '104': 0.8779383085900594,  
 '105': 1.0,  
 '106': 1.0,  
 '107': 3.1904029047477344e-09,  
 '108': 1.2884668003640035e-210,  
 '109': 0.800725788083745,  
 '110': 1.4882346123275508e-94,  
 '111': 1.0,  
 '112': 0.4779519542597044,  
 '118': 0.434667546199392,  
 '119': 2.1024335925785942e-200,  
 '120': 3.862206790140662e-05,  
 '121': 0.10504759448613488,  
 '122': 8.664540788959877e-58,  
 '123': 1.0,  
 '125': 0.5270577097126901,  
 '126': 8.702217081928747e-25,  
 '127': 8.702217081928747e-25,  
 '128': 8.702217081928747e-25,  
 '129': 8.702217081928747e-25,  
 '130': 0.007887321380490302,  
 '140': 0.0,  
 '141': 0.0,  
 '142': 0.0,  
 '143': 0.09314326497438503,  
 '144': 1.124235144809374e-16,  
 '145': 0.9870479857004859,  
 '146': 0.9870479857004859,  
 '154': 0.0}

In [10]:

len([x for x in ks\_pvalues.values() if x < 0.05])

Out[10]:

40

In [11]:

from matplotlib import pyplot as plt  
\_, ax = plt.subplots()  
ax.set\_xlabel("Feature")  
ax.set\_ylabel("p-value")  
vals = pd.Series(sorted(ks\_pvalues.values()))  
vals.plot(figsize=(10,5), title="Kolmogorov-Smirnov p-values for each feature")

Out[11]:

<matplotlib.axes.\_subplots.AxesSubplot at 0x27caacb9cf8>

![](data:image/png;base64;base64,)

High p-values mean that column probably comes from the same distribution, low pvalues mean they probably come from a different distribution, at least as far as the Kolmogorov-Smirnov test can tell. From the above you can see that 40 of the 69 features have a p-value of less than 0.05 and many of them have a p-value of zero.

Lets focus on the features that our team identified as being the most important for training the model. We found that just six features could be used to create a model that scored in excess of 98% in cross validation on the training data, but barely more than 52% on the test data. It turns out that all these features score close to zero p-value on the KS test:

In [12]:

best\_features = ['8', '38', '67', '76', '78', '119']  
[(f, ks\_pvalues[f]) for f in best\_features]

Out[12]:

[('8', 0.0),  
 ('38', 0.0),  
 ('67', 2.839482114752711e-64),  
 ('76', 0.0),  
 ('78', 0.0),  
 ('119', 2.1024335925785942e-200)]

In [13]:

X\_train[best\_features].describe()

Out[13]:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 8 | 38 | 67 | 76 | 78 | 119 |
| count | 97044.000000 | 97044.000000 | 97044.000000 | 97044.000000 | 97044.000000 | 97044.000000 |
| mean | 0.193837 | 0.713756 | 0.376979 | 0.003294 | 0.002077 | 0.000336 |
| std | 0.354444 | 0.109809 | 0.395372 | 0.021005 | 0.017849 | 0.011518 |
| min | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 |
| 25% | 0.037908 | 0.654075 | 0.000000 | 0.000563 | 0.001731 | 0.000000 |
| 50% | 0.037908 | 0.757210 | 0.307690 | 0.000563 | 0.001731 | 0.000000 |
| 75% | 0.054902 | 0.767550 | 0.615380 | 0.002253 | 0.001731 | 0.000000 |
| max | 1.000000 | 0.933540 | 1.000000 | 0.990990 | 0.995670 | 0.994750 |

In [14]:

X\_test[best\_features].describe()

Out[14]:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 8 | 38 | 67 | 76 | 78 | 119 |
| count | 40158.000000 | 40158.000000 | 40158.000000 | 40158.000000 | 40158.000000 | 40158.000000 |
| mean | 0.200639 | 0.944178 | 0.341793 | 0.021087 | 0.003357 | 0.001407 |
| std | 0.371463 | 0.040206 | 0.385898 | 0.037241 | 0.007230 | 0.005267 |
| min | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 |
| 25% | 0.016993 | 0.905910 | 0.000000 | 0.005291 | 0.002146 | 0.000000 |
| 50% | 0.016993 | 0.967410 | 0.000000 | 0.005291 | 0.002146 | 0.000000 |
| 75% | 0.060131 | 0.975510 | 0.615380 | 0.026455 | 0.002146 | 0.000000 |
| max | 1.000000 | 0.978310 | 1.000000 | 0.835980 | 0.963520 | 0.161700 |

Lets take a closer look at one of the features in particular that show a large discrepancy.  
We can compare the training and testing version of feature 38 by plotting them on a histogram. I am using a log scale, otherwise the smaller bins will not show up on the plot.

In [15]:

pd.DataFrame({'train':X\_train['38'], 'test':X\_test['38']}).hist(log=True, bins=20)

Out[15]:

array([[<matplotlib.axes.\_subplots.AxesSubplot object at 0x0000027CAAD90668>,  
 <matplotlib.axes.\_subplots.AxesSubplot object at 0x0000027CAAD5EE10>]],  
 dtype=object)

![](data:image/png;base64;base64,)

From the above histograms you can clearly see that the training data has a wide range of values and has most values between 0.5 and 0.8. The training data however has all its values up near 1.0 with a handful at zero. This is extremely unlikely if both data sets are identically and independantly drawn from the same population.

You can see the shape of the distributions more clearly on a KDE plot:

In [16]:

pd.DataFrame({'train':X\_train['38'], 'test':X\_test['38']}).plot.kde(figsize=(15,7), title="Distributions for Feature 38")

Out[16]:

<matplotlib.axes.\_subplots.AxesSubplot at 0x27cc1f59198>

![](data:image/png;base64;base64,)

From this plot it appears that the overall shape is similar but shifted up on the test set. One possible reason for this could be that the data sets were wrongly scaled. You could get this effect if it the training set and test set had been scaled independently instead of the scaler being fit to the training data and applied to the test data - an outlier in the training set could push all the other values down after scaling. Unfortunately all the data had been scaled to be between 0 and 1 before we got it and we do not have access to the original data to test this hypothesis.

Another indicator that this could be the case is that the maximum value of all features in the test set is never more than 1.0. If a scaler had been fit to the training set and then applied to the test set it would be likely that a few features in the test set would have a maximum value greater than 1.0 (unless the features were all categorical).

There are also several features that appear to be categorical with two or three categories, but the categories have been scaled to completely different values:

In [17]:

for col in ['128', '129', '122']:  
 print(f"Training data feature {col} (value, count)")  
 print(list(X\_train[col].value\_counts().items()))  
 print(f"Test data feature {col} (value, count)")  
 print(list(X\_test[col].value\_counts().items()))  
 print()

Training data feature 128 (value, count)  
[(0.0, 94954), (3.09e-05, 1943), (1.55e-05, 147)]  
Test data feature 128 (value, count)  
[(0.0, 38897), (1.0, 1120), (0.5, 141)]  
  
Training data feature 129 (value, count)  
[(0.0, 94954), (1.58e-05, 2090)]  
Test data feature 129 (value, count)  
[(0, 38897), (1, 1261)]  
  
Training data feature 122 (value, count)  
[(0.0, 93530), (0.0039683, 2571), (0.007936499999999999, 943)]  
Test data feature 122 (value, count)  
[(0.0, 38223), (0.33333, 1423), (0.66667, 512)]

From the above we see that for example feature 128 has three distinct values for both test and training data, but for the training data they take the values 0.0, 0.000015 and 0.00003, while for the test data they take the values 0.0, 0.5 and 1.0. It is the same for features 129 and 122 (and probably others) - there are orders of magnitude difference in the values taken for the categories.

We can get an idea of which features suffer from this problem by looking for features in the test set where the maximum value is more than twice that of the maximum value in the training set, or vice versa:

In [18]:

train\_max = X\_train\_with\_var.max()  
test\_max = X\_test\_with\_var.max()  
bad\_max = (train\_max < test\_max/2) | (test\_max < train\_max/2)  
  
pd.DataFrame({'train\_max': train\_max[bad\_max], 'test\_max': test\_max[bad\_max]})

Out[18]:

|  |  |  |
| --- | --- | --- |
|  | train\_max | test\_max |
| 80 | 0.539070 | 0.040964 |
| 104 | 1.000000 | 0.100000 |
| 108 | 0.001530 | 1.000000 |
| 110 | 0.000125 | 0.750000 |
| 119 | 0.994750 | 0.161700 |
| 121 | 0.004566 | 0.500000 |
| 122 | 0.007936 | 0.666670 |
| 125 | 0.333330 | 1.000000 |
| 126 | 0.000015 | 1.000000 |
| 127 | 0.015686 | 0.500000 |
| 128 | 0.000031 | 1.000000 |
| 129 | 0.000016 | 1.000000 |
| 144 | 0.865560 | 0.390870 |

# Conclusion

It is clear that there are serious discrepancies between the data in the test set and the data in the training set. This will cause serious problems for anyone trying to build a model using the training data and then validate it against the test set.

**Appendix B** **code**:

**Pre-processing code**

*# Initial modules*

**import** **numpy** **as** **np**

**import** **pandas** **as** **pd**

**import** **sklearn**

*#load data*

mdata= pd.read\_csv('train\_imperson\_without4n7\_balanced\_data.csv')

mdata.head()

### **Investigating the presence of missing values**

mdata.info()

*#data is complete*

m1= mdata.dropna()

m1.info()

### **Investigating the presence of duplicate cases**

*#We should assume data are iid, hence keep duplicates*

m2 = mdata.drop\_duplicates()

m2.info()

*#duplicate rows for reference*

mdata[mdata.duplicated() == **True**].head()

### **Descriptive statistics on the training set**

mdata.describe().T.describe()

*# Separate X and Y*

X, Y = mdata.loc[:, mdata.columns != '155'], mdata['155']

X.head()

### **Statistical analysis of the features matrix**

X.describe().T.describe()

X.std().value\_counts()

### **Dataset 1: features with variance greater than zero**

*#Xselected with 0 var taken out*

**from** **sklearn.feature\_selection** **import** VarianceThreshold

selector = VarianceThreshold()

selector.fit(X)

col = X.columns[selector.get\_support()]

Xselected= X.loc[:, col ]

Xselected.head()

Xselected.shape

Xselected.skew().median()

### **Dataset 2: variance greater than zero and data scaled to a range [0, 1]**

*#normalised range 0 1*

**from** **sklearn.preprocessing** **import** MinMaxScaler

scaler1 = MinMaxScaler().fit(Xselected)

scaled0\_1=scaler1.transform(Xselected)

*# dataframe normalised range 0\_1*

NX3df = pd.DataFrame(scaled0\_1, columns= Xselected.columns)

NX3df.head()

### **Dataset 3: variance greater than zero and power transform**

*#Power transform*

**from** **sklearn.preprocessing** **import** PowerTransformer

pt = PowerTransformer(method = 'yeo-johnson').fit(Xselected)

poweredX2 = pt.transform(Xselected)

*#dataframe power gaussian*

PX2df = pd.DataFrame(poweredX2, columns= Xselected.columns)

PX2df.head()

PX2df.skew().median()

### **Dataset 4: variance greater than zero and norm one**

*#Normalizer transforms to norm 1*

**from** **sklearn.preprocessing** **import** Normalizer

scaler = Normalizer().fit(Xselected)

normalizedX2 = scaler.transform(Xselected)

*#dataframe with norm 1*

NX2df = pd.DataFrame(normalizedX2, columns = Xselected.columns)

NX2df.head()

**Datasets**

Reduced dataset (down to 78 features) by removing zero variance features - dataset 1

* Xselected

Dataset 1 + rescaling each feature between 0 to 1 - dataset 2

* NX3df

Dataset 1 + transform the datset to be gaussian like (mean=0 and s.d=1) - dataset 3

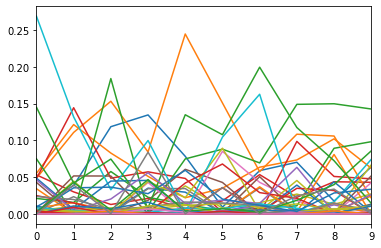
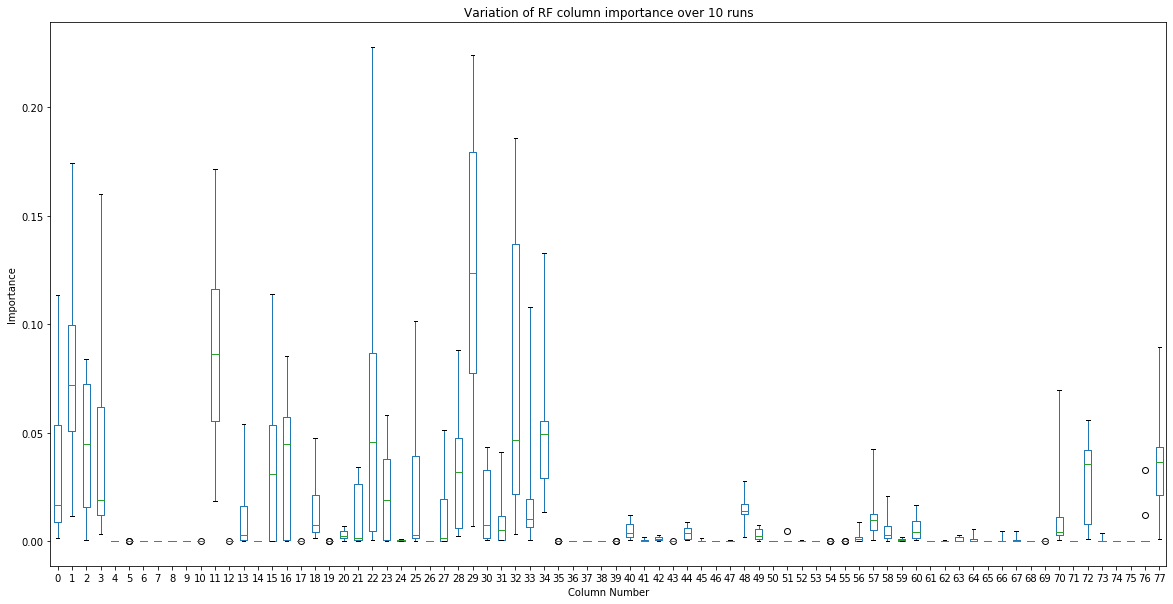
Dataset 1 + normalised the dataset by rescaling each row to add up to length one - dataset 4

* NX2df
* **Feature selection Part 1: Tree based feature importance**
* %matplotlib inline
* **import** **numpy** **as** **np**
* **import** **pandas** **as** **pd**
* mdata= pd.read\_csv('train\_imperson\_without4n7\_balanced\_data.csv')
* X, Y = mdata.loc[:, mdata.columns != '155'], mdata['155']
* *#eliminate features with 0 standard deviation*
* X1\_sigma = X.loc[:, (X.std()>0)]

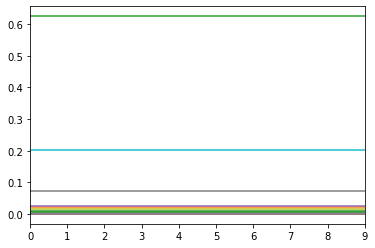
## Use Random Forest to get a ranking of the most important features

* %%time
* **from** **sklearn.ensemble** **import** RandomForestClassifier
* forest = RandomForestClassifier()
* forest.fit(X1\_sigma, Y)
* CPU times: user 1.1 s, sys: 120 ms, total: 1.23 s
* Wall time: 1.22 s
* RandomForestClassifier(bootstrap=True, class\_weight=None, criterion='gini',
* max\_depth=None, max\_features='auto', max\_leaf\_nodes=None,
* min\_impurity\_decrease=0.0, min\_impurity\_split=None,
* min\_samples\_leaf=1, min\_samples\_split=2,
* min\_weight\_fraction\_leaf=0.0, n\_estimators=10,
* n\_jobs=None, oob\_score=False, random\_state=None,
* verbose=0, warm\_start=False)
* rf\_importance = forest.feature\_importances\_
* rf\_importance
* array([1.21773936e-01, 6.74576184e-02, 1.27533858e-02, 3.25055934e-02,
* 0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00,
* 0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 1.12717330e-01,
* 0.00000000e+00, 5.11327629e-02, 1.88626154e-07, 1.28414849e-04,
* 4.23592171e-02, 0.00000000e+00, 1.14271398e-02, 0.00000000e+00,
* 2.17936612e-03, 1.49888873e-03, 1.21108806e-01, 4.33054528e-02,
* 1.67185815e-02, 5.45721778e-02, 1.93384693e-05, 1.91449751e-05,
* 4.85905652e-05, 1.13374263e-01, 2.76903295e-03, 7.75254640e-04,
* 5.86678622e-02, 2.31780047e-02, 2.37023777e-02, 0.00000000e+00,
* 0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00,
* 4.89837766e-05, 2.41471259e-03, 2.30145000e-05, 0.00000000e+00,
* 3.00601885e-04, 0.00000000e+00, 4.06378753e-06, 1.23660008e-06,
* 5.57234463e-03, 3.23144334e-03, 3.45114464e-05, 1.83193709e-05,
* 6.93238980e-05, 1.43063228e-04, 0.00000000e+00, 0.00000000e+00,
* 4.34973477e-03, 5.90496637e-03, 3.41019089e-03, 3.14387779e-10,
* 2.44441954e-03, 7.22502104e-06, 0.00000000e+00, 7.22399391e-03,
* 4.47292321e-06, 0.00000000e+00, 8.27123091e-06, 1.69551549e-05,
* 3.38403706e-06, 0.00000000e+00, 3.12374815e-02, 2.88408058e-05,
* 1.75234928e-03, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00,
* 2.38495566e-04, 1.73148707e-02])

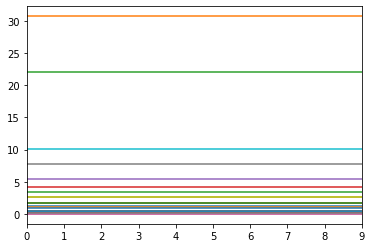
# Rank the features by importance

* rf\_ranked = sorted((rating, idx) **for** (idx, rating) **in** enumerate(rf\_importance))
* *# look at the 20 most important features*
* rf\_ranked[-20:]
* [(0.0059049663711367014, 57),
* (0.0072239939111368974, 63),
* (0.011427139812082803, 18),
* (0.012753385828900028, 2),
* (0.016718581453701524, 24),
* (0.017314870656399663, 77),
* (0.023178004703063564, 33),
* (0.02370237766100005, 34),
* (0.031237481500813548, 70),
* (0.03250559343416152, 3),
* (0.04235921705908447, 16),
* (0.04330545279612031, 23),
* (0.05457217784122165, 25),
* (0.05866786221888163, 32),
* (0.06745761842879369, 1),
* (0.11271733041699825, 11),
* (0.11337426256843093, 29),
* (0.12110880583459076, 22),
* (0.12177393604844935, 0)]
* I notice the rating are not stable - if you run the last three code cells multiple times the order and rating change quite a bit. Lets try running it a few times and plot the importance it assigns to each feature.
* importances = []
* **for** \_ **in** range(10):
* forest.fit(X1\_sigma, Y)
* importances.append(forest.feature\_importances\_)
* df = pd.DataFrame(importances)
* We can see how much the feature importance varies by plotting the importance for each feature across ten runs:
* df.plot(legend=**False**)
* 
* Another way that makes the variation clearer is to plot how much each of the feature importances varies using a boxplot:
* **from** **matplotlib** **import** pyplot **as** plt
* fig, ax = plt.subplots(figsize=(20,10))
* ax.set\_title("Variation of RF column importance over 10 runs")
* ax.set\_xlabel("Column Number")
* ax.set\_ylabel("Importance")
* df.boxplot(grid=**False**, whis=1000., ax=ax)
* 
* Clearly some features have very little importance but for the remainder the relative importance varies hugely from run to run due to the random nature of Random Forests.

# Using xgboost to get the feature importance

* %%time
* **from** **xgboost** **import** XGBClassifier
* xgb = XGBClassifier()
* xgb.fit(X1\_sigma, Y)
* xg\_importance = xgb.feature\_importances\_
* xg\_importance
* array([4.7329403e-03, 0.0000000e+00, 8.9810267e-03, 0.0000000e+00,
* 0.0000000e+00, 0.0000000e+00, 0.0000000e+00, 0.0000000e+00,
* 0.0000000e+00, 0.0000000e+00, 0.0000000e+00, 2.1276930e-02,
* 0.0000000e+00, 1.7434136e-03, 0.0000000e+00, 0.0000000e+00,
* 0.0000000e+00, 0.0000000e+00, 3.0411058e-03, 0.0000000e+00,
* 2.4640295e-03, 0.0000000e+00, 6.2512726e-01, 0.0000000e+00,
* 0.0000000e+00, 5.1427062e-04, 0.0000000e+00, 0.0000000e+00,
* 7.4304739e-04, 2.0335531e-01, 4.0172166e-04, 4.7231920e-04,
* 1.4258807e-03, 0.0000000e+00, 2.3888556e-02, 0.0000000e+00,
* 0.0000000e+00, 0.0000000e+00, 0.0000000e+00, 0.0000000e+00,
* 0.0000000e+00, 0.0000000e+00, 0.0000000e+00, 0.0000000e+00,
* 1.9602317e-03, 0.0000000e+00, 0.0000000e+00, 0.0000000e+00,
* 1.5265194e-02, 1.6785092e-03, 0.0000000e+00, 0.0000000e+00,
* 1.0695638e-03, 2.6311394e-04, 0.0000000e+00, 0.0000000e+00,
* 2.8898872e-03, 7.1290687e-02, 0.0000000e+00, 0.0000000e+00,
* 0.0000000e+00, 0.0000000e+00, 0.0000000e+00, 0.0000000e+00,
* 0.0000000e+00, 0.0000000e+00, 0.0000000e+00, 0.0000000e+00,
* 0.0000000e+00, 0.0000000e+00, 0.0000000e+00, 0.0000000e+00,
* 7.4149747e-03, 0.0000000e+00, 0.0000000e+00, 0.0000000e+00,
* 0.0000000e+00, 0.0000000e+00], dtype=float32)
* Notes: xgboost normalises the importance values so that they sum up to 1. Also most of the features have an importance of zero and are completely rejected by XGB.
* xg\_ranked = sorted((rating, idx) **for** (idx, rating) **in** enumerate(importance))
* *# look at the 20 most important features*
* xg\_ranked[-20:]
* [(0.0004723192, 31),
* (0.0005142706, 25),
* (0.0007430474, 28),
* (0.0010695638, 52),
* (0.0014258807, 32),
* (0.0016785092, 49),
* (0.0017434136, 13),
* (0.0019602317, 44),
* (0.0024640295, 20),
* (0.0028898872, 56),
* (0.0030411058, 18),
* (0.0047329403, 0),
* (0.0074149747, 72),
* (0.008981027, 2),
* (0.015265194, 48),
* (0.02127693, 11),
* (0.023888556, 34),
* (0.07129069, 57),
* (0.20335531, 29),
* (0.62512726, 22)]
* np.array([x[0] **for** x **in** reversed(xg\_ranked)]).cumsum()
* array([0.62512726, 0.82848257, 0.89977324, 0.92366177, 0.9449387 ,
* 0.9602039 , 0.96918494, 0.97659993, 0.9813329 , 0.984374 ,
* 0.98726386, 0.9897279 , 0.99168813, 0.99343157, 0.9951101 ,
* 0.99653596, 0.9976055 , 0.99834853, 0.9988628 , 0.9993351 ,
* 0.99973685, 0.99999994, 0.99999994, 0.99999994, 0.99999994,
* 0.99999994, 0.99999994, 0.99999994, 0.99999994, 0.99999994,
* 0.99999994, 0.99999994, 0.99999994, 0.99999994, 0.99999994,
* 0.99999994, 0.99999994, 0.99999994, 0.99999994, 0.99999994,
* 0.99999994, 0.99999994, 0.99999994, 0.99999994, 0.99999994,
* 0.99999994, 0.99999994, 0.99999994, 0.99999994, 0.99999994,
* 0.99999994, 0.99999994, 0.99999994, 0.99999994, 0.99999994,
* 0.99999994, 0.99999994, 0.99999994, 0.99999994, 0.99999994,
* 0.99999994, 0.99999994, 0.99999994, 0.99999994, 0.99999994,
* 0.99999994, 0.99999994, 0.99999994, 0.99999994, 0.99999994,
* 0.99999994, 0.99999994, 0.99999994, 0.99999994, 0.99999994,
* 0.99999994, 0.99999994, 0.99999994], dtype=float32)
* importances = []
* **for** \_ **in** range(10):
* xgb = XGBClassifier()
* xgb.fit(X1\_sigma, Y)
* importances.append(xgb.feature\_importances\_)
* df = pd.DataFrame(importances)
* df.plot(legend=**False**)
* 
* Clearly XGBoost is much more consistent and opinionated.

# Feature Importance with Catboost

* **from** **catboost** **import** CatBoostClassifier
* cb = CatBoostClassifier()
* %%time
* cb.fit(X1\_sigma, Y, verbose=100)
* Learning rate set to 0.061156 0: learn: 0.5237915 total: 30.2ms remaining: 30.2s 100: learn: 0.0003742 total: 2.6s remaining: 23.1s 200: learn: 0.0002458 total: 4.95s remaining: 19.7s 300: learn: 0.0002417 total: 7.21s remaining: 16.7s 400: learn: 0.0002413 total: 9.33s remaining: 13.9s 500: learn: 0.0002413 total: 11.4s remaining: 11.4s 600: learn: 0.0002413 total: 13.5s remaining: 8.96s 700: learn: 0.0002413 total: 15.6s remaining: 6.65s 800: learn: 0.0002413 total: 17.7s remaining: 4.39s 900: learn: 0.0002413 total: 19.8s remaining: 2.17s 999: learn: 0.0002413 total: 21.8s remaining: 0us CPU times: user 2min 45s, sys: 6.44 s, total: 2min 51s Wall time: 23.1 s
* cb\_importance = cb.get\_feature\_importance()
* cb\_importance
* array([1.18430010e+00, 1.14742527e+00, 3.39528732e+00, 4.21509614e+00,
* 0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00,
* 0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 3.07372845e+01,
* 0.00000000e+00, 4.99256200e-02, 0.00000000e+00, 2.80171153e-02,
* 1.50692376e-02, 0.00000000e+00, 2.66639475e+00, 0.00000000e+00,
* 4.67656710e-01, 0.00000000e+00, 2.21440580e+01, 0.00000000e+00,
* 0.00000000e+00, 0.00000000e+00, 8.43731181e-05, 0.00000000e+00,
* 3.10907694e-02, 1.00211719e+01, 8.59017031e-01, 4.30230428e-01,
* 1.76564068e+00, 1.76390195e+00, 5.38597299e+00, 0.00000000e+00,
* 0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00,
* 0.00000000e+00, 7.68228173e-02, 1.73180901e-01, 0.00000000e+00,
* 3.41893686e-02, 0.00000000e+00, 0.00000000e+00, 1.05503739e-04,
* 2.69208110e+00, 2.57220819e-01, 1.33463605e-02, 0.00000000e+00,
* 3.46336534e-02, 8.35212037e-02, 0.00000000e+00, 0.00000000e+00,
* 0.00000000e+00, 7.83041873e+00, 1.25920954e-02, 0.00000000e+00,
* 0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00,
* 0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00,
* 0.00000000e+00, 0.00000000e+00, 5.08903431e-01, 0.00000000e+00,
* 1.77428343e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00,
* 0.00000000e+00, 2.01075640e-01])
* cb\_ranked = sorted((rating, idx) **for** (idx, rating) **in** enumerate(cb\_importance))
* *# look at the 20 most important features*
* cb\_ranked[-20:]
* [(0.20107563990257385, 77),
* (0.2572208193324662, 49),
* (0.4302304280925102, 31),
* (0.46765671017600735, 20),
* (0.5089034308958107, 70),
* (0.8590170314474369, 30),
* (1.1474252661338882, 1),
* (1.184300100425819, 0),
* (1.763901945568504, 33),
* (1.7656406752657117, 32),
* (1.77428342770122, 72),
* (2.6663947507615036, 18),
* (2.6920810952842076, 48),
* (3.3952873186498995, 2),
* (4.2150961419053825, 3),
* (5.385972985838783, 34),
* (7.830418729096104, 57),
* (10.021171940679691, 29),
* (22.14405802336696, 22),
* (30.7372845201552, 11)]
* N.B. The feature importances from CatBoost are weighted to sum to 100 instead of 1.
* importances = []
* **for** \_ **in** range(10):
* cb = CatBoostClassifier()
* cb.fit(X1\_sigma, Y, verbose=500)
* importances.append(cb.get\_feature\_importance())
* df = pd.DataFrame(importances)
* df.plot(legend=**False**)
* 
* As we can see, CatBoost is as consistent as XGBoost, although it gives somewhat different weights to the features.
* np.array([x[0] **for** x **in** reversed(cb\_ranked)]).cumsum()
* array([0.62512726, 0.82848257, 0.89977324, 0.92366177, 0.9449387 , 0.9602039 , 0.96918494, 0.97659993, 0.9813329 , 0.984374 , 0.98726386, 0.9897279 , 0.99168813, 0.99343157, 0.9951101 , 0.99653596, 0.9976055 , 0.99834853, 0.9988628 , 0.9993351 , 0.99973685, 0.99999994, 0.99999994, 0.99999994, 0.99999994, 0.99999994, 0.99999994, 0.99999994, 0.99999994, 0.99999994, 0.99999994, 0.99999994, 0.99999994, 0.99999994, 0.99999994, 0.99999994, 0.99999994, 0.99999994, 0.99999994, 0.99999994, 0.99999994, 0.99999994, 0.99999994, 0.99999994, 0.99999994, 0.99999994, 0.99999994, 0.99999994, 0.99999994, 0.99999994, 0.99999994, 0.99999994, 0.99999994, 0.99999994, 0.99999994, 0.99999994, 0.99999994, 0.99999994, 0.99999994, 0.99999994, 0.99999994, 0.99999994, 0.99999994, 0.99999994, 0.99999994, 0.99999994, 0.99999994, 0.99999994, 0.99999994, 0.99999994, 0.99999994, 0.99999994, 0.99999994, 0.99999994, 0.99999994, 0.99999994, 0.99999994, 0.99999994], dtype=float32)
* **def** top20(ranked):
* **return** [x[1] **for** x **in** ranked[-20:]]
* print("RF:", top20(rf\_ranked))
* print("xgboost:", top20(xg\_ranked))
* print("catboost:", top20(cb\_ranked))
* print("Intersection:", set(top20(rf\_ranked)).intersection(top20(xg\_ranked)).intersection(top20(cb\_ranked)))
* RF: [57, 63, 18, 2, 24, 77, 33, 34, 70, 3, 16, 23, 13, 25, 32, 1, 11, 29, 22, 0]
* xgboost: [31, 25, 28, 52, 32, 49, 13, 44, 20, 56, 18, 0, 72, 2, 48, 11, 34, 57, 29, 22]
* catboost: [77, 49, 31, 20, 70, 30, 1, 0, 33, 32, 72, 18, 48, 2, 3, 34, 57, 29, 22, 11]
* Intersection: {0, 32, 2, 34, 11, 18, 22, 57, 29}
* selected\_features = set(top20(xg\_ranked)).intersection(top20(cb\_ranked))
* print("XGBoost and CatBoost Intersection:", selected\_features)
* XGBoost and CatBoost Intersection: {0, 32, 2, 34, 72, 11, 48, 49, 18, 20, 22, 57, 29, 31}

## Since the Random Forest feature importance rating is so unstable, for the next section we will use the features that both CatBoost and XGBoost both consider important

# Evaluating the models on the subset of features

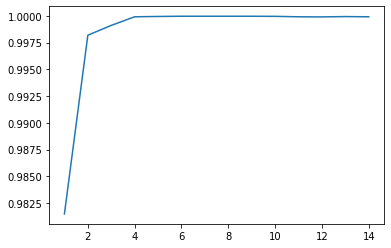
* **from** **sklearn.model\_selection** **import** cross\_validate
* X\_top\_features = X1\_sigma.iloc[:, list(selected\_features)]
* *# shuffle the rows*
* **from** **numpy** **import** random, arange
* indexes = arange(Y.shape[0])
* numpy.random.shuffle(indexes)
* X\_cv = X\_top\_features.iloc[indexes,:]
* Y\_cv = Y.iloc[indexes]
* rf = RandomForestClassifier()
* rf\_cv = cross\_validate(rf, X\_cv, Y\_cv, cv=5, scoring="accuracy")
* rf\_cv
* {'fit\_time': array([0.46078181, 0.41081405, 0.42314792, 0.42004848, 0.42713404]),
* 'score\_time': array([0.01698518, 0.01703095, 0.01725793, 0.01659465, 0.01717067]),
* 'test\_score': array([0.99989696, 0.99994848, 0.99963932, 0.99994847, 0.9997939 ])}
* gb = XGBClassifier(verbosity=0)
* gb\_cv = cross\_validate(gb, X\_cv, Y\_cv, cv=5, scoring="accuracy")
* gb\_cv
* {'fit\_time': array([4.75142622, 4.72674131, 4.71258163, 4.68719816, 4.67803741]),
* 'score\_time': array([0.06090856, 0.0629797 , 0.06166697, 0.06219172, 0.06117797]),
* 'test\_score': array([0.99984544, 0.99989696, 0.9997939 , 1. , 0.99989695])}
* cb = CatBoostClassifier(verbose=**False**, early\_stopping\_rounds=50)
* cb\_cv = cross\_validate(cb, X\_cv, Y\_cv, cv=5, scoring="accuracy")
* cb\_cv

## Conclusion

* We have reduced the feature set from 78 down to 14 by selecting those features that both XGBoost and CatBoost ranked as the most important. In 5-fold cross validation tests Random Forest, XGBoost and CatBoost all scored more than 99.9% accuracy, with CatBoost scoring highest, including getting 100% on three of the five folds. However CatBoost was the slowest to train (~19s compared with ~4.5s for both RF and XGB). On the other hand it was the fastest for prediction.

# Feature Selection Part 2: Recursive Feature Elimination

## Using the scikit-learn RFECV to find and validate the best combination of features[¶](https://render.githubusercontent.com/view/ipynb?commit=1212225a0b30d0af437432705949bb84cd1a9abf&enc_url=68747470733a2f2f7261772e67697468756275736572636f6e74656e742e636f6d2f646176656b697262792f414d4c5f70726f6a6563742f313231323232356130623330643061663433373433323730353934396262383463643161396162662f6e6f7465626f6f6b732f66696e616c5f666561747572655f73656c656374696f6e2e6970796e623f746f6b656e3d4141494f514e535537554f444550535933554b41495932364449463645&nwo=davekirby%2FAML_project&path=notebooks%2Ffinal_feature_selection.ipynb&repository_id=221760523&repository_type=Repository#Using-the-scikit-learn-RFECV-to-find-and-validate-the-best-combination-of-features)

* %matplotlib inline
* **import** **numpy** **as** **np**
* **import** **pandas** **as** **pd**
* mdata= pd.read\_csv('train\_imperson\_without4n7\_balanced\_data.csv')
* X, Y = mdata.loc[:, mdata.columns != '155'], mdata['155']
* *#eliminate features with 0 standard deviation*
* X1\_sigma = X.loc[:, (X.std()>0)]
* **from** **catboost** **import** CatBoostClassifier
* **from** **sklearn.feature\_selection** **import** RFECV
* initial\_selection = {0, 32, 2, 34, 72, 11, 48, 49, 18, 20, 22, 57, 29, 31}
* X\_top\_features = X1\_sigma.iloc[:, list(initial\_selection)]
* *# shuffle the rows*
* **from** **numpy** **import** random, arange
* indexes = arange(Y.shape[0])
* random.shuffle(indexes)
* X\_cv = X\_top\_features.iloc[indexes,:]
* Y\_cv = Y.iloc[indexes]
* %%time
* cb = CatBoostClassifier(verbose=**False**, early\_stopping\_rounds=50)
* rfecv = RFECV(cb, scoring="accuracy", n\_jobs=-1, verbose=1)
* rfecv.fit(X\_cv, Y\_cv)
* Fitting estimator with 14 features.
* Fitting estimator with 13 features.
* Fitting estimator with 12 features.
* Fitting estimator with 11 features.
* Fitting estimator with 10 features.
* Fitting estimator with 9 features.
* Fitting estimator with 8 features.
* Fitting estimator with 7 features.
* CPU times: user 24min 31s, sys: 1min 19s, total: 25min 51s
* Wall time: 23min 19s
* RFECV(cv='warn',
* estimator=<catboost.core.CatBoostClassifier object at 0x7fa54ce8a3c8>,
* min\_features\_to\_select=1, n\_jobs=-1, scoring='accuracy', step=1,
* verbose=1)
* rfecv.n\_features\_
* 6
* rfecv.grid\_scores\_
* array([0.98148263, 0.99818639, 0.9991035 , 0.99990726, 0.99993817,
* 0.99995878, 0.99995878, 0.99995878, 0.99995878, 0.99994848,
* 0.99990726, 0.99989695, 0.99992787, 0.99990726])
* rfecv.support\_
* array([False, False, True, False, False, True, False, False, False,
* False, True, True, True, True])
* **import** **matplotlib.pyplot** **as** **plt**
* plt.plot(range(1, len(rfecv.grid\_scores\_) + 1), rfecv.grid\_scores\_)
* 

# Validating the 6 selected features using 5-fold CV

* cb\_cv['test\_score'].mean()
* 0.9998660378103773
* **from** **sklearn.model\_selection** **import** cross\_validate
* X\_top\_6 = X\_cv[X\_cv.columns[rfecv.support\_]]
* cb = CatBoostClassifier(verbose=500, early\_stopping\_rounds=50)
* %time cb\_cv = cross\_validate(cb, X\_top\_6, Y\_cv, cv=5, scoring="accuracy")
* cb\_cv
* CPU times: user 10min 14s, sys: 30.9 s, total: 10min 45s
* Wall time: 1min 28s
* {'fit\_time': array([17.38743424, 17.89787388, 18.17471766, 17.69102573, 17.14631081]),
* 'score\_time': array([0.01382089, 0.0118432 , 0.01252604, 0.01238298, 0.01172662]),
* 'test\_score': array([0.99994848, 0.99989696, 0.99994847, 0.99969085, 0.99984542])}
* *# get the top column names/numbers from the original dataframe:*
* list(X\_top\_6.columns)
* ['8', '38', '67', '119', '76', '78']

# Conclusion

* Using RFECV we have reduced our 14 best features down to 6 features while still retaining an average accuracy rate of 99.987% on 5-fold cross validation. However the run time for fitting the model has only been reduced by a small amount, going from an average of 19.19 seconds to 17.66 seconds. Similarly the scoring time went down from 0.0136 seconds to 0.0125 seconds.

# Part 3: Feature selection using Chi-squared

* **import** **numpy** **as** **np**
* **import** **scipy**
* **import** **pandas** **as** **pd**
* **import** **sklearn**
* mdata\_test= pd.read\_csv('test\_imperson\_without4n7\_balanced\_data.csv')
* mdata\_train= pd.read\_csv('train\_imperson\_without4n7\_balanced\_data.csv')
* Xtrain, Ytrain = mdata\_train.loc[:, mdata\_train.columns != '155'], mdata\_train['155']
* Xtest, Ytest = mdata\_test.loc[:, mdata\_test.columns != '155'], mdata\_test['155']
* X1\_sigma = Xtrain.loc[:, (Xtrain.std()>0)]
* X2\_sigma = Xtest.loc[:, (Xtest.std()>0)]
* **from** **catboost** **import** CatBoostClassifier
* params = {'loss\_function':'Logloss','eval\_metric':'AUC','verbose':54}
* cb = CatBoostClassifier(\*\*params)
* cb.fit(Xtrain, Ytrain,eval\_set=(Xtest,Ytest),use\_best\_model = **True**,plot=**True**)
* feat\_important = [t **for** t **in** zip(features,cb.get\_feature\_importance())]
* feat\_important\_df = pd.DataFrame(feat\_important,columns=['feature', 'VarImp'])
* feat\_important\_df = feat\_important\_df.sort\_values('VarImp', ascending=**False**)
* feat\_important\_df[feat\_important\_df['VarImp']>0]

|  |  |  |
| --- | --- | --- |
|  | feature | VarImp |
| 35 | 38 | 25.758057 |
| 64 | 67 | 20.601619 |
| 73 | 76 | 8.377924 |
| 5 | 8 | 7.573238 |
| 6 | 9 | 6.343218 |
| 77 | 80 | 6.305031 |
| 116 | 119 | 5.711694 |
| 137 | 140 | 4.680064 |
| 79 | 82 | 4.518620 |
| 4 | 6 | 1.608593 |
| 104 | 107 | 1.593854 |
| 58 | 61 | 1.192325 |
| 74 | 77 | 1.153818 |
| 3 | 5 | 1.014938 |
| 72 | 75 | 0.825646 |
| 119 | 122 | 0.468592 |
| 95 | 98 | 0.453014 |
| 61 | 64 | 0.445794 |
| 67 | 70 | 0.427517 |
| 105 | 108 | 0.263410 |
| 91 | 94 | 0.238859 |
| 87 | 90 | 0.220640 |
| 109 | 112 | 0.096981 |
| 75 | 78 | 0.050865 |
| 108 | 111 | 0.031422 |
| 139 | 142 | 0.021802 |
| 106 | 109 | 0.011947 |
| 76 | 79 | 0.006509 |
| 69 | 72 | 0.001866 |
| 102 | 105 | 0.001475 |
| 115 | 118 | 0.000418 |
| 103 | 106 | 0.000248 |

* len(feat\_important\_df[feat\_important\_df['VarImp']>0])
* 32
* selected\_features=[]
* **for** i **in** feat\_important\_df[feat\_important\_df['VarImp']>0]['feature']:
* selected\_features.append(int(i))
* selected\_features = set(selected\_features)
* print(selected\_features)
* {5, 6, 8, 9, 140, 142, 38, 61, 64, 67, 70, 72, 75, 76, 77, 78, 79, 80, 82, 90, 94, 98, 105, 106, 107, 108, 109, 111, 112, 118, 119, 122}
* X\_top\_features = Xtrain.iloc[:,list(selected\_features)]
* **from** **sklearn.model\_selection** **import** cross\_validate
* **import** **numpy**
* **from** **numpy** **import** random, arange
* indexes = arange(Ytrain.shape[0])
* numpy.random.shuffle(indexes)
* cb = CatBoostClassifier(verbose=**False**, early\_stopping\_rounds=50)
* cb\_cv = cross\_validate(cb, X\_cv, Y\_cv, cv=5, scoring="accuracy")
* cb\_cv
* Y\_cv = Ytrain.iloc[indexes]
* {'fit\_time': array([15.91051698, 15.76953602, 16.30591202, 16.81542993, 17.78671002]),
* 'score\_time': array([0.02405715, 0.01588607, 0.01655197, 0.01478696, 0.02184224]),
* 'test\_score': array([0.99984544, 0.99979392, 0.99969085, 0.99969085, 0.99994847])}

## chi-squared (chi²) statistical test for non-negative features to select 10 of the best features

* **import** **pandas** **as** **pd**
* **import** **numpy** **as** **np**
* **from** **sklearn.feature\_selection** **import** SelectKBest
* **from** **sklearn.feature\_selection** **import** chi2
* data = pd.read\_csv('test\_imperson\_without4n7\_balanced\_data.csv')
* Xtrain, Ytrain = mdata\_train.loc[:, mdata\_train.columns != '155'], mdata\_train['155']
* *#apply SelectKBest class to extract top 10 best features*
* bestfeatures = SelectKBest(score\_func=chi2, k=10)
* fit = bestfeatures.fit(Xtrain,Ytrain)
* dfscores = pd.DataFrame(fit.scores\_)
* dfcolumns = pd.DataFrame(Xtrain.columns)
* *#concat two dataframes for better visualization*
* featureScores = pd.concat([dfcolumns,dfscores],axis=1)
* featureScores.columns = ['feature','Score'] *#naming the dataframe columns*
* print(featureScores.nlargest(10,'Score')) *#print 10 best features*
* feature Score
* 68 71 32124.682195
* 48 51 28936.000138
* 64 67 28321.490224
* 44 47 24346.021809
* 151 154 13311.733744
* 47 50 12317.666392
* 5 8 12076.290588
* 6 9 12076.290588
* 65 68 7911.296191
* 79 82 7720.479234
* selected\_features=[]
* **for** i **in** featureScores.nlargest(10,'Score')['feature']:
* selected\_features.append(int(i)-3)
* selected\_features = set(selected\_features)
* X\_top\_features = Xtrain.iloc[:, list(selected\_features)]
* indexes = arange(Ytrain.shape[0])
* numpy.random.shuffle(indexes)
* X\_cv = X\_top\_features.iloc[indexes,:]
* Y\_cv = Ytrain.iloc[indexes]
* cb = CatBoostClassifier(verbose=**False**, early\_stopping\_rounds=50)
* cb\_cv = cross\_validate(cb, X\_cv, Y\_cv, cv=5, scoring="accuracy")
* cb\_cv
* {'fit\_time': array([15.02702808, 15.33611178, 16.46628404, 15.27807379, 16.1256578 ]),
* 'score\_time': array([0.01370907, 0.01399207, 0.0147438 , 0.01426625, 0.01386023]),
* 'test\_score': array([0.9994848 , 0.9994848 , 0.99969085, 0.9997939 , 0.9995878 ])}
* **from** **xgboost** **import** XGBClassifier
* gb = XGBClassifier(verbosity=0)
* gb\_cv = cross\_validate(gb, X\_cv, Y\_cv, cv=5, scoring="accuracy")
* gb\_cv
* {'fit\_time': array([2.434901 , 2.356637 , 2.3485291 , 2.431885 , 2.38992834]),
* 'score\_time': array([0.03784108, 0.02991891, 0.03046775, 0.03504109, 0.03634882]),
* 'test\_score': array([0.99778465, 0.99783617, 0.99788747, 0.99799052, 0.99768137])}

## Conclusion

* Using Chi-squared testing of each feature against the target column and selecting the k best we can get as good performance as using the feature importance reported by tree based algorithms such as catboost.

**import** **numpy** **as** **np**

**import** **scipy**

**import** **pandas** **as** **pd**

**import** **sklearn**

**from** **sklearn.feature\_selection** **import** SelectKBest

**from** **sklearn.feature\_selection** **import** chi2

**from** **sklearn.model\_selection** **import** cross\_validate

**from** **numpy** **import** random, arange

**from** **xgboost** **import** XGBClassifier

Xtrain, Ytrain = Xselected, Y

*#apply SelectKBest class to extract top 10 best features*

bestfeatures = SelectKBest(score\_func=chi2, k=10)

fit = bestfeatures.fit(Xtrain,Ytrain)

dfscores = pd.DataFrame(fit.scores\_)

dfcolumns = pd.DataFrame(Xtrain.columns)

*#concat two dataframes for better visualization*

featureScores = pd.concat([dfcolumns,dfscores],axis=1)

*#naming the dataframe columns*

featureScores.columns = ['feature','Score']

*#print 10 best features*

print(featureScores.nlargest(10,'Score'))

selected\_features = set((25,16,22,13,77,15,2,3,23,34))

**import** **numpy**

X\_top\_features = Xselected.iloc[:, list(selected\_features)]

indexes = numpy.arange(Ytrain.shape[0])

numpy.random.shuffle(indexes)

X\_cv = X\_top\_features.iloc[indexes,:]

Y\_cv = Ytrain.iloc[indexes]

indexes = arange(Ytrain.shape[0])

numpy.random.shuffle(indexes)

X\_cv = X\_top\_features.iloc[indexes,:]

Y\_cv = Ytrain.iloc[indexes]

cb = CatBoostClassifier(verbose=**False**, early\_stopping\_rounds=50)

cb\_cv = cross\_validate(cb, X\_cv, Y\_cv, cv=5, scoring="accuracy")

cb\_cv

{'fit\_time': array([16.88924599, 18.57903004, 19.61032724, 16.98861003, 17.83964992]),

'score\_time': array([0.01466632, 0.0195148 , 0.01424694, 0.01427627, 0.0139029 ]),

'test\_score': array([0.99953632, 0.9994848 , 0.99963932, 0.99948475, 0.99969085])}

gb = XGBClassifier(verbosity=0)

gb\_cv = cross\_validate(gb, X\_cv, Y\_cv, cv=5, scoring="accuracy")

gb\_cv

{'fit\_time': array([2.38882828, 2.80635309, 2.65520096, 2.61705399, 2.46459889]),

'score\_time': array([0.0297358 , 0.03513479, 0.03065014, 0.03140807, 0.03049994]),

'test\_score': array([0.99783617, 0.99835137, 0.99773289, 0.99762984, 0.99824815])}

Xtrain, Ytrain = NX2df, Y

*#apply SelectKBest class to extract top 10 best features*

bestfeatures = SelectKBest(score\_func=chi2, k=10)

fit = bestfeatures.fit(Xtrain,Ytrain)

dfscores = pd.DataFrame(fit.scores\_)

dfcolumns = pd.DataFrame(Xtrain.columns)

*#concat two dataframes for better visualization*

featureScores = pd.concat([dfcolumns,dfscores],axis=1)

*#naming the dataframe columns*

featureScores.columns = ['feature','Score']

*#print 10 best features*

print(featureScores.nlargest(10,'Score'))

selected\_features = set((25,16,22,13,77,15,2,3,23,27))

X\_top\_features = NX2df.iloc[:, list(selected\_features)]

indexes = numpy.arange(Ytrain.shape[0])

numpy.random.shuffle(indexes)

X\_cv = X\_top\_features.iloc[indexes,:]

Y\_cv = Ytrain.iloc[indexes]

cb = CatBoostClassifier(verbose=**False**, early\_stopping\_rounds=50)

cb\_cv = cross\_validate(cb, X\_cv, Y\_cv, cv=5, scoring="accuracy")

cb\_cv

{'fit\_time': array([19.48945689, 20.40593696, 18.60622501, 19.67188191, 21.3007338 ]),

'score\_time': array([0.01459622, 0.01379013, 0.01371908, 0.01435018, 0.01378512]),

'test\_score': array([0.99747553, 0.99726945, 0.99701154, 0.99690849, 0.99696002])}

gb = XGBClassifier(verbosity=0)

gb\_cv = cross\_validate(gb, X\_cv, Y\_cv, cv=5, scoring="accuracy")

gb\_cv

{'fit\_time': array([4.14661694, 4.41106415, 4.12002206, 4.19900227, 4.16188002]),

'score\_time': array([0.03509212, 0.03049994, 0.03502893, 0.03120995, 0.03449702]),

'test\_score': array([0.99690881, 0.99696033, 0.99634171, 0.99654782, 0.99690849])}

Xtrain, Ytrain = NX3df, Y

*#apply SelectKBest class to extract top 10 best features*

bestfeatures = SelectKBest(score\_func=chi2, k=10)

fit = bestfeatures.fit(Xtrain,Ytrain)

dfscores = pd.DataFrame(fit.scores\_)

dfcolumns = pd.DataFrame(Xtrain.columns)

*#concat two dataframes for better visualization*

featureScores = pd.concat([dfcolumns,dfscores],axis=1)

*#naming the dataframe columns*

featureScores.columns = ['feature','Score']

*#print 10 best features*

print(featureScores.nlargest(10,'Score'))

selected\_features = set((25,16,22,13,77,15,2,3,23,34))

X\_top\_features = NX3df.iloc[:, list(selected\_features)]

indexes = numpy.arange(Ytrain.shape[0])

numpy.random.shuffle(indexes)

X\_cv = X\_top\_features.iloc[indexes,:]

Y\_cv = Ytrain.iloc[indexes]

cb = CatBoostClassifier(verbose=**False**, early\_stopping\_rounds=50)

cb\_cv = cross\_validate(cb, X\_cv, Y\_cv, cv=5, scoring="accuracy")

cb\_cv

{'fit\_time': array([18.59923005, 16.90425181, 16.79899788, 17.6864748 , 18.00517201]),

'score\_time': array([0.01949787, 0.0240171 , 0.01366305, 0.01427627, 0.01497293]),

'test\_score': array([0.9997424 , 0.9997424 , 0.99953627, 0.99969085, 0.9991756 ])}

gb = XGBClassifier(verbosity=0)

gb\_cv = cross\_validate(gb, X\_cv, Y\_cv, cv=5, scoring="accuracy")

gb\_cv

{'fit\_time': array([2.54442596, 2.60541296, 2.54903793, 2.38389492, 2.48685789]),

'score\_time': array([0.03097701, 0.03572512, 0.02722311, 0.03145814, 0.026968 ]),

'test\_score': array([0.99804225, 0.99783617, 0.99778442, 0.99809357, 0.99747527])}

**from** **sklearn** **import** feature\_selection

Xtrain, Ytrain = PX2df, Y

*#apply SelectKBest class to extract top 10 best features*

bestfeatures = SelectKBest(k=10)

fit = bestfeatures.fit(Xtrain,Ytrain)

dfscores = pd.DataFrame(fit.scores\_)

dfcolumns = pd.DataFrame(Xtrain.columns)

*#concat two dataframes for better visualization*

featureScores = pd.concat([dfcolumns,dfscores],axis=1)

*#naming the dataframe columns*

featureScores.columns = ['feature','Score']

*#print 10 best features*

print(featureScores.nlargest(10,'Score'))

selected\_features = set((22,32,25,13,16,15,29,28,23,0))

X\_top\_features = PX2df.iloc[:, list(selected\_features)]

indexes = numpy.arange(Ytrain.shape[0])

numpy.random.shuffle(indexes)

X\_cv = X\_top\_features.iloc[indexes,:]

Y\_cv = Ytrain.iloc[indexes]

cb = CatBoostClassifier(verbose=**False**, early\_stopping\_rounds=50)

cb\_cv = cross\_validate(cb, X\_cv, Y\_cv, cv=5, scoring="accuracy")

cb\_cv

gb = XGBClassifier(verbosity=0)

gb\_cv = cross\_validate(gb, X\_cv, Y\_cv, cv=5, scoring="accuracy")

gb\_cv

**from** **sklearn.ensemble** **import** RandomForestClassifier

**from** **sklearn** **import** feature\_selection

*#apply SelectKBest class to extract top 10 best features*

bestfeatures = SelectKBest(k=10)

fit = bestfeatures.fit(Xtest,Ytest)

dfscores = pd.DataFrame(fit.scores\_)

dfcolumns = pd.DataFrame(Xtest.columns)

*#concat two dataframes for better visualization*

featureScores = pd.concat([dfcolumns,dfscores],axis=1)

*#naming the dataframe columns*

featureScores.columns = ['feature','Score']

*#print 10 best features*

print(featureScores.nlargest(10,'Score'))

f = msb / msw

selected\_features = set((66,72,64,76,35,47,48,44,78,5))

X\_top\_features = Xtest.iloc[:, list(selected\_features)]

indexes = numpy.arange(Ytest.shape[0])

numpy.random.shuffle(indexes)

Xtest\_cv = X\_top\_features.iloc[indexes,:]

Ytest\_cv = Ytest.iloc[indexes]

Xtrain, Ytrain = Xselected, Y

*#apply SelectKBest class to extract top 10 best features*

bestfeatures = SelectKBest(score\_func=chi2, k=10)

fit = bestfeatures.fit(Xtrain,Ytrain)

dfscores = pd.DataFrame(fit.scores\_)

dfcolumns = pd.DataFrame(Xtrain.columns)

*#concat two dataframes for better visualization*

featureScores = pd.concat([dfcolumns,dfscores],axis=1)

*#naming the dataframe columns*

featureScores.columns = ['feature','Score']

*#print 10 best features*

print(featureScores.nlargest(10,'Score'))

selected\_features = set((25,16,22,13,77,15,2,3,23,34))

**import** **numpy**

X\_top\_features = Xselected.iloc[:, list(selected\_features)]

indexes = numpy.arange(Ytrain.shape[0])

numpy.random.shuffle(indexes)

X\_cv = X\_top\_features.iloc[indexes,:]

Y\_cv = Ytrain.iloc[indexes]

**from** **sklearn.ensemble** **import** RandomForestClassifier

*#Create a Gaussian Classifier*

clf=RandomForestClassifier(n\_estimators=100)

*#Train the model using the training sets y\_pred=clf.predict(X\_test)*

clf.fit(X\_cv,Y\_cv)

*# prediction on test set*

y\_pred=clf.predict(Xtest\_cv)

*#Import scikit-learn metrics module for accuracy calculation*

**from** **sklearn** **import** metrics

*# Model Accuracy, how often is the classifier correct?*

print("Accuracy:",metrics.accuracy\_score(Ytest\_cv, y\_pred))

Xtrain, Ytrain = NX2df, Y

*#apply SelectKBest class to extract top 10 best features*

bestfeatures = SelectKBest(score\_func=chi2, k=10)

fit = bestfeatures.fit(Xtrain,Ytrain)

dfscores = pd.DataFrame(fit.scores\_)

dfcolumns = pd.DataFrame(Xtrain.columns)

*#concat two dataframes for better visualization*

featureScores = pd.concat([dfcolumns,dfscores],axis=1)

*#naming the dataframe columns*

featureScores.columns = ['feature','Score']

*#print 10 best features*

print(featureScores.nlargest(10,'Score'))

selected\_features = set((25,16,22,13,77,15,2,3,23,27))

X\_top\_features = NX2df.iloc[:, list(selected\_features)]

indexes = numpy.arange(Ytrain.shape[0])

numpy.random.shuffle(indexes)

X\_cv = X\_top\_features.iloc[indexes,:]

Y\_cv = Ytrain.iloc[indexes]

*#Create a Gaussian Classifier*

clf=RandomForestClassifier(n\_estimators=100)

*#Train the model using the training sets y\_pred=clf.predict(X\_test)*

clf.fit(X\_cv,Y\_cv)

*# prediction on test set*

y\_pred=clf.predict(Xtest\_cv)

*#Import scikit-learn metrics module for accuracy calculation*

**from** **sklearn** **import** metrics

*# Model Accuracy, how often is the classifier correct?*

print("Accuracy:",metrics.accuracy\_score(Ytest\_cv, y\_pred))

Xtrain, Ytrain = NX3df, Y

*#apply SelectKBest class to extract top 10 best features*

bestfeatures = SelectKBest(score\_func=chi2, k=10)

fit = bestfeatures.fit(Xtrain,Ytrain)

dfscores = pd.DataFrame(fit.scores\_)

dfcolumns = pd.DataFrame(Xtrain.columns)

*#concat two dataframes for better visualization*

featureScores = pd.concat([dfcolumns,dfscores],axis=1)

*#naming the dataframe columns*

featureScores.columns = ['feature','Score']

*#print 10 best features*

print(featureScores.nlargest(10,'Score'))

selected\_features = set((25,16,22,13,77,15,2,3,23,34))

X\_top\_features = NX3df.iloc[:, list(selected\_features)]

indexes = numpy.arange(Ytrain.shape[0])

numpy.random.shuffle(indexes)

X\_cv = X\_top\_features.iloc[indexes,:]

Y\_cv = Ytrain.iloc[indexes]

*#Create a Gaussian Classifier*

clf=RandomForestClassifier(n\_estimators=100)

*#Train the model using the training sets y\_pred=clf.predict(X\_test)*

clf.fit(X\_cv,Y\_cv)

*# prediction on test set*

y\_pred=clf.predict(Xtest\_cv)

*#Import scikit-learn metrics module for accuracy calculation*

**from** **sklearn** **import** metrics

*# Model Accuracy, how often is the classifier correct?*

print("Accuracy:",metrics.accuracy\_score(Ytest\_cv, y\_pred))

Xtrain, Ytrain = PX2df, Y

*#apply SelectKBest class to extract top 10 best features*

bestfeatures = SelectKBest(k=10)

fit = bestfeatures.fit(Xtrain,Ytrain)

dfscores = pd.DataFrame(fit.scores\_)

dfcolumns = pd.DataFrame(Xtrain.columns)

*#concat two dataframes for better visualization*

featureScores = pd.concat([dfcolumns,dfscores],axis=1)

*#naming the dataframe columns*

featureScores.columns = ['feature','Score']

*#print 10 best features*

print(featureScores.nlargest(10,'Score'))

**import** **numpy**

selected\_features = set((22,32,25,13,16,15,29,28,23,0))

X\_top\_features = PX2df.iloc[:, list(selected\_features)]

indexes = numpy.arange(Ytrain.shape[0])

numpy.random.shuffle(indexes)

X\_cv = X\_top\_features.iloc[indexes,:]

Y\_cv = Ytrain.iloc[indexes]

*#Create a Gaussian Classifier*

clf=RandomForestClassifier(n\_estimators=100)

*#Train the model using the training sets y\_pred=clf.predict(X\_test)*

clf.fit(X\_cv,Y\_cv)

*# prediction on test set*

y\_pred=clf.predict(Xtest\_cv)

*#Import scikit-learn metrics module for accuracy calculation*

**from** **sklearn** **import** metrics

*# Model Accuracy, how often is the classifier correct?*

print("Accuracy:",metrics.accuracy\_score(Ytest\_cv, y\_pred))

**from** **sklearn** **import** feature\_selection

*#apply SelectKBest class to extract top 10 best features*

bestfeatures = SelectKBest(k=10)

fit = bestfeatures.fit(Xtest,Ytest)

dfscores = pd.DataFrame(fit.scores\_)

dfcolumns = pd.DataFrame(Xtest.columns)

*#concat two dataframes for better visualization*

featureScores = pd.concat([dfcolumns,dfscores],axis=1)

*#naming the dataframe columns*

featureScores.columns = ['feature','Score']

*#print 10 best features*

print(featureScores.nlargest(10,'Score'))

f = msb / msw

selected\_features = set((66,72,64,76,35,47,48,44,78,5))

X\_top\_features = Xtest.iloc[:, list(selected\_features)]

indexes = numpy.arange(Ytest.shape[0])

numpy.random.shuffle(indexes)

Xtest\_cv = X\_top\_features.iloc[indexes,:]

Ytest\_cv = Ytest.iloc[indexes]

**from** **sklearn.naive\_bayes** **import** GaussianNB

Xtrain, Ytrain = Xselected, Y

*#apply SelectKBest class to extract top 10 best features*

bestfeatures = SelectKBest(score\_func=chi2, k=10)

fit = bestfeatures.fit(Xtrain,Ytrain)

dfscores = pd.DataFrame(fit.scores\_)

dfcolumns = pd.DataFrame(Xtrain.columns)

*#concat two dataframes for better visualization*

featureScores = pd.concat([dfcolumns,dfscores],axis=1)

*#naming the dataframe columns*

featureScores.columns = ['feature','Score']

*#print 10 best features*

print(featureScores.nlargest(10,'Score'))

selected\_features = set((25,16,22,13,77,15,2,3,23,34))

**import** **numpy**

X\_top\_features = Xselected.iloc[:, list(selected\_features)]

indexes = numpy.arange(Ytrain.shape[0])

numpy.random.shuffle(indexes)

X\_cv = X\_top\_features.iloc[indexes,:]

Y\_cv = Ytrain.iloc[indexes]

%time

model = GaussianNB()

model.fit(X\_cv, Y\_cv)

GaussianNB(priors=None, var\_smoothing=1e-09)

model.score(X\_cv,Y\_cv)

0.9606673261613289

model.score(Xtest\_cv,Ytest\_cv)

%time

Xtrain, Ytrain = NX2df, Y

*#apply SelectKBest class to extract top 10 best features*

bestfeatures = SelectKBest(score\_func=chi2, k=10)

fit = bestfeatures.fit(Xtrain,Ytrain)

dfscores = pd.DataFrame(fit.scores\_)

dfcolumns = pd.DataFrame(Xtrain.columns)

*#concat two dataframes for better visualization*

featureScores = pd.concat([dfcolumns,dfscores],axis=1)

*#naming the dataframe columns*

featureScores.columns = ['feature','Score']

*#print 10 best features*

print(featureScores.nlargest(10,'Score'))

selected\_features = set((25,16,22,13,77,15,2,3,23,27))

X\_top\_features = NX2df.iloc[:, list(selected\_features)]

indexes = numpy.arange(Ytrain.shape[0])

numpy.random.shuffle(indexes)

X\_cv = X\_top\_features.iloc[indexes,:]

Y\_cv = Ytrain.iloc[indexes]

%time

model = GaussianNB()

model.fit(X\_cv, Y\_cv)

model.predict(X\_cv)

model.score(X\_cv,Y\_cv)

model.score(Xtest\_cv,Ytest\_cv)

%time

Xtrain, Ytrain = NX3df, Y

*#apply SelectKBest class to extract top 10 best features*

bestfeatures = SelectKBest(score\_func=chi2, k=10)

fit = bestfeatures.fit(Xtrain,Ytrain)

dfscores = pd.DataFrame(fit.scores\_)

dfcolumns = pd.DataFrame(Xtrain.columns)

*#concat two dataframes for better visualization*

featureScores = pd.concat([dfcolumns,dfscores],axis=1)

*#naming the dataframe columns*

featureScores.columns = ['feature','Score']

*#print 10 best features*

print(featureScores.nlargest(10,'Score'))

selected\_features = set((25,16,22,13,77,15,2,3,23,34))

X\_top\_features = NX3df.iloc[:, list(selected\_features)]

indexes = numpy.arange(Ytrain.shape[0])

numpy.random.shuffle(indexes)

X\_cv = X\_top\_features.iloc[indexes,:]

Y\_cv = Ytrain.iloc[indexes]

model = GaussianNB()

model.fit(X\_cv, Y\_cv)

model.predict(X\_cv)

model.score(X\_cv,Y\_cv)

model.score(Xtest\_cv,Ytest\_cv)

Xtrain, Ytrain = PX2df, Y

*#apply SelectKBest class to extract top 10 best features*

bestfeatures = SelectKBest(k=10)

fit = bestfeatures.fit(Xtrain,Ytrain)

dfscores = pd.DataFrame(fit.scores\_)

dfcolumns = pd.DataFrame(Xtrain.columns)

*#concat two dataframes for better visualization*

featureScores = pd.concat([dfcolumns,dfscores],axis=1)

*#naming the dataframe columns*

featureScores.columns = ['feature','Score']

*#print 10 best features*

print(featureScores.nlargest(10,'Score'))

**import** **numpy**

selected\_features = set((22,32,25,13,16,15,29,28,23,0))

X\_top\_features = PX2df.iloc[:, list(selected\_features)]

indexes = numpy.arange(Ytrain.shape[0])

numpy.random.shuffle(indexes)

X\_cv = X\_top\_features.iloc[indexes,:]

Y\_cv = Ytrain.iloc[indexes

model = GaussianNB()

model.fit(X\_cv, Y\_cv)

model.predict(X\_cv)

model.score(X\_cv,Y\_cv)

model.score(Xtest\_cv,Ytest\_cv)

refining

*#load data*

traindata= pd.read\_csv(r'C:\Users\Student\Git\AML\_project\datasets\train\_imperson\_without4n7\_balanced\_data.csv')

testdata= pd.read\_csv(r'C:\Users\Student\Git\AML\_project\datasets\test\_imperson\_without4n7\_balanced\_data.csv')

*# Separate X and Y*

X\_train, Y\_train = traindata.loc[:, traindata.columns != '155'], traindata['155']

X\_test, Y\_test = testdata.loc[:, testdata.columns != '155'], testdata['155']

**from** **sklearn.model\_selection** **import** GridSearchCV

pipeline1 = Pipeline([

('zero variance', VarianceThreshold()),

('top features', SelectKBest(f\_classif)),

('ada',AdaBoostClassifier())])

grid=GridSearchCV(cv=10,

estimator=pipeline1,

param\_grid={'ada\_\_n\_estimators': [10,50,100,150],

'ada\_\_learning\_rate':[0.001,0.1,0.01],

'top features\_\_k':[15,20,30,35]},

scoring = 'accuracy',

n\_jobs=-1)

grid.fit(X\_train, Y\_train)

print(grid.best\_params\_)

print(grid.best\_score\_)

{'ada\_\_learning\_rate': 0.01, 'ada\_\_n\_estimators': 100, 'top features\_\_k': 35}

**from** **sklearn.model\_selection** **import** GridSearchCV

pipeline2= Pipeline([

('zero variance', VarianceThreshold()),

('norm 1', Normalizer()),

('top features', SelectKBest(chi2)),

('model',LogisticRegression())])

grid=GridSearchCV(cv=10,

estimator=pipeline2,

param\_grid={'top features\_\_k':[15,20,30,35],

'model\_\_C': [0.01, 0.1, 1, 10, 100, 1000]},

scoring = 'accuracy',

n\_jobs=-1)

grid.fit(X\_train, Y\_train)

*# sorted(pipeline2.get\_params().keys()) #list of the parameters you can tune*

print(grid.best\_params\_)

print(grid.best\_score\_)

C:\Users\amatu\Anaconda3\lib\site-packages\sklearn\linear\_model\logistic.py:432: FutureWarning: Default solver will be changed to 'lbfgs' in 0.22. Specify a solver to silence this warning.

FutureWarning)

{'model\_\_C': 1000, 'top features\_\_k': 20}

0.98313136309303

**from** **sklearn.model\_selection** **import** GridSearchCV

pipeline2a= Pipeline([

('zero variance', VarianceThreshold()),

('minmax', MinMaxScaler()),

('top features', SelectKBest(chi2)),

('model',LogisticRegression())])

grid=GridSearchCV(cv=10,

estimator=pipeline2a,

param\_grid={'top features\_\_k':[15,20,30,35],'model\_\_C': [0.01, 0.1, 1, 10, 100, 1000]},

scoring = 'accuracy',

n\_jobs=-1)

grid.fit(X\_train, Y\_train)

*#sorted(pipeline1.get\_params().keys()) #list of the parameters you can tune*

print(grid.best\_params\_)

print(grid.best\_score\_)

C:\Users\amatu\Anaconda3\lib\site-packages\sklearn\linear\_model\logistic.py:432: FutureWarning: Default solver will be changed to 'lbfgs' in 0.22. Specify a solver to silence this warning.

FutureWarning)

{'model\_\_C': 1000, 'top features\_\_k': 20}

0.9875520382506904

pipeline3 = Pipeline([

('zero variance', VarianceThreshold()),

('scale 0\_1', MinMaxScaler()),

('top features', SelectKBest(f\_classif)),

('ada',AdaBoostClassifier())])

grid=GridSearchCV(cv=10,

estimator=pipeline3,

param\_grid={'ada\_\_n\_estimators': [10,50,100,150],

'ada\_\_learning\_rate':[0.001,0.1,0.01],

'top features\_\_k':[15,20,30,35]},

scoring = 'accuracy',

n\_jobs=-1)

grid.fit(X\_train, Y\_train)

*#sorted(pipeline1.get\_params().keys()) #list of the parameters you can tune*

print(grid.best\_params\_)

print(grid.best\_score\_)

{'ada\_\_learning\_rate': 0.01, 'ada\_\_n\_estimators': 100, 'top features\_\_k': 35}

pipeline1 = Pipeline([

('zero variance', VarianceThreshold()),

('top20 features', SelectKBest(f\_classif, k=35)),

('ada',AdaBoostClassifier(n\_estimators=100,learning\_rate=0.01))])

pipeline1.fit(X\_train,Y\_train)

Y\_predict=pipeline1.predict(X\_test)

accuracy\_score(Y\_test, Y\_predict)\*100

pipeline2= Pipeline([

('zero variance', VarianceThreshold()),

('norm 1', Normalizer()),

('top20 features', SelectKBest(chi2, k=20)),

('model',LogisticRegression(C=1000,solver='newton-cg'))])

pipeline2.fit(X\_train,Y\_train)

Y\_predict=pipeline2.predict(X\_test)

accuracy\_score(Y\_test, Y\_predict)\*100

pipeline3 = Pipeline([

('zero variance', VarianceThreshold()),

('scale 0\_1', MinMaxScaler()),

('top20 features', SelectKBest(f\_classif, k=35)),

('ada',AdaBoostClassifier(n\_estimators=100,learning\_rate=0.01))])

pipeline3.fit(X\_train,Y\_train)

Y\_predict=pipeline3.predict(X\_test)

accuracy\_score(Y\_test, Y\_predict)\*100

pipeline5 = Pipeline([

('zero variance', VarianceThreshold()),

('scale 0\_1', MinMaxScaler()),

('top20 features', SelectKBest(chi2, k=20)),

('ada',AdaBoostClassifier())])

grid=GridSearchCV(cv=10,

estimator=pipeline5,

param\_grid={'ada\_\_n\_estimators': [10,50,100,150],

'ada\_\_learning\_rate':[0.001,0.1,0.01]},

scoring = 'accuracy',

n\_jobs=-1)

grid.fit(X\_train, Y\_train)

print(grid.best\_params\_)

print(grid.best\_score\_)

pipeline5 = Pipeline([

('zero variance', VarianceThreshold()),

('scale 0\_1', MinMaxScaler()),

('top20 features', SelectKBest(chi2, k=20)),

('ada',AdaBoostClassifier())])

grid=GridSearchCV(cv=10,

estimator=pipeline5,

param\_grid={'ada\_\_n\_estimators': [10,50,100,150],

'ada\_\_learning\_rate':[0.001,0.1,0.01]},

scoring = 'accuracy',

n\_jobs=-1)

grid.fit(X\_train, Y\_train)

print(grid.best\_params\_)

print(grid.best\_score\_)

pipeline1 = Pipeline([

('zero variance', VarianceThreshold()),

('top20 features', SelectKBest(f\_classif, k=20)),

('ada',AdaBoostClassifier(n\_estimators=150,learning\_rate=0.1))])

pipeline1.fit(X\_train,Y\_train)

Y\_predict=pipeline1.predict(X\_test)

accuracy\_score(Y\_test, Y\_predict)\*100

pipeline2 = Pipeline([

('zero variance', VarianceThreshold()),

('scale 0\_1', MinMaxScaler()),

('top20 features', SelectKBest(f\_classif, k=20)),

('ada',AdaBoostClassifier(n\_estimators=150,learning\_rate=0.1))])

pipeline2.fit(X\_train,Y\_train)

Y\_predict=pipeline2.predict(X\_test)

accuracy\_score(Y\_test, Y\_predict)\*100

pipeline3= Pipeline([

('zero variance', VarianceThreshold()),

('scale 0\_1', MinMaxScaler()),

('top20 features', SelectKBest(chi2, k=20)),

('model',LogisticRegression(C=1000,solver='newton-cg'))])

pipeline3.fit(X\_train,Y\_train)

Y\_predict=pipeline3.predict(X\_test)

accuracy\_score(Y\_test, Y\_predict)\*100

evaluation

*Pipeline 1 - Accuracy*

pipeline1 = Pipeline([

('zero variance', VarianceThreshold()),

('top20 features', SelectKBest(f\_classif, k=20)),

('ada',AdaBoostClassifier(n\_estimators=150,learning\_rate=0.1))])

pipeline1.fit(X\_train,Y\_train)

Y\_predict1=pipeline1.predict(X\_test)

accuracy\_score(Y\_test, Y\_predict1)\*100

matrix = confusion\_matrix(Y\_test, Y\_predict1)

print(matrix)

p1\_TP = matrix[0][0]

p1\_TN = matrix[1][1]

p1\_FP = matrix[1][0]

p1\_FN = matrix[0][1]

print('')

print('True Positive = **%.0f**' % p1\_TP)

print('True Negative = **%.0f**' % p1\_TN)

print('')

print('False Positive = **%.0f**' % p1\_FP)

print('False Negative = **%.0f**' % p1\_FN)

[[19708 371]

[ 0 20079]]

True Positive = 19708

True Negative = 20079

False Positive = 0

False Negative = 371

*#Pipeline 1 - Evaluation Metrics*

p1\_Acc = (p1\_TP + p1\_TN) / (p1\_TP + p1\_TN + p1\_FP + p1\_FN)

print('Acc = **%.4f**' % p1\_Acc)

p1\_DR = p1\_TP / (p1\_TP + p1\_FN)

print('DR(Recall) = **%.4f**' % p1\_DR)

p1\_Prec = p1\_TP / (p1\_TP + p1\_FP)

print('Precision = **%.4f**' % p1\_Prec)

p1\_FAR = p1\_FP / (p1\_TN + p1\_FP)

print('FAR = **%.4f**' % p1\_FAR)

p1\_FNR = p1\_FN / (p1\_FN + p1\_TP)

print('FNR = **%.4f**' % p1\_FNR)

p1\_F1 = (2 \* p1\_TP) / (2\*p1\_TP + p1\_FP + p1\_FN)

print('F1 = **%.4f**' % p1\_F1)

p1\_MCC = ((p1\_TP \* p1\_TN) - (p1\_FP \* p1\_FN)) / math.sqrt((p1\_TP + p1\_FP)\*(p1\_TP + p1\_FN)\*(p1\_TN + p1\_FP)\*(p1\_TN + p1\_FN))

print('MCC = **%.4f**' % p1\_MCC)

Acc = 0.9908

DR(Recall) = 0.9815

Precision = 1.0000

FAR = 0.0000

FNR = 0.0185

F1 = 0.9907

MCC = 0.9817

*#Pipeline 1 - ROC Curve*

probs = pipeline1.predict\_proba(X\_test)

probs = probs[:, 1]

auc = roc\_auc\_score(Y\_test, probs)

print('AUC: **%.3f**' % auc)

fpr, tpr, thresholds = roc\_curve(Y\_test, probs)

pyplot.plot([0,1],[0,1], linestyle = '--')

pyplot.plot(fpr,tpr,marker='.')

pyplot.show()

*#Pipeline 2*

pipeline2 = Pipeline([

('zero variance', VarianceThreshold()),

('scale 0\_1', MinMaxScaler()),

('top20 features', SelectKBest(f\_classif, k=20)),

('ada',AdaBoostClassifier(n\_estimators=150,learning\_rate=0.1))])

pipeline2.fit(X\_train,Y\_train)

Y\_predict2=pipeline2.predict(X\_test)

accuracy\_score(Y\_test, Y\_predict2)\*100

*#Pipeline 2 - Confusion Matrix*

matrix = confusion\_matrix(Y\_test, Y\_predict2)

print(matrix)

p2\_TP = matrix[0][0]

p2\_TN = matrix[1][1]

p2\_FP = matrix[1][0]

p2\_FN = matrix[0][1]

print('')

print('True Positive = **%.0f**' % p2\_TP)

print('True Negative = **%.0f**' % p2\_TN)

print('')

print('False Positive = **%.0f**' % p2\_FP)

print('False Negative = **%.0f**' % p2\_FN)

True Positive = 19708

True Negative = 20079

False Positive = 0

False Negative = 371

*#Pipeline 2 - Evaluation Metrics*

p2\_Acc = (p2\_TP + p2\_TN) / (p2\_TP + p2\_TN + p2\_FP + p2\_FN)

print('Acc = **%.4f**' % p2\_Acc)

p2\_DR = p2\_TP / (p2\_TP + p2\_FN)

print('DR(recall) = **%.4f**' % p2\_DR)

p2\_Prec = p2\_TP / (p2\_TP + p2\_FP)

print('Precision = **%.4f**' % p2\_Prec)

p2\_FAR = p2\_FP / (p2\_TN + p2\_FP)

print('FAR = **%.4f**' % p2\_FAR)

p2\_FNR = p2\_FN / (p2\_FN + p2\_TP)

print('FNR = **%.4f**' % p2\_FNR)

p2\_F1 = (2 \* p2\_TP) / (2\*p2\_TP + p2\_FP + p2\_FN)

print('F1 = **%.4f**' % p2\_F1)

p2\_MCC = ((p2\_TP \* p2\_TN) - (p2\_FP \* p2\_FN)) / math.sqrt((p2\_TP + p2\_FP)\*(p2\_TP + p2\_FN)\*(p2\_TN + p2\_FP)\*(p2\_TN + p2\_FN))

print('MCC = **%.4f**' % p2\_MCC)

Acc = 0.9908

DR(recall) = 0.9815

Precision = 1.0000

FAR = 0.0000

FNR = 0.0185

F1 = 0.9907

MCC = 0.9817

*#Pipeline 2 - ROC Curve*

probs = pipeline2.predict\_proba(X\_test)

probs = probs[:, 1]

auc = roc\_auc\_score(Y\_test, probs)

print('AUC: **%.3f**' % auc)

fpr, tpr, thresholds = roc\_curve(Y\_test, probs)

pyplot.plot([0,1],[0,1], linestyle = '--')

pyplot.plot(fpr,tpr,marker='.')

pyplot.show()

*#Pipeline 3*

pipeline3= Pipeline([

('zero variance', VarianceThreshold()),

('scale 0\_1', MinMaxScaler()),

('top20 features', SelectKBest(chi2, k=20)),

('model',LogisticRegression(C=1000,solver='newton-cg'))])

pipeline3.fit(X\_train,Y\_train)

Y\_predict3=pipeline3.predict(X\_test)

accuracy\_score(Y\_test, Y\_predict3)\*100

*#Pipleline 3 - Classification Report*

report = classification\_report(Y\_test, Y\_predict3)

print(report)

precision recall f1-score support

0 0.99 0.98 0.99 20079

1 0.98 0.99 0.99 20079

accuracy 0.99 40158

macro avg 0.99 0.99 0.99 40158

weighted avg 0.99 0.99 0.99 40158

matrix = confusion\_matrix(Y\_test, Y\_predict3)

print(matrix)

p3\_TP = matrix[0][0]

p3\_TN = matrix[1][1]

p3\_FP = matrix[1][0]

p3\_FN = matrix[0][1]

print('')

print('True Positive = **%.0f**' % p3\_TP)

print('True Negative = **%.0f**' % p3\_TN)

print('')

print('False Positive = **%.0f**' % p3\_FP)

print('False Negative = **%.0f**' % p3\_FN)

True Positive = 19669

True Negative = 19938

False Positive = 141

False Negative = 410

*#Pipeline 3 - Evaluation Metrics*

p3\_Acc = (p3\_TP + p3\_TN) / (p3\_TP + p3\_TN + p3\_FP + p3\_FN)

print('Acc = **%.4f**' % p3\_Acc)

p3\_DR = p3\_TP / (p3\_TP + p3\_FN)

print('DR(recall) = **%.4f**' % p3\_DR)

p3\_Prec = p3\_TP / (p3\_TP + p3\_FP)

print('Precision = **%.4f**' % p3\_Prec)

p3\_FAR = p3\_FP / (p3\_TN + p3\_FP)

print('FAR = **%.4f**' % p3\_FAR)

p3\_FNR = p3\_FN / (p3\_FN + p3\_TP)

print('FNR = **%.4f**' % p3\_FNR)

p3\_F1 = (2 \* p3\_TP) / (2\*p3\_TP + p3\_FP + p3\_FN)

print('F1 = **%.4f**' % p3\_F1)

p3\_MCC = ((p3\_TP \* p3\_TN) - (p3\_FP \* p3\_FN)) / math.sqrt((p3\_TP + p3\_FP)\*(p3\_TP + p3\_FN)\*(p3\_TN + p3\_FP)\*(p3\_TN + p3\_FN))

print('MCC = **%.4f**' % p3\_MCC)

Acc = 0.9863

DR(recall) = 0.9796

Precision = 0.9929

FAR = 0.0070

FNR = 0.0204

F1 = 0.9862

MCC = 0.9726

*#Pipeline 3 - ROC Curve*

probs = pipeline3.predict\_proba(X\_test)

probs = probs[:, 1]

auc = roc\_auc\_score(Y\_test, probs)

print('AUC: **%.3f**' % auc)

fpr, tpr, thresholds = roc\_curve(Y\_test, probs)

pyplot.plot([0,1],[0,1], linestyle = '--')

pyplot.plot(fpr,tpr,marker='.')

pyplot.show()