COMP3009
Data Mining Assignment

Troy Engelhardt | 18815179

SATURDAY, 28 SEPTEMBER 2019

Summary

As a student taking the data science major this assignment wasn't necessarily too many new concepts, but it was still extremely enjoyable! Having a dataset which you don't necessarily know anything about and being forced to explore it in many different ways creates a very interesting workflow that is somewhat similar, but also different in ways to the traditional analytics workflow. The opportunity to freely experiment with Python machine learning frameworks (like sklearn) has been instrumental, and very rewarding to finally get a good grasp on a framework which is cross-industry recognised. The data preparation/cleaning stage was tedious, as it usually is, but having the opportunity to leverage ipython widgets really, really helped the process go much faster. I learned a lot about how pandas data frames deal with typecasting, and how transposing the data frame **after** nicely formatting your types will completely ruin all your hard work.

I hadn't had much experience standardising/scaling data, and reading into the sklearn utilities for doing so have really taught me a lot. Digging deeper into the dataset to ensure that when I was scaling the data I was preserving as much shape as possible; that was quite rewarding to figure out. Similarly, reading the documentation surrounding the sklearn feature selection was quite intriguing. In statistics we've been taught methods of forward/backward selection, and exposed to a lot of the others too. However, this time around I decided to dive into all three of the main methods used within the documentation to see how it was to implement, and how effective it was overall.

Fitting the models and comparing them is something we're quite seasoned at, but within Python it just feels a bit different - and it's nice to get this experience. In R, a lot of the statistical models can be implemented across a variety of different packages. This means sometimes when you're fitting a different model with the same data, you still have to do a lot of preprocessing just to try a new model. Due to sklearn's standardisation of the model fitting process, this is definitely not the case. As soon as you have your data in the correct format it's all system's go and you can implement a huge amount of different models which suit the application at hand. Although, this is somewhat dangerous for people naively fitting models without properly understanding the statistics, or the underlying assumptions, but it sure makes the implementation part easier. In a similar sense, the model comparison/evaluation is just as easy; confusion matrices, accuracy scores, f1 scores, and almost any other metric you can think of has an easy to understand, standardised method call.

With regards to the content of the assignment, the major findings were that a significant amount of the dataset was categorical, this severely inflates the dimensionality as soon as you push all these categorical variables into their *dummy* form too. It was found through my analysis that the k-nearest neighbour classification algorithm was the best performer overall; having consistently high accuracy and f1 scores, along with getting nearly an exact 50/50 split in the final 100 row prediction set. This model was set to have a nearest neighbour parameter equal to 73, and the p parameter set to one, meaning the model is utilising the Manhattan distance measure. Other models performed quite similarly across the board, but the k-means classifier was consistently horrible, with accuracy and f1 measures hovering around the 50% area; barely benchmarking better than a blind guess. The support vector machine model was an interesting case as it scored highly in accuracy and f1 during cross validation. It even managed to score quite high within the testing set which was set aside too, roughly 80% accuracy & f1. However, when it came to predicting the final 100 rows the model seemed to fall apart, predicting 41 zeros and 59 ones, quite a ways away from the 50/50 split which is expected. All the other models seemed to perform as you might expect, with accuracy & f1 scores fluctuating around the low 70% range, test set generated accuracy & f1 scores which matched that, along with predictions on the final 100 rows which weren't perfect 50/50 splits - but definitely weren't too far off.

Overall, this assignment has been a great way to improve my current data science skills, experiment with new tools that I've never got the opportunity to do so, and to learn about some completely new concepts. As a suggestion, it would've been nice if we could submit our properly formatted *predict.csv* file through an API which returns back an overall accuracy score. To prevent spam, or students hitting the API with every possible combination of the 100 zeros and ones, you could have it attached to a student's ID. This could allow a limited number of pushes of predictions to the API per week/semester - just an idea.

Methodology

Data Preparation

Irrelevant Attributes

The first obvious step when approaching a new dataset is to try reduce its dimensionality by getting rid of irrelevant attributes. It's hard to judge what constitutes an irrelevant attribute, especially when you don't know the intrinsic meaning behind what each variable is representing. However, this analysis can at a first glance conclude that the *'ID'* attribute is indeed irrelevant, and thus should be removed from the overall dataset. This was the only attribute which came under the category of being irrelevant but didn't also come under any of the more specific sections which follow, like missing entries or duplication.

- **Feature:** ID
  - **Type:** numeric, integer (actually a row-wise label)
  - **Issue(s):** label column holds no meaning when we're assuming the dataset has no sequential/time-series structure
  - **Decision:** drop (delete) the entire column from the dataset
  - **Reason:** row number has no bearing on our analysis and will contribute negatively to any future prediction attempts
  - **Method(s) used:** pd.df.drop( )

Missing Entries

Dealing with missing entries within the overall dataset was perhaps one of the steps which took the most care. Depending on how much data was missing from a variable the method of dealing with the issue can vary widely. As a first step a simple table was created which displays the percentage of data missing with respect to each column within the dataset. Firstly, the missing data within the *'Class'* variable can be ignored, this is because these are the values we are expected to predict within the final stages of the analysis, but everything else must be dealt with properly.

Looking at this table it is quite obvious that *'att13'* and *'att19'* have almost their whole column of data missing, 93% and 94% missing respectively. As a result, these columns were dropped from the overall dataset completely because there is no logical way to deal with such a vast amount of missing entries; especially knowing nothing about the feature's meaning themselves.

The next step was to look into the remaining four features which seemed to all have less than one percent missing data. As there is such a small amount of missing data here we can definitely just *'fill in the blanks'* so to speak, but this process still requires careful attention due to the **types** of data represented in these columns. When exploring these features further it's clear that *'att3'* and *'att9'* are categorical (nominal) variables, this means that it would be completely meaningless to fill in missing values with an overall column mean. Instead, it was decided that the most occurring value (mode) within the column would be a better choice to deal with the missing values within these two columns.

Now, with regards to missing data, the only remaining two columns to deal with are *'att25'* and *'att28.'* These two columns are quite clearly representing numeric variables, and thus can be dealt with using traditional column median or mean methods. Due to such a small amount of missing values the choice of median or mean would likely have an insignificant effect on any future analysis. However, by looking at a simple boxplot for both of these variables it's obvious that *'att25'* covers a much wider range of values and has a larger standard deviation, thus it was decided to take the median value across this column to fill in the blanks. Taking the same approach we can see that *'att28'* has a much smaller range of values, a smaller standard deviation, and therefore it was decided to take the mean value as the method to fill in the blanks within this column.

As a final sanity check the original table showcasing the proportion of missing values per column was rerun. The only column which appeared was the *'Class'* column which we already have stated **should** actually contain roughly 10% of its data missing for our predictions. Therefore, we know all our remaining columns have 0% missing data and we can conclude this part of our data cleaning/preparation.

- **Feature:** att13
  - **Type:** categorical
  - **Issue(s):** 93% of the column values are missing
  - **Decision:** drop (delete) the entire column from the dataset
  - **Reason:** filling in missing entries with such a small sample will do more harm than good
  - **Method(s) used:** pd.df.drop( )

- **Feature:** att19
  - **Type:** numeric
  - **Issue(s):** 94% of the column values are missing
  - **Decision:** drop (delete) the entire column from the dataset
  - **Reason:** filling in missing entries with such a small sample will do more harm than good
  - **Method(s) used:** pd.df.drop( )

- **Feature:** att3
  - **Type:** categorical
  - **Issue(s):** 0.36% of the column values are missing
  - **Decision:** fill missing values in with the overall column mode
  - **Reason:** the best way to deal with missing data in categorical columns, that aren't known to be time-series, is to fill them in with the column mode
  - **Method(s) used:** pd.df.fillna( )

- **Feature:** att9
  - **Type:** categorical
  - **Issue(s):** 0.45% of the column values are missing
  - **Decision:** fill missing values in with the overall column mode
  - **Reason:** the best way to deal with missing data in categorical columns, that aren't known to be time-series, is to fill them in with the column mode
  - **Method(s) used:** pd.df.fillna( )

- **Feature:** att25
  - **Type:** numeric
  - **Issue(s):** 0.27% of the column values are missing
  - **Decision:** fill missing values with overall column median
  - **Reason:** large column range and standard deviation makes median a better choice than mean
  - **Method(s) used:** pd.df.fillna( )

- **Feature:** att28
  - **Type:** numeric
  - **Issue(s):** 0.55% of the column values are missing
  - **Decision:** fill missing values with overall column mean
  - **Reason:** column mean is suitable due to the smaller overall range and standard deviation
  - **Method(s) used:** pd.df.fillna( )

	column_name	percent_missing
Class	Class	9.090909
att3	att3	0.363636
att9	att9	0.454545
att13	att13	93.454545
att19	att19	94.000000
att25	att25	0.272727
att28	att28	0.545455

TABLE 1. TABLE SHOWCASING THE PROPORTION OF MISSING VALUES (BEFORE PROCESSING).

	column_name	percent_missing
Class	Class	9.090909

TABLE 2. TABLE SHOWCASING THE PROPORTION OF MISSING VALUES (AFTER PROCESSING).

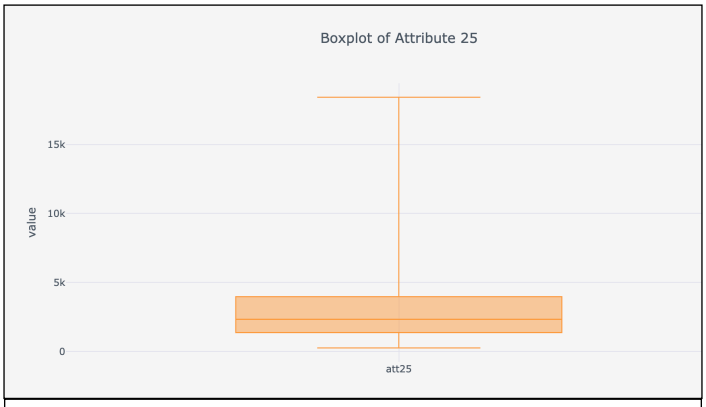

FIGURE 1. BOXPLOT OF ATTRIBUTE 25.

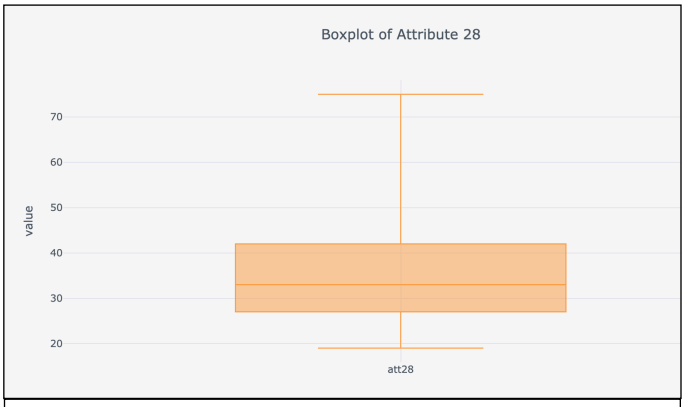

FIGURE 2. BOXPLOT OF ATTRIBUTE 28.

Duplicates

Duplicated values can cause numerous issues when following through with any analysis, so they must be dealt with during the data preparation phase. There are different types of duplication within a dataset; namely columns which contain the exact same value (full within column duplication), numerous columns which contain matching values (pairwise column duplication), and rows which contain matching values (pairwise row duplication). Similar to our various cases of missing values, each type of duplication here usually has to be approached uniquely, and carefully.

Firstly, columns which contain the exact same value for every single row have no effect on any future predictions, as they are quite literally unchanged by the differing target class. The features named *'att14'* and '*att17'* fall under this category of duplication, and this means that they can be dealt with simply and swiftly by dropping them from the overall dataset.

The next type of duplication, pairwise column duplication, is perhaps the most important to deal with. If two columns are exact duplicates of one another this means that they are perfectly correlated. This becomes a problem when fitting a variety of statistical techniques as perfect correlation means one column is a linear combination of another; meaning it will be impossible to invert your overall matrix. A method to deal with this duplication could be to calculate your correlation and delete all columns where the absolute correlation is equal to one - but this has its issues and is rather tedious. Instead, we can transpose our whole dataset and leverage the pandas drop duplicates function. This method typically only works with duplicated rows, but if we transpose our dataset, our columns literally become our rows, allowing us to utilise this function. This process removes columns *'att8'* and '*att24'* as they are known to perfectly correlate with columns '*att1'* and *'att18'* respectively.

The final type of duplication, pairwise row duplication, is possible the most controversial of the three to deal with, and as a result should take the most attention. In general, just because a row is perfectly duplicated doesn't necessarily mean it should be instantly dismissed. For example, you could do an experiment twice and get the exact same result; does this mean the experiment is useless and data should be throw away, or rather that the relationships are predictable and data should be kept? This is a subjective choice taken by the analyser, and there really is no right or wrong answer across all cases. For this analysis the choice was made to simply delete all duplicated rows, except for the first instance of the duplication of course. This choice was made because the the likelihood of 25 parameters all being the same is rather low, especially considering their rather low correlation across the board. It has been deemed that the duplication would likely have been due to processing, or data entry error, thus the choice was made to delete these rows. A total of 200 rows were deleted in this process, if required a full list of all 200 row indexes which were dropped can be found within the Python code.

- **Feature:** att14, att17
  - **Type:** categorical
  - **Issue(s):** column is constant; i.e. every row value is duplicated
  - **Decision:** drop (delete) the entire column from the dataset
  - **Reason:** column values do not change with respect to our target class, thus columns carry no predictive meaning

  - **Method(s) used:** pd.df.drop( )

- **Feature:** att8, att24
  - **Type:** categorical & numeric respectively
  - **Issue(s):** columns are an exact duplicate of other column(s) within the dataset
  - **Decision:** drop (delete) the entire column from the dataset
  - **Reason:** columns showcase perfect correlation with other columns which will cause problems in further statistical modelling/prediction (non-finite solution to matrix inverse & determinant)
  - **Method(s) used:** pd.df.T & pd.df.drop_duplicates( )

- **Rows:** see analysis for full list; 200 rows total
  - **Type:** rows (not columns), so mixed in types
  - **Issue(s):** rows are an exact duplicate of other row(s) within the dataset
  - **Decision:** drop (delete) duplicated rows from the dataset
  - **Reason:** likely *operator* error, so these data instances will give unfair weightings within predictions
  - **Method(s) used:** pd.df.drop_duplicates( )

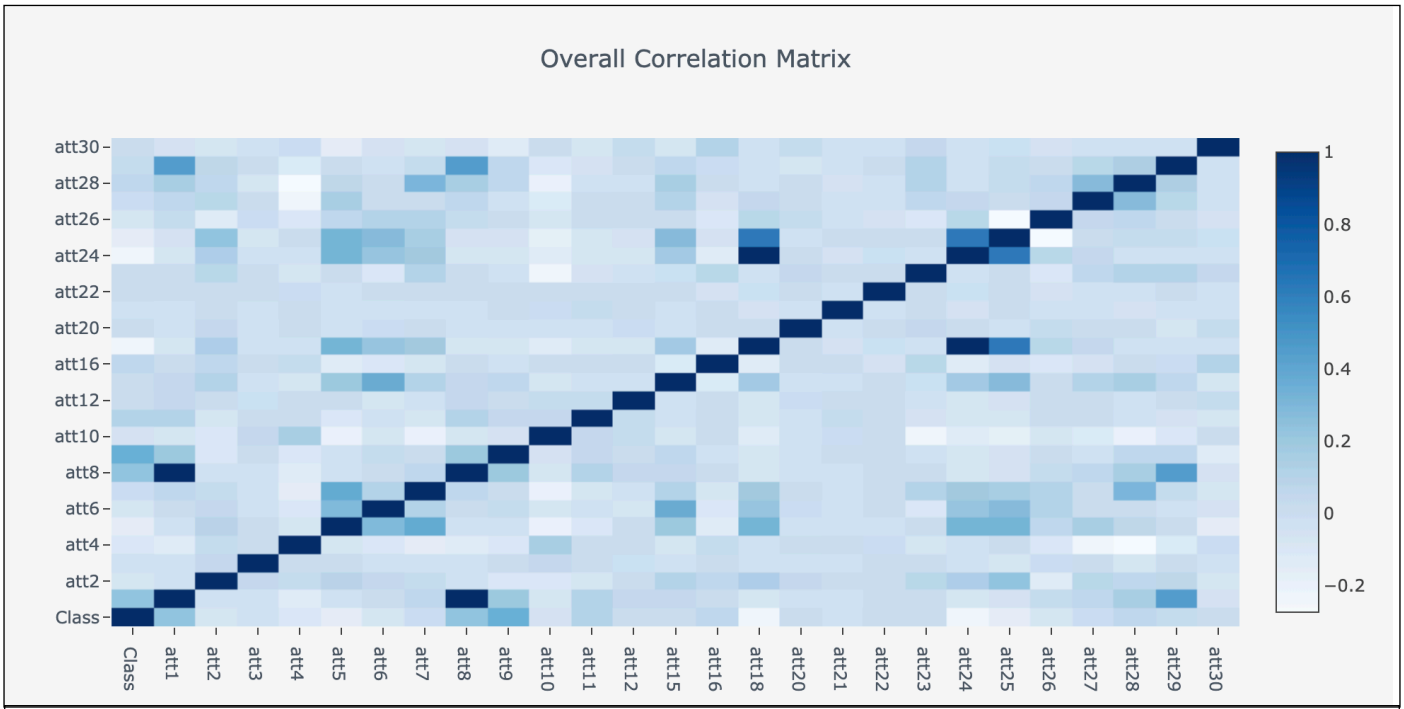

FIGURE 3. CORRELATION MATRIX OF INCLUDED ATTRIBUTES UP TO THIS POINT. NOTICE ATT1 - ATT8 AND ATT18 - ATT24 (AND VICE-VERSA) HAVE DARK BLUE SQUARES (PERFECT CORRELATION); ONE OF EITHER OF THESE PAIRS IS DELETED.

Data Types

Ensuring that each feature within your dataset is being considered as its proper datatype is perhaps the most important process within data cleaning. If not correctly dealt with, a statistical model might completely ignore categorical information (due to it not being encoded properly), or outright treat coded categorical variables as if they were numerical in nature. These are only a couple of the vast amount of problems which can be prevented by properly type casting the overall dataset. There is really no reliably efficient and accurate way to identify correct datatypes besides tediously assessing every single parameter. Although an annoying task, it only has to be completed once and then you'll have two lists containing your numeric features and categorical features separately.

To make this process *somewhat* efficient, interactive widgets were used within the jupyter notebook environments. By using the dropdown menu for all attributes, and assessing the structure of the outputted boxplot for each, it is easy to identify which variables are categorical or numeric in nature. All column labels for each group were added into their respective lists and amply named *'to_categorical'* and *'to_numeric.'* As the names suggest, these lists were then parsed into pandas methods to ensure the features are being treated as the correct types. To save time the categorical variables all were split

automatically into their *dummy* format; ensuring each level of the factor variable is assigned to its own column. This process could've been carried out in a variety of different ways, perhaps utilising the unique value counts within each columns, but visually assessing the boxplot of each attribute was deemed the most thorough way of carrying out this process.

- **Feature:** att18, att20, att21, att22, att25, att28
  - **Type:** numeric
  - **Issue(s):** columns must be cast to the correct (numeric) datatype
  - **Decision:** force all attributes to be considered numeric
  - **Reason:** if not considered numeric these features could have incorrect bearings on predictions
  - **Method(s) used:** pd.df.apply(pd.to_numeric, errors='ignore')
    - Knowing all the known numeric columns contained only numbers a shortcut was taken here, the pd.to_numeric function was parsed to the whole dataset, ignoring any errors which would've arose from categorical variables being forced to numeric (all datatypes were then checked afterwards).

- **Feature:** att30, att29, att27, att26, att23, att16, att15, att12, att11, att10, att9, att7, att6, att5, att4, att3, att2, att1
  - **Type:** categorical
  - **Issue(s):** columns must be cast to the correct (categorical) datatype
  - **Decision:** force all known categorical attributes into their *dummy* form
  - **Reason:** all these columns are going to be converted to dummy zeros and ones anyway, so might as well do it as soon as possible to save time
  - **Method(s) used:** pd.get_dummies(df, columns = to_categorical)

FIGURE 4. ATTRIBUTE 21 BEFORE STANDARD SCALING.

Scaling & Standardisation

Scaling and standardisation of variables is an important part of the data analytics process. Variables with large ranges (and values) can dominate variables with smaller ranges (and values) in certain statistical methods; especially ones which utilise Euclidean distance. A choice could be made here to apply min-max scaling across all variables which are numeric, but doing this first could lose a lot of shape in our data, so we choose to do statistical scaling before *min-max* scaling. Looking at the statistical summary of our numeric columns it is obvious that the minimum values in some columns are negative, while the maximums are positive. What this means is the polarity of our values within these columns could hold some intrinsic meaning, so instantly forcing all these columns to a range between zero and one can be quite dangerous.

Instead, an approach which takes more time, but applies more care, was taken. Interactive widgets were utilised to cycle quickly through all our known numeric columns, but this time histograms were used. Looking at the data in this manner it was obvious that *'att20,' 'att21'* and *'att22'* are all normally distributed. The *'att18'* was less obvious, but it was assumed to be normal too, but just lacking in samples across the range which forces it to look skewed. As a result, all of this features were forced to be normal utilising sklearn's standard scaler.

This still left *'att25'* and *'att28'* to be considered for scaling. When looking at these distributions it's obvious that they're not normally distributed, they are much closer to a skewed log-normal distribution, or something quite similar. As a result, a decision was made to utilise sklearn's robust scaler in order to preserve the shape of the distribution but still scale the results between the desired zero and one range.

As a final check all the histograms were again checked using the interactive widgets within jupyter notebooks. They showcased that the distributions had nearly exactly the same shape, but their ranges are now between zero and one as was desired. We already know all of our categorical variables have been put into their *dummy* forms, meaning they exist along the correct scale, thus we can conclude the overall scaling/standardisation section of the analysis.

- **Feature:** att18, att20, att21, att22
  - **Type:** numeric
  - **Issue(s):** columns must be scaled/standardised
  - **Decision:** force all columns to a new Gaussian (normal) distribution set between zero and one

- **Reason:** assessing the histograms of all of these features, they looked almost identical to the normal distribution - just scaled differently
- **Method(s) used:** sklearn.preprocessing.StandardScaler( ) then .MinMaxScaler( )

- **Feature:** att25, att28
  - **Type:** numeric
  - **Issue(s):** columns must be scaled/standardised
  - **Decision:** force all columns to a new distribution using the column median and quantiles
  - **Reason:** these column histograms weren't so normal, they were skewed, thus an approach which scaled them to this zero/one range was necessary, while still preserving their shape
  - **Method(s) used:** sklearn.preprocessing.RobustScaler( ) then .MinMaxScaler( )

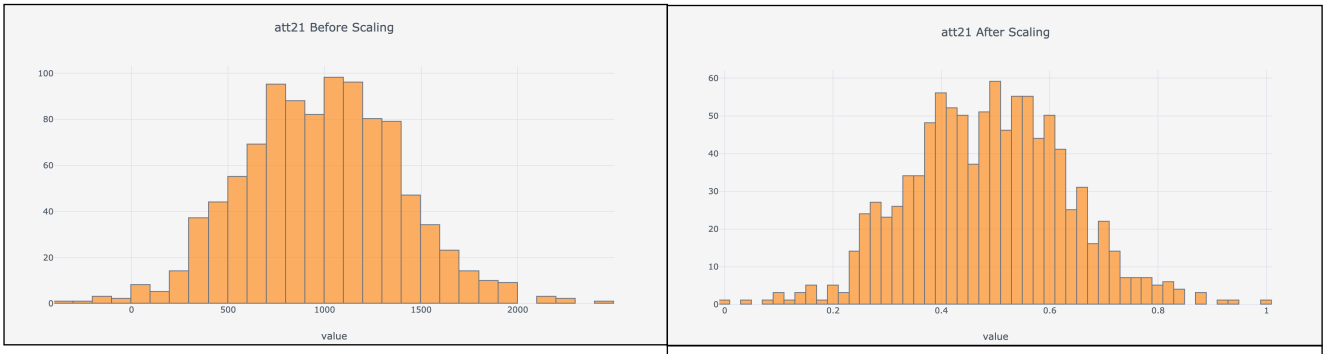

FIGURE 5. ATTRIBUTE 21 AFTER STANDARD SCALING.

Data Imbalance

Having an imbalanced dataset, especially with any classification application, can be detrimental to your model accuracy and robustness. Through simply counting the amount of zeros and the amount of ones within our target 'Class' we can see a clear imbalance does indeed exist. There're 650 rows which correspond to predictions of class one, but there're only 250 rows which correspond to class zero. What this means is less than 30% of the overall dataset is representative of class zero - a major imbalance.

To cure this imbalance the choice has been made to upsample from the minority class (class zero). This decision was made because to reduce the majority class in this case we'd be losing a significant amount of information. Although upsampling from 250 to 650 is a significant jump, the variation seen within the minority class shows that the upsampling wouldn't yield *'plain'* results.

- **Rows:** see analysis for full list; 250 rows total
  - **Type:** rows (not columns), so mixed in types
  - **Issue(s):** class imbalance
  - **Decision:** these rows will be randomly sampled to upscale until the minority class is of ample size
  - **Reason:** class imbalance will lead to dominated predictions from the majority class
  - **Method(s) used:** sklearn.utils.resample(minority_class, replace = True, n_samples = (650 - 250))

Feature Engineering

Feature engineering is a process which usually takes place making heavy use of domain expertise within the problem/prediction space. As the intrinsic meaning of all the variables is unknown to us there really isn't much logical feature engineering we can do. If, for instance, we knew that one of our variables was time taken to run a race, if we knew the distance of the race we could recalculate the column to represent velocity. However, as explained, we have no idea what any of our columns represent so any feature engineering efforts would likely be a lost cause; perhaps causing more harm than good.

Feature Selection

Feature selection is a very important part of the overall model fitting process. If we don't trim down the amount of variables that we're putting into our model then any model built runs the risk of being severely overfit. For most statistical methods, as you add more variables into the mix they're bound to explain

some amount of variance with respect to the predictive class. This becomes a problem because these variables seemingly have little to no effect in predicting our target class, but they're still being considered in mass. In general, it is a much better idea to take a smaller subset of your total feature space, only utilising the features which show the highest importance with respect to the target class. Following this process ensures that any model you build will be robust to unseen data, newer trends, and not lack all generalisation because it's *'hugging'* too close/tightly to the data seen during training.

Three methods of feature selection were carried out during this analysis. By ordering the results and plotting the explained variance respective to each variable it was quite clear that a subset of variables seemed to be important. The amount of explained variance quickly tapers off to a point where all the variables seem to be showcasing, roughly, the same amount of importance; these features are likely *'noise'* when we consider our prediction problem.

The first method tests only the univariate cases of importance. What this means practically is each of our features is compared against the target class and given a measure of importance when judged individually in this manner. This method does not take into consideration joint effects with respect to the target class, but it's still a good starting point for looking at overall importance. When looking at the deterioration of importance, a subjective choice was taken to carry through only the top 15 most important features according to this test.

The second method for exploring feature importance is perhaps the most popular; tree-based feature selection. In this method we've fit the default tree-based model, keeping to the 100 estimator default parameter. The same process was followed by plotting the feature importances in order with respect to each variable. In this attempt the deterioration of explained variance was much steeper, so a subjective decision to take only the top 11 most importance features was made.

The last method of finding important features was to exploit the properties of either logistic regression or linear support vector machines. The property being that unimportant features tend towards zero. By fitting a model and then setting a specific threshold we can return the list of important features with respect to this given threshold. These results weren't plotted like the last two, but the set of considered important features were added to the overall running list.

The full list of features was 79 (after all the categorical variables were expanded to their dummy forms), this number has been reduced to 33 features which seem to have a high amount of importance; the full list follows…

att9_D, att1_V4, att9_A, att11_C, att3_V4, att1_V1, att5_V4, att1_V2, att1_V0, att7_V1, att11_A, att25, att18, att10_A, att2_V5, att21, att9_B, att2_V1, att22, att20, att2_V4, att26_4, att7_V2, att3_V3, att26_2, att3_V2, att7_V3, att28, att10_B, att5_V1, att4_V4, att3_V1, att4_V2.

- **Method(s) used:**
  - sklearn.feature_selection.SelectKBest( )
  - sklearn.feature_selection.chi2( )
  - sklearn.feature_selection.SelectFromModel( )
  - sklearn.ensemble.ExtraTreesClassifier( )
  - sklearn.svm.LinearSVC( )

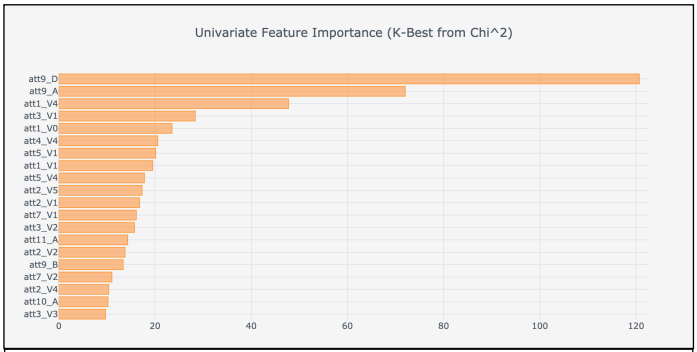

FIGURE 5. A SUBSET OF MOST IMPORTANT FROM THE UNIVARIATE FEATURE SELECTION PROCESS.

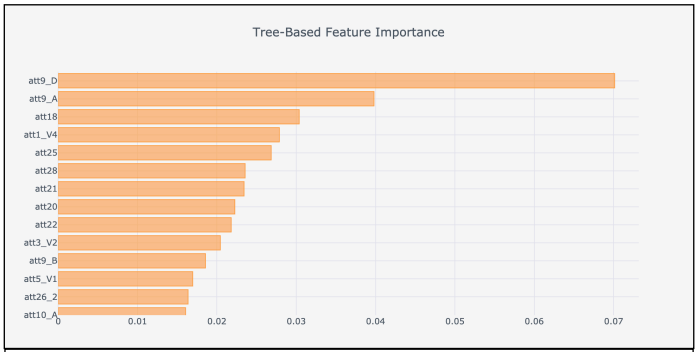

FIGURE 6. A SUBSET OF MOST IMPORTANT FROM THE TREE-BASED FEATURE SELECTION PROCESS.

Train, Test & Validation Sets

Splitting your data up into training, testing and validation sets is integral to the model building process. If you have no test data set aside, your model will be impossible to evaluate and likely be completely overfit if you tried to put it into practice. Validation sets are important for tuning your models, by utilising cross validation we can have multiple validation sets which all serve to make the model the best it can possibly be with respect to its tuneable parameters.

The overall dataset contains 1300 trainable rows, along with the 100 final rows which are set aside for the final predictions which are to be handed in. The decision here has been to split the 1300 trainable rows into a training and testing set using a 80/20 split. This leaves us with a training set with 1040 instances and a testing set of 260 instances. The validation sets will be created using the k-folds method during cross validation. This means that we can tune our model using the validation sets, make sure it gets the best predictions on our testing set, and then create predictions on the last 100 rows which were set aside for the best two models & their predictions.

- **Method(s) used:**
  - sklearn.model_selection.train_test_split( )
  - X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20, random_state=42069)

Data Classification

Classifier Selection

The advantage of using a framework as popular as sklearn means that as soon as we have our data in matrix form, split into training and testing, it makes it trivial to fit a vast amount of different classifiers. The set of classification models which are to be fit and tested to this dataset are listed below.

- K-Means Classifier
- Random Forest Classifier
- Logistic Regression
- Support Vector Machines
- Naive Bayes Classifier
- AdaBoost Classifier
- K-Nearest Neighbour Classifier

All of these classifiers will be tested using cross-validation to trial different parameter values and attempt to tune them to their best values. The parameter tuning mostly took place through a rather manual process. A pipeline was created with the various sets of classifiers and the possible set of parameters contained within. Sklearn then iterates through this pipeline and returns the model with the best performance metrics with respect to the validation set in this case. This process was repeated for many different combinations of models and parameters; the code is available but commented out within the overall analysis. These models are then tested on the 20% of data which was set aside (260 rows) and various evaluation metrics are calculated; including (but not restricted to) a confusion matrix, accuracy score and f1-scores. Another metric which will be taken considered, but taken with a grain of salt, is the proportion of zero and one predictions across the final test set (last 100 rows). We've been told in the assignment sheet that out of the final 100 predictions, half of them should be zeros and half of them should be ones; thus this metric will be used for evaluation. All of these results will be discussed during the classifier comparison section which follows within this report.

- **Method(s) used:**
  - sklearn.cluster.KMeans( )
  - sklearn.ensemble.RandomForestClassifier( )
  - sklearn.linear_model.LogisticRegression( )
  - sklearn.svm( )
  - sklearn.naive_bayes.GaussianNB( )
  - sklearn.ensemble.AdaBoostClassifier( )
  - sklearn.neighbors.KNeighborsClassifier( )
  - sklearn.model_selection.GridSearchCV( )
  - sklearn.pipeline.Pipeline( )

Cross Validation

Cross validation is again, something that's made quite trivial when utilising the sklearn framework. Once you've created your model you simple parse that model into a function, along with your training/testing sets, then you just define the amount of folds you wish to create along with the performance measure you'd like to receive back. It was decided that a folding size of 10 would be sufficient in our analysis. This results in having 1040 training instances to work with, which means that during the cross validation process we are also setting 104 instances aside as a test set. The process is then repeated for a total of 10 iterations, calculating a performance metric of your choice in each run. As this process is rather quick to process we will run k-folds cross validation to generate an accuracy score, along with generating an f1-score on a consecutive second run. This process will be completed across all the discussed models, and then the results will be compared and discussed in the following classier comparison section.

- **Method(s) used:**
  - sklearn.model_selection.cross_val_score( )
  - sklearn.model_selection.cross_val_predict( )

Classifier Comparison

As explained in earlier sections, k-folds cross validation has taken place (folds = 10) across all the models, which generates a set of 10 accuracy and f1 scores for each of the models. A boxplot has been generated for each model's accuracy and f1 scores, which can be seen below in *figure 7*. From a glance it is clear that the k-means classifier has performed overall worst, having an accuracy score which is sometimes lower than 50% - meaning a blind guess would be better in a vast amount of cases. However, all the other models seem to fluctuate within a pretty tight band from 70% to 80% in both accuracy and f1 measures.

The stand out two models seem to be the support vector machine and the k-nearest neighbour models. The support vector machine seems to reach even into the low 80% range, but this seems to be balanced out with some heavy dips into the low 70% range, verging on the edge of high the 60% range. Running somewhat counter to this we see the k-nearest neighbour classifier which has an even higher accuracy measure, comparable f1 measure, and they both seem to exist in a much tighter, more consistent, band centred around the upper 70% and low 80% ranges.

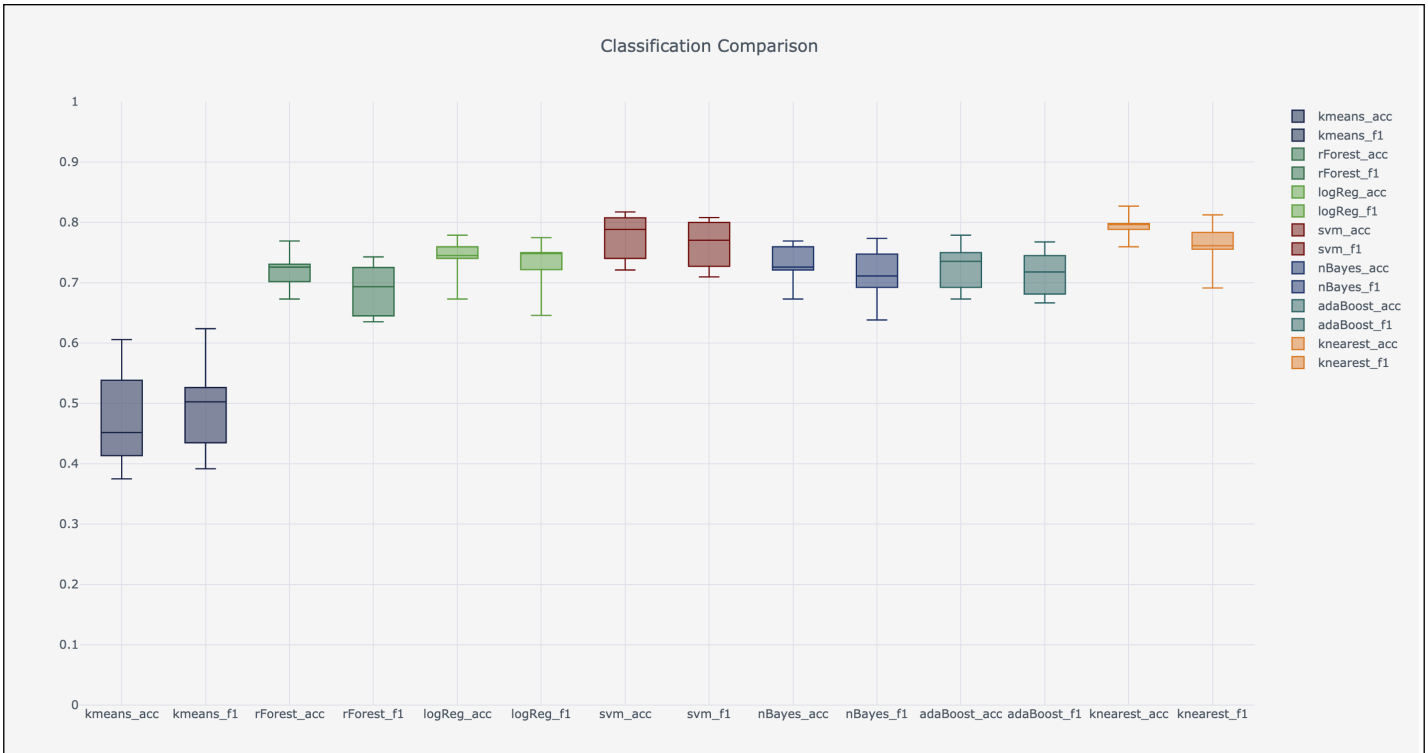

FIGURE 7. BOXPLOT COMPARISON OF EVERY MODELS CROSS-VALIDATED ACCURACY & F1 METRICS. FROM LEFT TO RIGHT WE HAVE: K-MEANS, RANDOM FOREST, LOGISTIC REGRESSION, SVM, NAIVE BAYES, ADABOOST & K-NEAREST NEIGHBOURS.

Looking just at this visualisation, I would score the k-nearest neighbour model as the overall best performer due to its consistency. The support vector machine model comes in at a close second due to its performance reaching into similar high ranges. As for the rest of the models, they seem to be overall quite comparable. The logistic regression model would probably be the next choice due to its easy interpretability, followed by the naive Bayes model for the exact same reason. Next would come the AdaBoost classifier due to it just performing slightly better than the random forest. Lastly, we can't forget the k-means classifier which comes in far behind the rest of the models (discussed prior).

This isn't the end of our evaluation though, we still have that vital piece of information from the assignment sheet which states that within the final predictions there should be a 50/50 split between each of the target classes. Predictions were run again, using these same models, but this time using the final 100 rows of data to make the predictions on the missing target class values. It's unclear which values should specifically be ones or zeros, but we can calculate the proportion of each, and how far, or close, from this 50% mark the proportions are.

	kmeans	rForest	logReg	svm	nBayes	adaBoost	knearest
n_zeros	41.00	44.00	43.00	41.00	44.00	39.00	49.00
n_ones	59.00	56.00	57.00	59.00	56.00	61.00	51.00
difference	0.18	0.12	0.14	0.18	0.12	0.22	0.02
correct	0.82	0.88	0.86	0.82	0.88	0.78	0.98

TABLE 3. TARGET CLASS VALUE COUNT FROM LAST 100 ROW, FINAL PREDICTIONS.

As can be seen within *table 3*, the k-nearest neighbour model predicted 49 zeros and 51 ones in the final 100 rows, this is the closest to the 50% split from all the tested models. Second closest seems to be a tie between random forest and naive Bayes models. But, perhaps what is most interesting result is that the support vector machine model predicted the same proportion as our worst (until now) k-means classifier. It is not uncommon for a support vector machine, especially with a linear kernel, to create a rather unsophisticated cut-off point which is highly dependent (overfit) on the training data; this could explain the distance from the perfect 50% split. Furthest away is our AdaBoost model which, as boosting trees often do, seems to have focused too heavily on a trend within the training data which hasn't generalised well to practical predictions.

It is of course understood that purely just because a model doesn't have the perfect 50% split doesn't mean it's a bad model. In fact, if we consider the AdaBoost model as only misclassifying these known 11 values, that places it at 91% accuracy! However, true-negatives and false-positives will of course cancel each other out and make it look like the accuracy is better than it is - when approaching evaluation from this crude metric. It is an extra piece of information which does help us assess, at least roughly, different model's ability to generalise.

Overall this information helps to paint a slightly different picture of the best and worst models when compared to just the previous visualisation. The k-nearest neighbour model has definitely cemented itself in first place; just like the k-means model has cemented itself in last. However, the support vector machine can't quite be trusted due to its variability here, so I'd say second place is now taken by the much more consistent, and interpretable, logistic regression model. The breakdown of all models from best (first) to worst (seventh) can be seen below.

1. K-Nearest Neighbour Classifier
2. Logistic Regression
3. Naive Bayes Classifier
4. Random Forest Classifier
5. Support Vector Machine
6. AdaBoost Classifier
7. K-Means Classifier

Thus, the two models which will be giving my final predictions will be the k-nearest neighbour model along with the logistic regression model.

Prediction

As mentioned previously, the k-nearest neighbour and logistic regression models are the models will be making the final predictions on the last 100 rows. For the **k-nearest neighbour model** my best estimate of the accuracy comes from my median result during cross-validation which is **80%**. For the **logistic regression model** my best estimate of the accuracy again comes from the median result during cross-validation which was **75%**. The exact predictions which can be found in the predict.csv which will be stored within this project directory. To ensure consistency and accountability a set of screenshots of the overall prediction set follows.

ID	Predict 1	Predict 2	1021	1	1	1041	1	0	1061	0	1	1081	1	1
1001	1	0	1022	1	1	1042	0	0	1062	1	1	1082	1	1
1002	0	0	1023	1	1	1043	0	1	1063	0	0	1083	1	0
1003	0	0	1024	1	1	1044	1	1	1064	0	0	1084	1	1
1004	0	0	1025	0	1	1045	1	1	1065	0	0	1085	1	0
1005	0	0	1026	1	1	1046	0	0	1066	1	1	1086	0	1
1006	1	1	1027	0	0	1047	0	0	1067	1	1	1087	1	0
1007	1	1	1028	0	0	1048	0	0	1068	1	1	1088	0	0
1008	0	0	1029	1	1	1049	1	1	1069	1	1	1089	1	1
1009	0	1	1030	1	1	1050	0	0	1070	0	1	1090	1	1
1010	0	0	1031	1	1	1051	1	1	1071	1	1	1091	1	1
1011	0	1	1032	1	1	1052	0	0	1072	0	0	1092	0	0
1012	1	1	1033	1	1	1053	0	1	1073	0	0	1093	1	1
1013	1	1	1034	0	0	1054	0	1	1074	1	1	1094	0	1
1014	0	1	1035	0	0	1055	1	1	1075	0	0	1095	0	0
1015	0	1	1036	1	1	1056	1	1	1076	1	1	1096	0	0
1016	0	0	1037	0	0	1057	1	1	1077	1	1	1097	0	0
1017	0	0	1038	0	0	1058	1	1	1078	1	1	1098	0	0
1018	0	0	1039	0	0	1059	1	1	1079	1	1	1099	0	0
1019	1	1	1040	1	1	1060	0	0	1080	1	0	1100	1	1
1020	0	0												

TABLE 4. FINAL 100 ROW PREDICTION BREAKDOWN; PREDICT 1 IS K-NEAREST NEIGHBOURS, PREDICT 2 IS LOGISTIC REGRESSION.

If you for some reason need to copy and paste the full series for each prediction they are listed below. As mentioned prior, predict 1 is representative of the k-nearest neighbour mode and predict 2 represents the logistic regression model.

Predict 1: [1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1, 1, 0, 1, 0, 0, 1, 1, 1, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1]

Predict 2: [0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 0, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 1, 0, 1]

References

Brownlee, J. (2019). *Feature Selection For Machine Learning in Python*. [online] Machine Learning Mastery. Available at: https://machinelearningmastery.com/feature-selection-machine-learning-python/ [Accessed 28 Sep. 2019].

Brownlee, J. (2019). *How to Tune Algorithm Parameters with Scikit-Learn*. [online] Machine Learning Mastery. Available at: https://machinelearningmastery.com/how-to-tune-algorithm-parameters-with-scikit-learn/ [Accessed 28 Sep. 2019].

GitHub. (2019). *santosjorge/cufflinks*. [online] Available at: https://github.com/santosjorge/cufflinks [Accessed 28 Sep. 2019].

Hale, J. (2019). *Scale, Standardize, or Normalize with Scikit-Learn*. [online] Medium. Available at: https://towardsdatascience.com/scale-standardize-or-normalize-with-scikit-learn-6ccc7d176a02 [Accessed 28 Sep. 2019].

Pandas.pydata.org. (2019). *pandas 0.25.1 documentation*. [online] Available at: https://pandas.pydata.org/pandas-docs/stable/ [Accessed 28 Sep. 2019].

Scikit-learn.org. (2019). *Classifier comparison*. [online] Available at: https://scikit-learn.org/stable/auto_examples/classification/plot_classifier_comparison.html [Accessed 28 Sep. 2019].

Scikit-learn.org. (2019). *Feature selection*. [online] Available at: https://scikit-learn.org/stable/modules/feature_selection.html [Accessed 28 Sep. 2019].

Scikit-learn.org. (2019). *Parameter estimation using grid search with cross-validation*. [online] Available at: https://scikit-learn.org/stable/auto_examples/model_selection/plot_grid_search_digits.html [Accessed 28 Sep. 2019].

Scikit-learn.org. (2019). *Tuning the hyper-parameters of an estimator*. [online] Available at: https://scikit-learn.org/stable/modules/grid_search.html [Accessed 28 Sep. 2019].

Shetye, A. (2019). *Feature Selection with sklearn and Pandas*. [online] Medium. Available at: https://towardsdatascience.com/feature-selection-with-pandas-e3690ad8504b [Accessed 28 Sep. 2019].

Van Dorpe, S. (2019). *Preprocessing with sklearn: a complete and comprehensive guide*. [online] Medium. Available at: https://towardsdatascience.com/preprocessing-with-sklearn-a-complete-and-comprehensive-guide-670cb98fcfb9 [Accessed 28 Sep. 2019].

Appendix

For convenience, the Jupyter notebook is being hosted online at the following link. This is of course not the entire analysis, and every step taken, but it is the *clean* version which contains most of the crucial information necessary to get from data input all the way to prediction output.

https://yortug.github.io/data_mining_assignment/data_mining_assignment.html

To ensure continuity the *README.txt* file is displayed below. It explains not just the instructions necessary to reproduce the analysis, but also the overall file structure of the project directory, along with an explanation surrounding the supplementary resources. You can also view this *README.txt* online, with all the accompanying resources at the following link.

https://github.com/yortug/data_mining_exercise

```
--------------------------------------------------------------------------------

AUTHOR: TROY ENGELHARDT
STUDENT ID: 18815179
DATE: 2019-09-24

--------------------------------------------------------------------------------

FOR YOUR CONVENIENCE THE KNITTED JUPYTER NOTEBOOK IS BEING HOSTED ONLINE HERE:

https://yortug.github.io/data_mining_assignment/data_mining_assignment.html

AND THE FULL REPOSITORY IS BEING HOSTED HERE:

https://github.com/yortug/data_mining_exercise

--------------------------------------------------------------------------------

PRELUDE

--------------------------------------------------------------------------------


THE FOLLOWING IS A TREE-STRUCTURE OVERVIEW OF THIS PROJECT DIRECTORY. THE MAIN
FILES TO CONSIDER ARE THE ANALYSIS.PY, PREDICT.CSV AND REPORT.PDF FILES. THE
ANALYSIS.PY FILE ONLY HAS NUMPY, PANDAS AND SKLEARN DEPENDENCIES. THE OTHER
LARGER NOTEBOOK ANALYSIS FILES HAVE EXTERNAL DEPENDENCIES, BUT THEIR RESULTS
CAN BE SEEN IN THE KNITTED .HTML OR .PDF FILES AS YOU PLEASE. THE REPORT.PDF
FILE CONTAINS THE OVERALL WRITE-UP OF THE ASSIGNMENT, IT SHOULD BE OF PRIMARY
FOCUS, SUPPLEMENTED BY THE ANALYSIS.PY FILE, AND THEN OVERALL ACCURACY CAN BE
MEASURED USING THE PREDICT.CSV FILE. ALL OTHER FILES ARE SUPPLEMENTARY.

.
├── README.txt
├── assignment_resources
│   ├── assignment2019.pdf
│   ├── assignment2019_updated.pdf
│   └── declaration_of_originality\ [signed].pdf
├── comp3009_assignment_notebook.ipynb
├── data2019.student.csv
├── notebooks
│   ├── comp3009_assignment_notebook.html
│   └── comp3009_assignment_notebook.pdf
├── predict.csv
├── report.pdf
├── run.py
├── train_test_val
│   ├── arff
│   │   ├── test.arff
│   │   ├── train.arff
│   │   └── val.arff
│   ├── csv
│   │   ├── test.csv
│   │   ├── train.csv
│   │   └── val.csv
│   └── final_100_set.csv
└── visualisations
    ├── 1.\ boxplot_att25.html
    ├── 2.\ boxplot_att28.html
    ├── 3.\ boxplot_att20_numeric.html
    ├── 4.\ boxplot_att12_categorical.html
```

```
    ├── 5.\ hist_att21_before_scale.html
    ├── 6.\ hist_att21_after_scale.html
    ├── 7.\ hbar_univariate_feature_importance.html
    ├── 8.\ hbar_tree_feature_importance.html
    └── 9.\ boxplot_classification_comparison.html
```

MORE INFORMATION

EXAMPLE VISUALISATIONS FROM THE ANALYSIS CAN BE FOUND IN THE VISUALISATIONS DIRECTORY. FAR MORE WERE GENERATED DURING ANALYSIS USING INTERACTIVE WIDGETS, THE CODE FOR WHICH CAN BE FOUND COMMENTED OUT IN THE ACCOMPANYING NOTEBOOKS.

THE TRAINING, TESTING AND VALIDATION SETS CAN BE FOUND WITHIN THE TRAIN_TEST_VAL DIRECTORY, CONTAINING SUB-DIRECTORIES FOR BOTH ARFF AND CSV FORMATS. THE ARFF FILES WERE CONVERTED FROM THEIR PANDAS DATA FRAMES USING A FUNCTION FOUND ONLINE, HOSTED ON GITHUB UNDER THE MIT LICENCE.

https://github.com/saurabhnagrecha/Pandas-to-ARFF/blob/master/pandas2arff.py

THE ONLY DIRECTORY WHICH HASN'T BEEN MENTIONED IS THE NOTEBOOKS DIRECTORY. THIS FOLDER CONTAINS TWO KNITTED VERSIONS OF THE NOTEBOOK USED FOR ANALYSIS, FORMATTED TO PDF AND HTML FORMAT FOR YOUR CONVENIENCE. THIS IS DONE SO YOU DON'T HAVE TO RUN THE OVERALL NOTEBOOK ON A LOCAL SERVER JUST TO SEE THE ANALYSIS/OUTPUT GENERATED THERE.

INSTRUCTIONS TO RUN THE CODE

THE BEST WAY TO RUN THE CODE IS AS FOLLOWS:

1. OPEN YOUR TERMINAL
2. NAVIGATE INTO THE PROJECT DIRECTORY
3. ENTER INTO TERMINAL:
   > python run.py
4. WAIT APPROXIMATELY 10 TO 30 SECONDS FOR EXECUTION TO COMPLETE
5. OBSERVE THE STRING RESULTS OUTPUT TO YOUR TERMINAL
   I. TABLE OF ACC/F1 (DURING CV) FOR ALL TESTED MODELS
   II. TABLE OF PROPORTION OF PREDICTED 0/1'S ON THE FINAL 100 ROWS
   III. TABLE OF THE SET OF PREDICTIONS OF THE FINAL 100 ROWS

NOTE: EXECUTING THIS CODE WILL NOT OUTPUT ANY FILES. THIS IS BY DESIGN. DUE TO THE RANDOM ELEMENTS OF SPECIFIC MODELS, PREDICTIONS CAN CHANGE. YOU CAN, IF YOU WANT, ADD THE .TO_CSV('FILENAME') METHOD TO THE END OF THE OUTPUT FILES TO TEST THEM YOURSELF. HOWEVER, IT CAN CLEARLY BE SEEN WITHIN THE CODE, AND WITHIN THE NOTEBOOKS THAT THE PREDICTIONS FILE PRESENT HAS BEEN GENERATED FROM THE SETS OF MODELS WHICH WERE PROVIDED.

ALTERNATIVE INSTRUCTIONS

YOU CAN RUN THE FOLLOWING COMMAND WHILE INSIDE THE MAIN DIRECTORY TO RUN THE JUPYTER NOTEBOOK SERVER AND DYNAMICALLY VIEW THE IPYNB FILE.

```
    > jupyter notebook
```

THEN YOU CAN SIMPLY SELECT `CELLS` AT THE TOP, AND CLICK `RUN ALL` TO RUN ALL THE CELLS WITHIN THE NOTEBOOK. YOU CAN THEN SCROLL THROUGH THE DOCUMENT AND MAKE CHANGES WHERE EVER YOU PLEASE.

HOWEVER, THIS OF COURSE HAS A DEPENDENCY ON THE JUPYTER PACKAGE. AN ALTERNATIVE IS TO SIMPLY VIEW THE EXPORTED HTML AND PDF FILES WITHIN THE NOTEBOOKS FOLDER. THIS DOES NOT REQUIRE THE WEB SERVER AND ANALYTICS PLATFORM TO BE RUN LOCALLY.

CLOSING

IF YOU HAVE ANY PROBLEMS RUNNING THE FILES, PLEASE DON'T HESITATE TO CONTACT ME AT 18815179@STUDENT.CURTIN.EDU.AU! THESE INSTRUCTIONS HOPEFULLY SUFFICE.
