MET CS699 – Project Assignment

Final Report

Brandon Bartol – Chuong Nguyen

1. **Introduction**

Data mining stands as a crucial step in data analysis, particularly in today's era of abundant data availability. Throughout history, humanity has drawn insights from past events to anticipate and differentiate between impending threats and opportunities for societal betterment. However, this process has often been passive, lacking the immediacy needed for timely action. Take, for instance, weather forecasting: once regarded as unpredictable disasters attributed to supernatural forces, technological advancements now enable us to define, explain, and predict natural events with a considerable lead time. This capability hinges on our ability to detect and analyze patterns in data, unveiling meaningful insights.

The primary aim of this project is to develop and assess the performance of various classification models tailored to predict depressive disorders. Our approach involves analyzing a comprehensive dataset featuring a diverse array of demographic, lifestyle, and health-related variables. Utilizing advanced machine learning techniques—such as naïve Bayes, logistic regression, decision trees, random forests, K-nearest neighbors (KNN), and neural networks—we seek to construct robust predictive models adept at accurately discerning individuals with depressive disorders from those without.

Beyond model construction lies a comprehensive comparison of these algorithms. By systematically evaluating performance metrics—such as accuracy, precision, recall, F1-score, and area under the receiver operating characteristic curve (AUC-ROC)—our aim is to pinpoint the most effective classifier for this task. This holistic analysis promises valuable insights into each model's strengths and weaknesses, facilitating informed recommendations for real-world application.

Additionally, we are eager to explore diverse feature selection techniques and sampling methods, particularly in the context of imbalanced datasets. Class distribution imbalance—wherein one class (e.g., individuals with depressive disorders) disproportionately outweighs the other—poses distinct challenges. To address this, we will investigate oversampling, undersampling, and bootstrap data generation methods to rebalance the dataset and alleviate the impact of class imbalance on model performance.

In summary, our project endeavors to develop and assess multiple classification models for predicting depressive disorders based on demographic, lifestyle, and health-related factors. By leveraging machine learning algorithms and exploring various techniques, we aim to enhance the accuracy and efficacy of predictive models, ultimately contributing to early detection and intervention strategies in mental healthcare.

1. **Data Preprocessing**

Before we could begin model building, our first step was to preprocess the data, this is done to ensure the quality our input is of high quality and can be correctly leveraged by our models. The first step in this practice is to identify and remove columns that were missing data for the majority of patients. In our case we chose to remove columns that were missing data more than 20% of the time. This number was chosen because at a greater rate we could not reliably use smoothing methods to fill in the missing data as discussed later. To remove the columns the following code is used:

A computer code with text

Description automatically generated with medium confidence

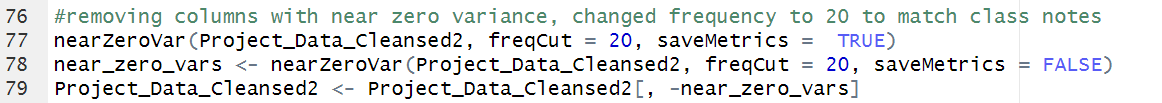
This code is written to count the number of null rows in each column and log the columns that have a value greater than 1000, this corresponds to 20% since our starting dataset is 5000 rows. The columns that were found to be 0 are then dropped.

Next, we attempted to fill in the values that were missing using the other values that were present in the columns. This was done in 2 different ways. Our first method was done when the number of unique values in the column is less than or equal to 5, we consider this column likely categorical data. Because it is likely categorical taking an average would not provide a benefit so instead, we substitute the missing values with the most common value or the mode. The second method is performed when there are more than 5 unique values. We consider this data numeric so we filled in the missing values with the median value. This was done in a stratified way so that if the class is Y then the median or mode of the Y values was used and the same was done for N. The following code represents these actions.

A close up of a computer screen

Description automatically generated

After filling in the missing data values were then removed that had near zero variance. If the variance in data columns is very small then it is unlikely that the column will act as a good predictor because the value is not changing with the class. To help cut down on these cases we used the near zero frequency function with a frequency cut of 20. This means if the first most common value in a column has 20 times the prevalence as the second most frequent column then the value is removed. This is seen in the following code.

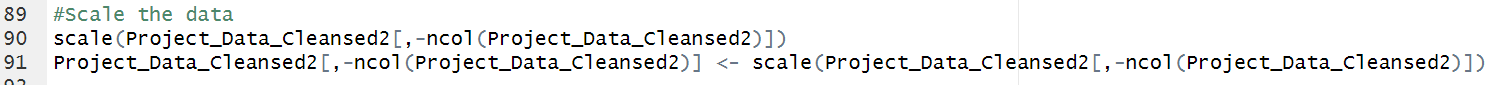


In addition to nearly static data being not helpful for predictions, data that is highly correlated are redundant and not helpful in predations. This means columns that scale together provide the same information for the model and are therefore not helpful and only take up resources unnecessarily. To identify these values we used collinearity analysis to create a Pearson coefficient. When this coefficient is close to zero the values being compared are uncorrelated, when it is less than zero they are negatively correlated, and when the values are greater than 0 and close to 1 then the values are positively correlated. We focused on these positively correlated values and when the value was greater than 0.7 then the two columns being compared were thought to provide the same information. We were then able to remove one of these columns in the pair from the dataset to reduce the number of columns but maintain the prediction information. These are the steps performed by this following code.

A computer code with text

Description automatically generated

At this point the data had been cleansed so that missing values were removed or filled in and unhelpful columns were removed. The last step in this section is to scale the data. Scaling the data means resize the data so that is all in the same scale. In our case we chose to scale so that every column has a mean that is represented by zero and standard deviation is equal to 1. By doing this the values all have the same weight meaning large numbers will no longer disproportionally affect the models, rather it is the difference from the mean that will determine their significance. The significance each column has on the models will now be the same for all.



With this final section of code the preprocessing section of the assignment was completed.

1. **Feature Selection and Imbalanced Data Treatment**
   1. Splitting the data into training and testing sets

The next step after preprocessing is splitting the data. To do this the Class column was made a factor with values “Y” or “N”. Next, the split function was used to split the data into a training and test dataset. The data was split so that 80% of the 5000 initial rows are used as training data while the remaining 20% will be used as test data. Often times the split will be at 66% but in order to maintaining enough values in our training dataset 80% was used. This is seen as the prop level in the following code.

**A computer code with text

Description automatically generated**

After the data is split the feature selection functions were created. The feature selection functions are run on our training data to further identify the columns that we want to focus on.

* + 1. Boruta

The first feature selection function is Boruta. The input for this function is the training dataset from the previous split. This works by entering our data frame into the Boruta function and identifying our target column, in this case “Class”. Boruta runs and will output a “finalDecision” for each column which corresponds to its importance in predicting the output class. We then identify which columns were identified as “Rejected” and remove these columns from our data frame. This function returns a list of the Rejected column names which can be used to remove these.

A computer code with many colored text

Description automatically generated with medium confidence

* + 1. Information Gain

The second feature selection method we chose to used is information gain. The “info\_gain\_selection” function was written and takes the training data frame as its input. The “information.gain” function is then applied to this data frame and our “Class” column is selected as the target variable. The output of the “information.gain” function is then transformed into a table containing 2 columns, the list of Attributes and the corresponding “Info Gain”. This table is sorted in decreasing order by “Info Gain”. The info gain helps to quantify the amount of information gained about the resulting class through the selected Attribute. Elements with high information gain can be seen as the most important. The next part of our code sums the “info gain” column and creates a “target\_sum” that corresponds to half of the total value of info gain. By taking a cumulative sum on the “Info Gain” column we figure out the number of attributes that correspond to half the information gain and these are the columns we want to select. These are selected because they are the most important in provide the most information when predicting the class. 50% of the information gain was chosen because it allows us to select the most important columns while leaving out ones that do not contribute to the majority of the information. Now that our table only contains the Attributes that are most important, it is turned into a list of Attributes. This list of column names is then used to subset our data frame so only the highest information gain columns are now shown.

A screenshot of a computer program

Description automatically generated

* + 1. Principal Component Analysis (PCA)

The last feature selection method chosen is PCA or Principal Component Analysis. The training data is input to the PCA function and this returns a table containing a Principal Component for each non-Class attribute. Each principal component includes a composite of the predictor attributes and these can be summarized by their standard deviation, variability, and cumulative variability. We are focused on the cumulative variability this corresponds to the percent of total variability these components are responsible for. We then take the number of principal components that correspond to 90% of the total variability and return these principal components. A for loop is sued on the selection part of this function so that our main split has PCA applied to both the train and test dataset, it is also applied to are secondary split so this training and testing data frame also have PCA applied to their non-Class columns.

A screenshot of a computer program

Description automatically generated

* 1. Imbalanced data treatment methods:
     1. Under-sampling and Over-sampling from ROSE library

This severe class imbalance in the dataset presents significant challenges for the classification models to effectively learn and generalize from the data. With a ratio of 1 to 100 between participants with depressive disorder and those without the condition, the minority class (participants with depressive disorder) is heavily underrepresented, leading to several issues. Firstly, models trained on imbalanced data tend to exhibit a bias towards the majority class, as they prioritize maximizing overall accuracy without adequately capturing patterns within the minority class. This results in poor classification performance, especially in accurately identifying instances belonging to the minority class. Additionally, imbalanced data can lead to decision boundaries being skewed towards the majority class, making it challenging for models to distinguish between the two classes accurately. As a result, the predictive power of classification models is significantly compromised, highlighting the critical importance of addressing class imbalance through appropriate data sampling techniques and model evaluation strategies. (Aguiar, 2024)

To tackle the issue discussed above, there are several balancing treatment methods, including oversampling, undersampling, a combination of both, and bootstrap resampling were implemented. Each of these methods aims to mitigate the challenges posed by class imbalance and enhance the performance of classification models in predicting depressive disorders based on demographic, lifestyle, and health-related factors.

The first method, over-sampling, involves increasing the number of instances in the minority class to achieve a balanced class distribution. In our implementation, the over\_balance function utilizes the ROSE package's ovun.sample function with the "over" method to generate additional instances for the minority class, effectively increasing its representation in the dataset.

Conversely, undersampling reduces the number of instances in the majority class to achieve balance. The under\_balance function employs the "under" method in the ovun.sample function to randomly select a subset of instances from the majority class. While effective in balancing the dataset, this method may lead to the loss of valuable information present in the majority class.

To comprehensively address class imbalance, a combination of over-sampling and under-sampling techniques is employed. The combine\_sampling\_balance function simultaneously generates synthetic instances for the minority class and downsamples the majority class using the "both" method in the ovun.sample function. By leveraging the strengths of both methods, this approach aims to create a balanced dataset while minimizing information loss.One note is that all three above methods, over-sampling, under-sampling, and combination of both utilized ROSE package in R in R 4.2.1 environment.

* + 1. Bootstrap Resampling

Finally, bootstrap resampling involves creating balanced datasets by repeatedly sampling instances from the minority class with replacement. The bootstrap\_balance function implements this method by randomly selecting instances from the minority class and combining them with instances from the majority class. Unlike traditional over-sampling techniques, bootstrap balancing relies on resampling existing instances, potentially preserving the original distribution more accurately. There is one parameter to determine the target ratio between the majority and minority and it was set to 1 to 1 in this project.

These balancing treatment methods are essential in mitigating the impact of class imbalance on classification model performance. By addressing class distribution disparities, these techniques enable more robust and unbiased model training, ultimately improving the model's predictive power in identifying individuals with depressive disorders.

1. **Model Building**
   1. Splitting the training and testing sets

Splitting data into training and testing sets is crucial in machine learning for several reasons. Firstly, it allows us to evaluate the performance of our model on unseen data, simulating how it would perform in real-world scenarios. By reserving a portion of the data for testing, we can assess how well the model generalizes to new instances, providing insights into its ability to make accurate predictions beyond the training data. Moreover, splitting the data helps prevent overfitting, where the model learns to memorize the training data rather than capturing underlying patterns. By training the model on one set of data and testing it on another independent set, we can identify and mitigate overfitting, ensuring that the model's performance is more robust and reliable.

In particular, the data was split twice in this project. In the first split, we aimed to preserve one version of the test dataset for final evaluation. This was done before passing the data into any machine learning models, including feature selections, balancing methods, and classification algorithms. Due to the heavy imbalance of the given dataset, the first split was done with 20% in the testing set and 80% in the training sets to preserve as much information as possible for later data training. In the code, these sets of data were named *main.train.df* and *main.test.df*.

The second split was performed right before triggering all the data training in the pipeline with six classification models. The initial training data was then split again in a ratio of 66% to 33%. This split provided immediate training and testing data to validate and tune parameters in each classification algorithm. In the code, these sets of data were named *split.train* and *split.test*.

* 1. Building the classification models:
     1. Naive Bayes

The first model used in the project is Naïve Bayes. It is wrapped in a function with three parameters including the immediate training dataset, the immediate testing dataset, and the testing dataset from the initial splitting. The function first builds a Naive Bayes model running on the training dataset, then, it predicts the class labels for the instances in the testing dataset using the trained model. It calculates performance metrics, such as accuracy, precision, recall, and F1-score, based on the predictions compared to the actual class labels in the testing dataset. Additionally, the function tests the model on a separate testing dataset, initaldata\_test, and computes performance metrics for this dataset as well. Finally, the function returns a list containing the performance metrics for both the testing dataset used for model training (inner\_split\_test) and the separate testing dataset (inital\_split\_test). This approach allows for the evaluation of the model's performance on both the immediate testing data and a separate validation dataset, providing insights into its generalization ability.

* + 1. Logistic Regression

The second model utilized in this project is the logistic regression algorithm. The process commences with training an initial logistic regression model (*logitModel1*) using the provided training dataset. All logistic models are fitted using the *glm* function with the binomial family. A summary of this model's coefficients and statistical information is generated, providing insights into the relationships between predictors and the target variable.

Once trained, the model is applied to the test dataset to predict class probabilities (*logitmdl.pred.prob*). We implemented a function to determine the optimal threshold called the findOptimalThreshold function. The calculation prioritizes a combined metric balancing sensitivity and specificity. Since we are detecting people with depressive disorder and potentially developing preventive actions against the development of this condition, the thresholds are calculated weighing more on sensitivity. There is a balanced trade-off between total accuracy and recall, but we decided to put more focus on minimizing Type 2 errors.

Subsequently, variable importance is assessed using the *varImp* function, ranking predictors based on their contribution to the model's performance. The top-ranked variables are selected for further analysis and used to train a secondary logistic regression model (new\_logitModel). This refined model focuses on a subset of the most influential predictors, potentially improving interpretability and computational efficiency.

Finally, the performance of both models is evaluated on the original test dataset (*original\_test*). Predictions from the secondary model (*logitmdl\_main\_pred*) are generated using the selected top variables, and performance metrics are computed and compared to those of the initial model. This comprehensive approach not only enhances model interpretability through variable selection but also iteratively refines predictive accuracy, ensuring robust performance in real-world classification tasks.

* + 1. Decision Tree

The classification model in this project utilized the decision tree algorithm and it was built with two distinct functions in R: *J48* and *rpart*. The process begins by configuring the training control parameters, specifying a repeated 10-fold cross-validation approach to robustly assess model performance. For the J48 model, a grid of hyperparameters (C and M) is defined to tune the model during training. This grid facilitates the exploration of different parameter combinations to identify the optimal configuration that maximizes predictive accuracy.

Subsequently, the *J48* model is trained using the specified method and hyperparameter grid, leveraging the train function from the caret package. Once trained, the model is applied to the provided test dataset to generate predictions. These predictions are evaluated using predefined performance metrics (the function named *compute\_metrics\_and\_build\_table*).

Similarly, the *rpart* model is trained using the *rpart* algorithm, following the same cross-validation methodology and parameter tuning approach as the J48 model. The train function is again employed to train the *rpart* model, with the number of tuning iterations specified through the *tuneLength* parameter. Predictions are generated for the test dataset using the trained *rpart* model, and performance metrics are computed to evaluate its predictive capabilities.

Finally, this classification model outputs two sets of results for each decision tree method it employs.

* + 1. Random Forest

The fourth model used in this project is the random forest model. This was built using the caret package and the “rf” training method. The first step in this process was establishing control parameters the method chosen was cross-validation or “CV” and the number 10 indicates it will be 10 steps. The next step in this was creating the “mtryValues” sequence which specifies the number of features considered at each split. In this case “ncol(train)-1” was used to specify the number of features should be the number of columns minus 1.

Now that the initial parameters were set the random forest model could be run. The features chosen were all of columns except Class because class was selected as the target variable. The “mtryValues was used to create a tune grid and the importance of each feature was specified to be calculated too. This resulted in a model called *rfFit*.

With this rfFit model created the next step was to test how well this model can predict class values. The first way this was tested was on the secondary test dataset then this is tested on the original test dataset and the performance metrics are returned. Testing on both helps to ensure the model accuracy.

* + 1. K-Nearest Neighbors (KNN)

The KNN model is the 5th model chosen. The first step in this process is using the train control function to perform a repeated 10-fold cross-validation approach in order to robustly assess model performance. This cross validation is saved as the train control and then used in the following KNN model creation. The KNN model is created my using the “knn” method in the caret package.

In my first KNN model a tune length of 50 is chosen however, this model is later improved upon using a tune grid. The tune grid is created for a sequence of 25 numbers starting at 1 and increasing by 2 each time. This tune grid is then used in a tuned KNN model that is created using the same train control validation. The tune grid’s purpose is to try different K values within the model so the K length is not static and then the best model is selected. “Class” is chosen as the target variable and all other attributes in the training data frame are used to help predict in the model. The model is then used to predict the Class on the test data frames and the model as well as performance statistics on the test data frames are retuned within the “train\_and\_select\_knnModel” function.

* + 1. Neural Network (NN)

The final model chosen was a Neural Network. The control parameter selected was a 10 fold cross-validation which will divide the data into 10 parts to be trained and evaluated 10 times. After this a “nnetGrid” is established, this specifies a size of 1:5 which varies the number of hidden units in the neural layer network. The weight decay is also specified here which will perform L2 regularization meaning larger weights will be penalized to prevent overfitting. With the tuning parameter established the model can be trained.

The ”nnet” method within the caret function is chosen for this model. All columns that are not the Class in the training data frame are chosen as features for the model creation and the Class is chosen as the target variable. The tunning parameters that were specified earlier are applied here as well. This model returns a nnetFit$bestTune that was used when selecting the initial control parameters as well as the best model. This model is then run on the test data frames and the performance of the model is captured. The “train\_nnetModel” will return the model created as well as the performance stats on the test data.

* 1. Executing Classification Models with Feature Selection and Data Balancing

To combine all attribute selection techniques, imbalanced data treatments, and classification algorithms, a comprehensive process was implemented to iterate through each pre-defined function and execute all the possible combinations.

Initially, an empty dataframe *all\_results* is created to store the evaluation results. The code then iterates over each balancing function in the balance\_functions list. For each balancing function, the training data is balanced using the respective method. There are 4 balancing methods but we decided to reduce to the more complete ones to decrease total runtime.

A computer code on a black background

Description automatically generated

Subsequently, feature selection techniques, namely Boruta, information gain, and principal component analysis (PCA), are trained on the balanced training dataset from the second split then the irrelevant columns will be filtered on the rest of the split training and testing datasets. This step aims to identify the most relevant features for classification while maintaining the unbias of the data.

A screen shot of a computer code

Description automatically generated

The code then defines a set of classification models, including Naive Bayes, Logistic Regression, Decision Tree, Random Forest, KNN, and Neural Network. For each classification model, the training process is executed on different datasets obtained from the various feature selection methods.

A computer screen shot of text

Description automatically generated

Within each iteration, the evaluation results are stored in the results list. For each feature selection method, if the results exist, they are flattened and appended to the all\_results list along with information about the balancing method, classification model, and feature selection technique.

Finally, all the collected results are combined into a single dataframe named results\_df, consolidating the performance metrics of each model across different feature selection methods and data balancing techniques.

This systematic approach allows for a thorough comparison of classification model performance under various conditions, facilitating informed decision-making in model selection and feature engineering strategies.

1. **Model Evaluation & Interpretation**
   1. Evaluation metrics computation and interpretation

In order to extract the performance of each classification model, a function named *compute\_metrics\_and\_build\_table* is designed to compute various evaluation metrics for binary classification tasks and build a table summarizing these metrics.

The function takes two arguments: predicted\_classes (predicted class labels) and actual\_labels (true class labels). With these inputs, it computes both the confusion matrix (cm) considering the “Y” value positive and an inverted confusion matrix (cm\_inverted) considering the “N” value positive using the *confusionMatrix* function from the *caret* package.

A screen shot of a computer program

Description automatically generated

For each class (Class 1 and Class 2), the function calculates several evaluation metrics, including Accuracy, True Positive Rate (TPR), False Positive Rate (FPR), Precision, Recall, F-Measure, Matthews Correlation Coefficient (MCC), and Kappa. It iterates through the confusion matrices to compute these metrics accurately.

Additionally, the function calculates the ROC curve and its area under the curve (ROC Area) using the roc and auc functions from the *pROC* package. The ROC Area is then also added to the result table for each class, providing insight into the model's discriminatory power.

To structure the results, the function constructs a data frame named class\_metrics to store the computed metrics for each class. It also calculates the weighted average of metrics across classes and appends this to the class\_metrics data frame. Finally, it combines class\_metrics with the weighted average row to create the result\_table.

A screenshot of a computer screen

Description automatically generated

All the transformations and calculations are wrapped within a user-defined function for easy access in each classification model. This function is later used in all models in the final pipeline to evaluate the performance of each combination and to assist in creating the result dataframe.

* 1. Comparison of model performance using various balancing methods
  2. Discussion on the effectiveness of each classification model
  3. Interpretation of the results and implications for predicting depressive disorders

1. **Conclusion**
   1. Summary of key findings and insights from the project
   2. Reflection on the effectiveness of different classification models
   3. Suggestions for future research or improvements in methodology
   4. Closing remarks
2. **VII. References**

* List of all sources referenced in the paper, including datasets, packages, and methodologies used