AAI-510-01: Assignment 2.1 - Ian Feekes
https://github.com/USD-AAI/aai-510-01-su22-new-ianfeekes-sandiego

Classification Tree:

Top 10 Variables:

Under the Global Variables snippet of python code, a list denoting the top 10 variables selected in the Module 1 assignment is declared:

```
# Chosen top 10 variables from ASG1
chosenTopTenVariables = ['TARGET',
                              'NAME_CONTRACT_TYPE',
                        'FLAG_OWN_CAR',
                        'CODE_GENDER',
                        'EXT_SOURCE_1',
                        'DAYS_BIRTH',
                        'CNT_CHILDREN',
                        'AMT_CREDIT',
                        'NAME_INCOME_TYPE',
                        'NAME_EDUCATION_TYPE',
                        'ORGANIZATION_TYPE']
```

Classification Tree Summary:

The sklearn library DecisionTreeClassifier structure was primarily used for plotting and predicting the data (which was split at a 70 training 30 testing from training.csv, giving about 30,000 training entries after data cleaning).

The decision tree was trained using various parameters - Gini was initially used to save on training time, but most models discussed in this submission for the tree were trained with entropy for slight accuracy boosts. Various levels of max depth were used for initial pruning on the decision tree, and the model was ran once on an un-pruned decision tree (which had very long training time and awful visualization.). The code for creating and plotting the decision tree is as in the main method as follows:

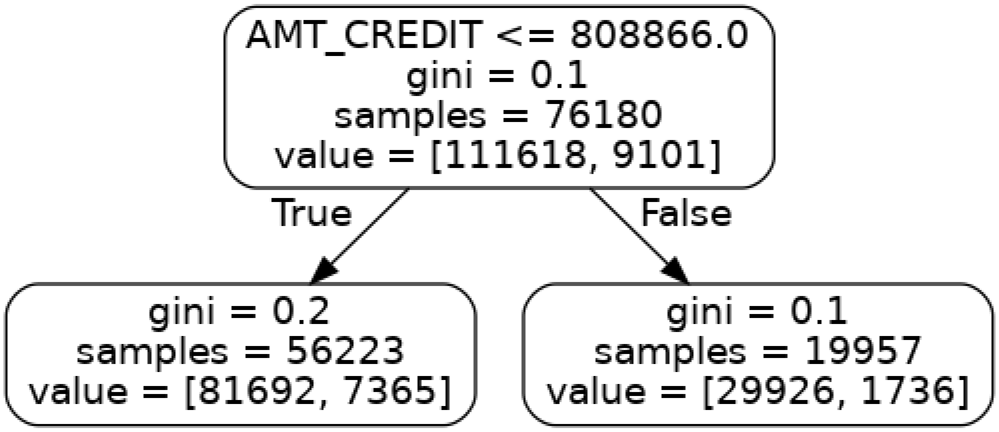
```
# Split dataset into training set and test set
X_train, X_test, y_train, y_test = train_test_split(features, labels, test_size=0.7, random_state=1)

# Create Decision Tree classifer object
clf = DecisionTreeClassifier(criterion="entropy")
clf = clf.fit(X_train, y_train)
# Try to load the model from disk, else retrain (very time intensive)
try:
    clf = pickle.load(open('decision_tree_classifier.sav', 'rb'))
    print("CLF loaded")
except Exception:
    # Train Decision Tree Classifier
    clf = clf.fit(X_train, y_train)
    print("CLF fitted")
```

```python
# save the model to disk
pickle.dump(clf, open('decision_tree_classifier_no_prune.sav', 'wb'))

# Predict the response for test dataset
y_pred = clf.predict(X_test)
# Model Accuracy, how often is the classifier correct?
print("Accuracy of single decision tree:",metrics.accuracy_score(y_test, y_pred))

# Visualize tree
dot_data = StringIO()
export_graphviz(clf, out_file=dot_data,
            filled=True, rounded=True,
            special_characters=True,feature_names = list(features.columns), class_names=['0','1'])
graph = pydotplus.graph_from_dot_data(dot_data.getvalue())
graph.write_png('CLF.png')
Image(graph.create_png())
print("Finished writing tree")

# Plot Feature Importance
importances = clf.feature_importances_
indices = np.argsort(importances)
plt.title('Feature Importances')
plt.barh(range(len(indices)), importances[indices], color='b', align='center')
plt.yticks(range(len(indices)), [features.columns[i] for i in indices])
plt.xlabel('Relative Importance')
plt.savefig("featureImportances_single_tree.png")
print("Finished plotting tree")
```

Classification Tree Visualizations and Plotting:

As mentioned, many classification trees were created with various depths and training methods, and they will be summarized in increasing levels of accuracy. Each tree will be shown with its unique parameters, its visualization, its accuracy output from python, and its variable importance plots/python output:

1. Classification Tree trained with gini and depth of one

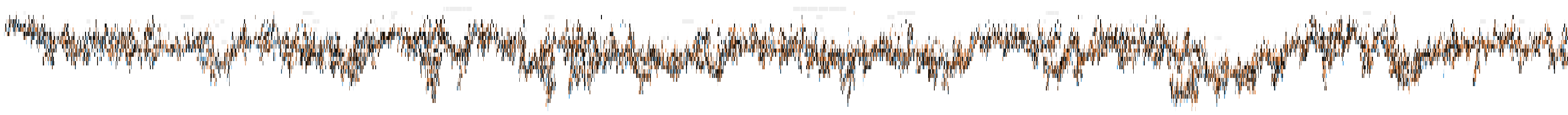


Accuracy: 0.82355218890192

As you can see from the visualization, this most basic tree found AMT_CREDIT to be the most important variable and has no analysis for the remaining.

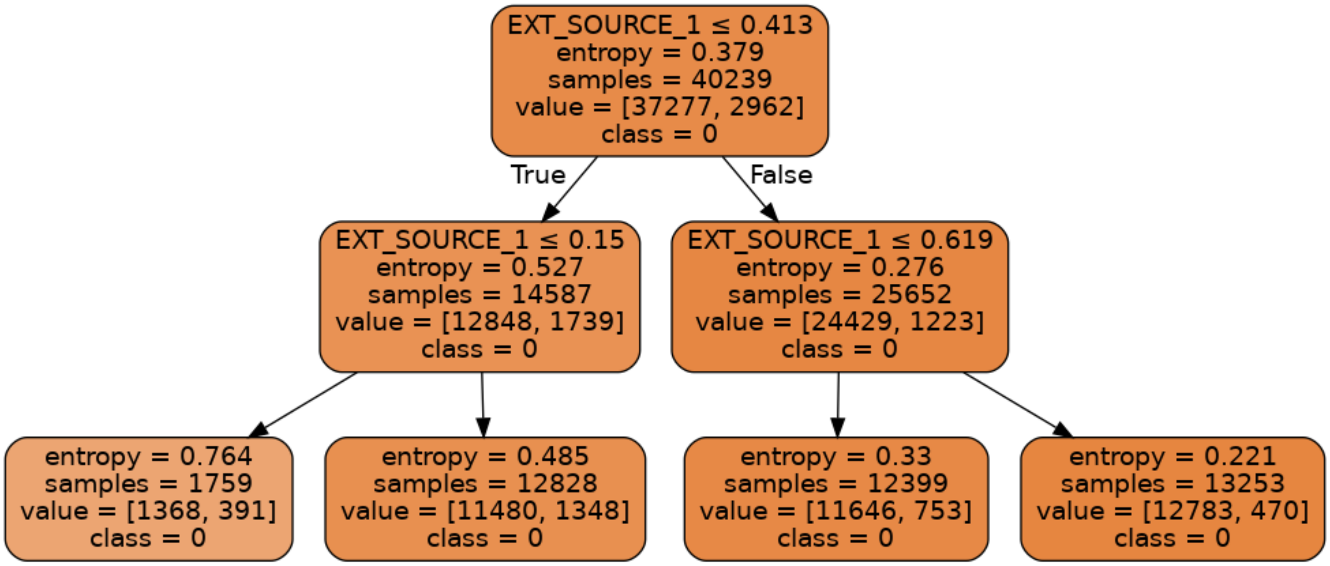
2. Unpruned Classificaition Tree, trained on entropy

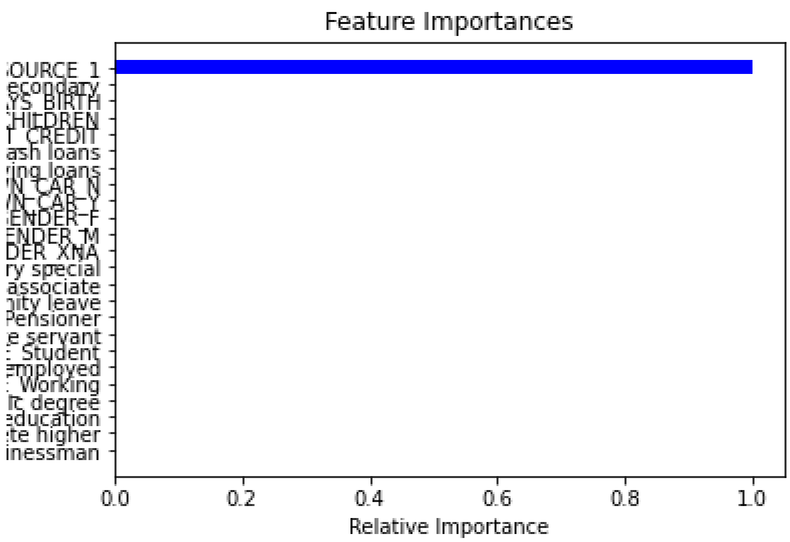
Visualization for this tree is absolutely colossal in a .5GB file that cannot be properly shown in this file. Below are two trimmed versions of the file simply to illustrate the size of such a tree while trained with one-hot-encoding of ORGANIZATION_TYPE which has many variables



Accuracy: 0.857685999701804

3. Classification tree pruned at depth of 1, trained on entropy, ORGANIZATION_TYPE removed

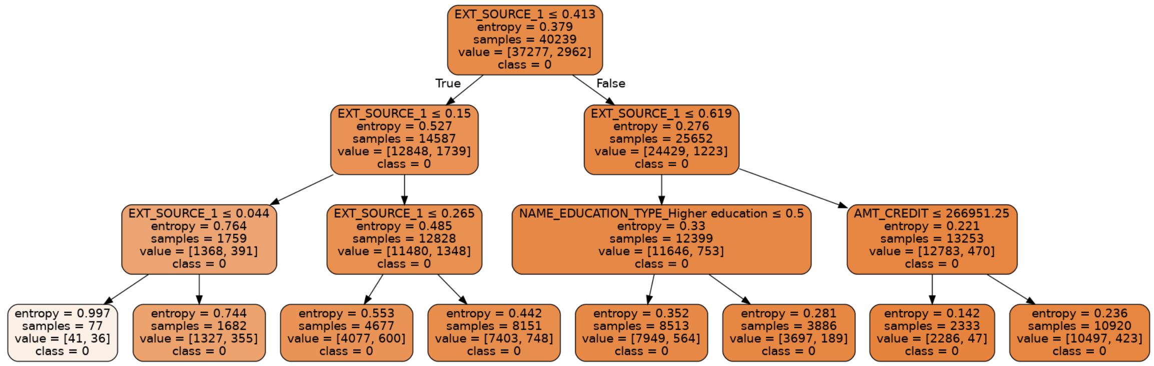


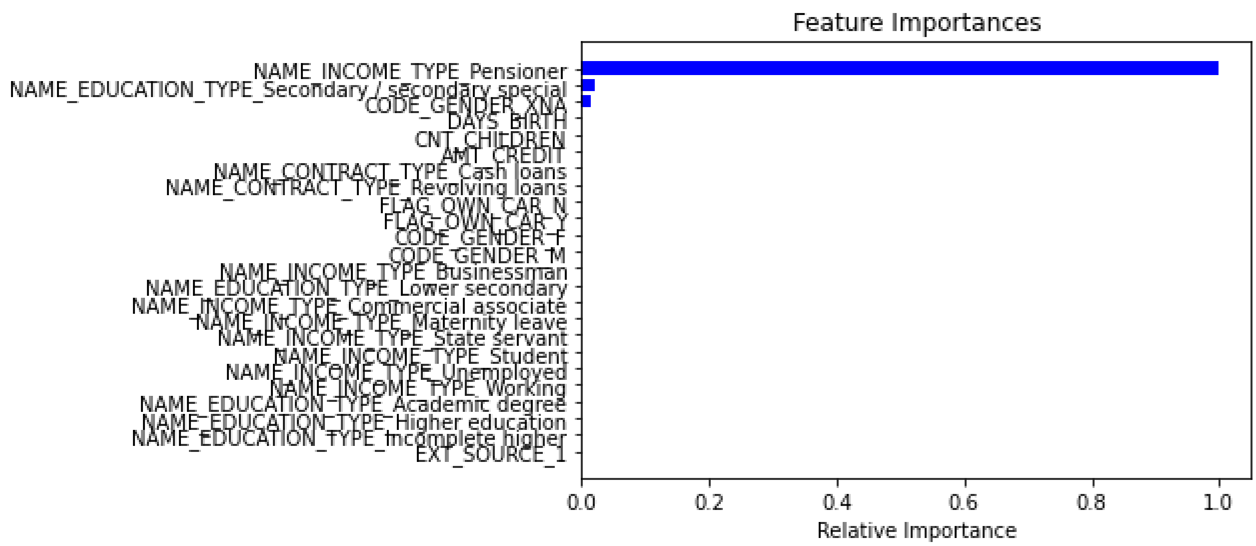


Accuracy: 0.9299985090204265

While moving down slightly in depth provided slightly nicer visualization, the overfitting of a classification tree, especially with such shallow pruning, is especially apparent. This accuracy is not reflective of the decision tree necessarily being so accurate in predictions, but rather in the disproportionate training set.

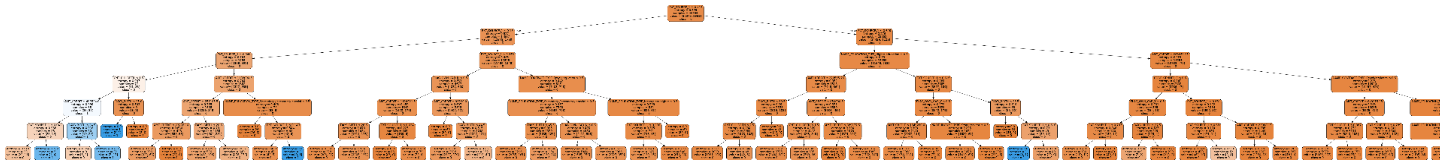
4. Classification tree pruned at depth of 2, trained on entropy, ORGANIZATION_TYPE removed

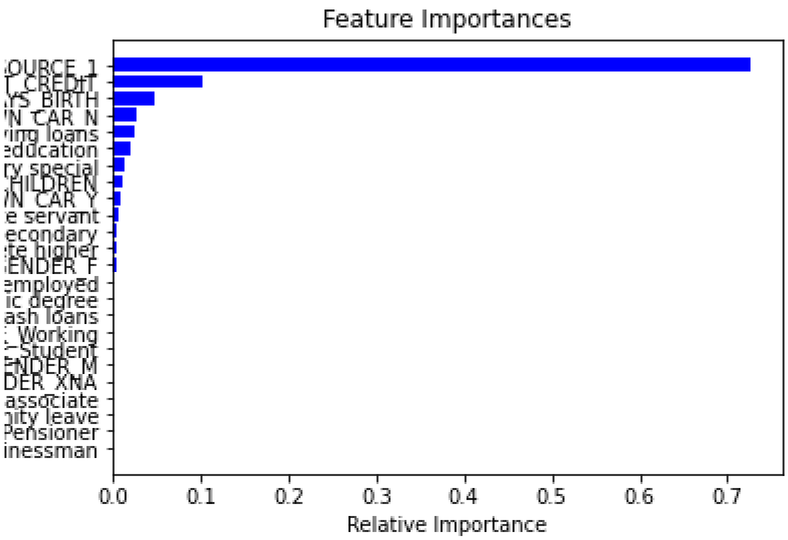




Accuracy of single decision tree: 0.9241165569684964

5. Classification tree pruned at depth of 6, trained on entropy, ORGANIZATION_TYPE removed





Accuracy of single decision tree: 0.9071165569684964

Random Forest

Random Forest Summary:

Similarly, sklearn's RandomForestClassifier was used with varying numbers of random states. The meat of the code for creating, training, and plotting the random forest is as below:

```
 # Create random tree forest
model = RandomForestClassifier(n_estimators=10000, random_state=1)

try:
    model = pickle.load(open('decision_tree_forest_2.sav', 'rb'))
except Exception:
    model.fit(X_train, y_train)
y_pred = model.predict(X_test)
print("Random forest accuracy: ", metrics.accuracy_score(y_test, y_pred))

tree_small = model.estimators_[0]
export_graphviz(tree_small,
                out_file = 'small_tree.dot',
                feature_names = list(features.columns),
               rounded = True, precision = 1)
(graph, ) = pydot.graph_from_dot_file('small_tree.dot')
graph.write_png('small_tree.png')
pickle.dump(model, open('decision_tree_forest_bad.sav', 'wb'))

for name, importance in zip(list(features.columns), model.feature_importances_):
    print(name, " = ", importance)

importances = model.feature_importances_
indices = np.argsort(importances)
plt.title('Feature Importances')
plt.barh(range(len(indices)), importances[indices], color='b', align='center')
plt.yticks(range(len(indices)), [features.columns[i] for i in indices])
plt.xlabel('Relative Importance')
plt.savefig("featureImportances.png")
```

Random Forest Feature Importance:

1. **Random forest of 50, trained on top 10 variables chosen:**

   Random forest accuracy: 0.9299985090204265

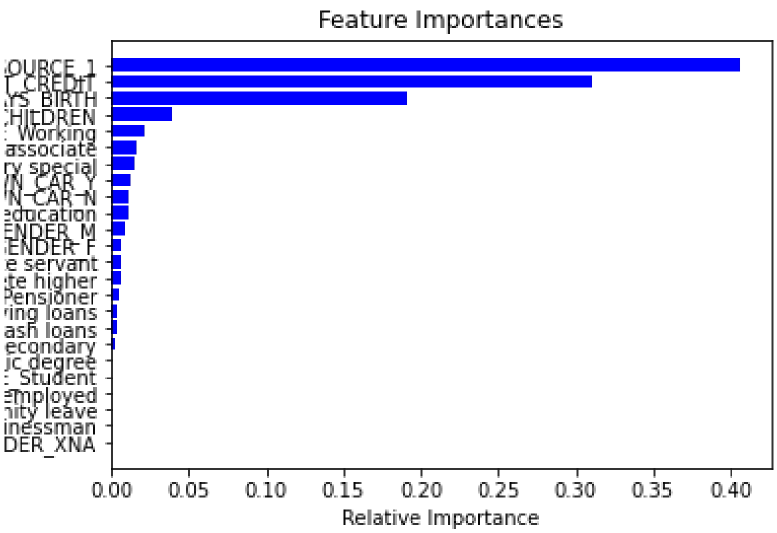
   Feature Importance:

   1. EXT_SOURCE_1 = 0.5431439288670782
   2. DAYS_BIRTH = 0.13451212861948503
   3. CNT_CHILDREN = 0.004615209489777085
   4. AMT_CREDIT = 0.044917726733461526
   5. NAME_CONTRACT_TYPE_Revolving loans = 0.022298432442162956
   6. FLAG_OWN_CAR_Y = 0.013249576885113152

7. CODE_GENDER_M = 0.03271312382403999
8. CODE_GENDER_XNA = 1.877738723675614e-06
9. NAME_INCOME_TYPE_Commercial associate = 0.0033248511608900723
10. NAME_INCOME_TYPE_Maternity leave = 0.0008910519087986602
11. NAME_INCOME_TYPE_Pensioner = 0.010274348399659
12. NAME_INCOME_TYPE_State servant = 0.002281877249666908
13. NAME_INCOME_TYPE_Student = 4.6105073227634346e-07
14. NAME_INCOME_TYPE_Unemployed = 0.0007887600972858415
15. NAME_INCOME_TYPE_Working = 0.027139483747454486
16. NAME_EDUCATION_TYPE_Higher education = 0.09818341896459336
17. NAME_EDUCATION_TYPE_Incomplete higher = 0.0015985878204307616
18. NAME_EDUCATION_TYPE_Lower secondary = 0.00039578199601524413
19. NAME_EDUCATION_TYPE_Secondary / secondary special = 0.05966937300463179

2. Random Tree Forest unpruned of 100, trained on top 10 variables chosen:
Random forest accuracy: 0.9299985090204265
Feature Importance:

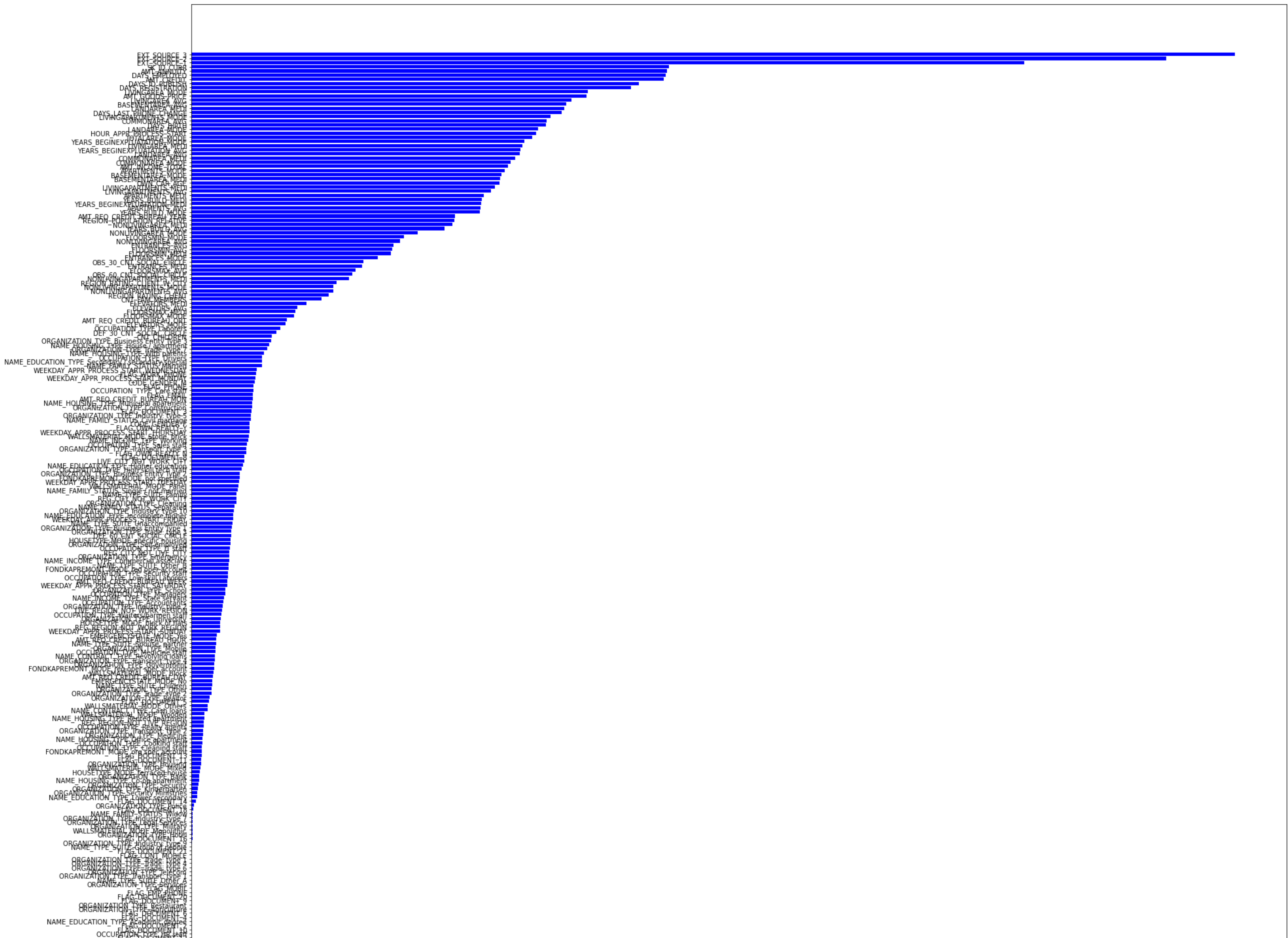
1. EXT_SOURCE_1 = 0.4041234272945421
2. DAYS_BIRTH = 0.19085473053437377
3. CNT_CHILDREN = 0.039811979513917274
4. AMT_CREDIT = 0.31071369472713856
5. NAME_CONTRACT_TYPE_Cash loans = 0.00240177345352055
6. NAME_CONTRACT_TYPE_Revolving loans = 0.002483105758847902
7. FLAG_OWN_CAR_N = 0.004701809419534974
8. FLAG_OWN_CAR_Y = 0.004809837248107228
9. CODE_GENDER_F = 0.00398380351722254
10. CODE_GENDER_M = 0.004248730482684896
11. CODE_GENDER_XNA = 0.0
12. NAME_INCOME_TYPE_Businessman = 1.3867370905692722e-07
13. NAME_INCOME_TYPE_Commercial associate = 0.005681821382424121
14. NAME_INCOME_TYPE_Maternity leave = 1.5715417804698155e-07
15. NAME_INCOME_TYPE_Pensioner = 0.0024861635453846515
16. NAME_INCOME_TYPE_State servant = 0.003468226750246607
17. NAME_INCOME_TYPE_Student = 3.062217858817667e-05
18. NAME_INCOME_TYPE_Unemployed = 1.3200058272102097e-05
19. NAME_INCOME_TYPE_Working = 0.005810474794718584
20. NAME_EDUCATION_TYPE_Academic degree = 0.00019499600713487045
21. NAME_EDUCATION_TYPE_Higher education = 0.004487909990881818
22. NAME_EDUCATION_TYPE_Incomplete higher = 0.0029942190480386783
23. NAME_EDUCATION_TYPE_Lower secondary = 0.0018701992214735564
24. NAME_EDUCATION_TYPE_Secondary / secondary special = 0.004828979245060068



3. Random Tree forest Trained on all Variables:

The chart may be difficult to make out - the top features shown are:

1. EXT_SOURCE_3
2. EXT_SOURCE_2
3. EXT_SOURCE_1
4. AMT_ANNUITY
5. DAYS_EMPLOYED
6. AMT_CREDIT
7. DAYS_ID_PUBLISH
8. DAYS_REGISTRATION
9. LIVINGAREA_MODE
10. AMT_GOODS_PRICE



Random Forest Analysis:

The top 10 variables in feature importance from the random forest above, ran on all variables, has only a few variables that I previously chose.

EXT_SOURCE_1, my chosen variable, was top in all of the random tree analysis, and EXT_SOURCE_3, 2, and 1 respectively are top in the random forest feature importance. As mentioned in assignment 1 analysis, it’s quite likely that it would be most appropriate in feature engineering to create an aggregate of the external credit sources.

AMT_ANNUITY was not on my list, however AMT_CREDIT was on my list, and it would likely be most appropriate to generate a relative financial loan risk from common economical equations taking a values from ratios on the annuity, credit, and cost of living (which would be AMT_GOODS_PRICE).

DAYS_EMPLOYED trumps my DAYS_BORN for longevity and ability to handle responsibility, and DAYS_ID_PUBLISH would likely be aggregated into some variable that takes the three of these.

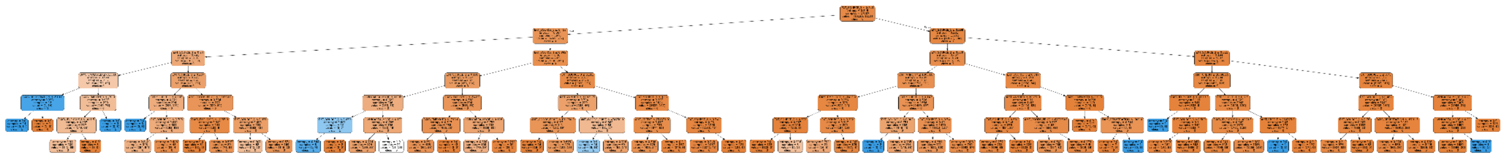
The one “category” of variable that was completely absent from my list which showed up in the random forest analysis was LIVINGAREA_MODE. Certainly looking back I should have chosen at least one variable that had to do with living space, or something directly correlated to asset values, especially given that they had some strong correlations in the heatmap.

Replotting Single Tree vs Random Forest with new Top 10:

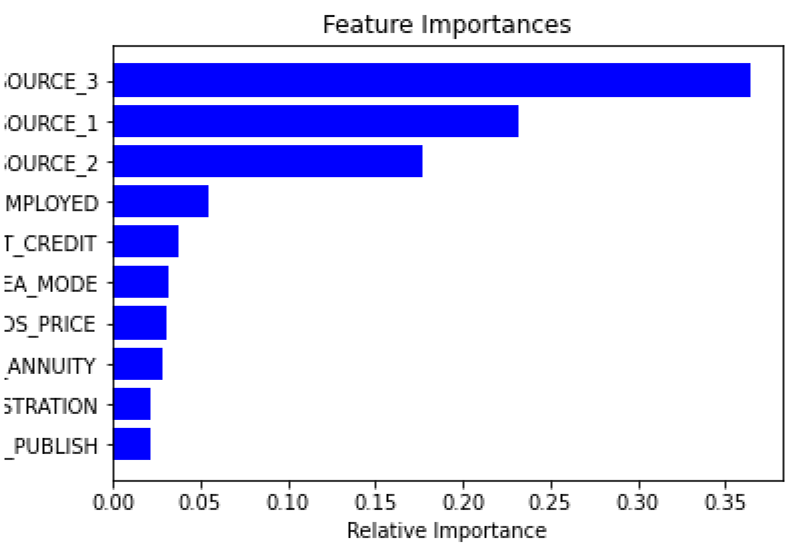
Results:

Very simple code modifications allowed the new variables to be taken in and trained. The following is plots and output for two single-decision trees (one pruned at a maximum depth of 6, one un-pruned) and a forests at length 1000:

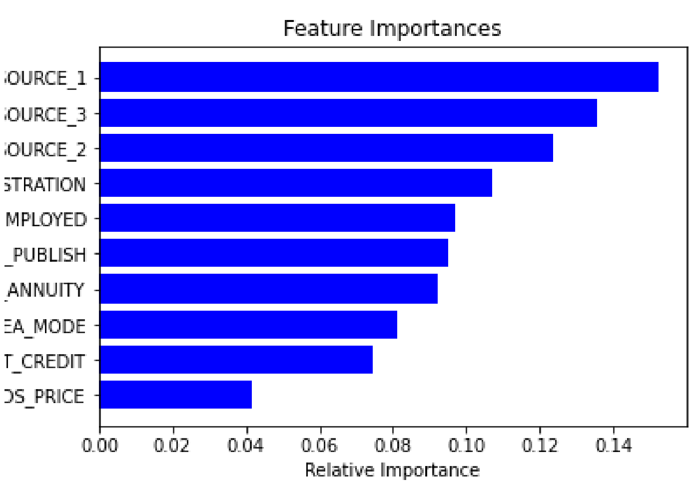
1. Single decision tree maximum depth of 6, trained under entropy



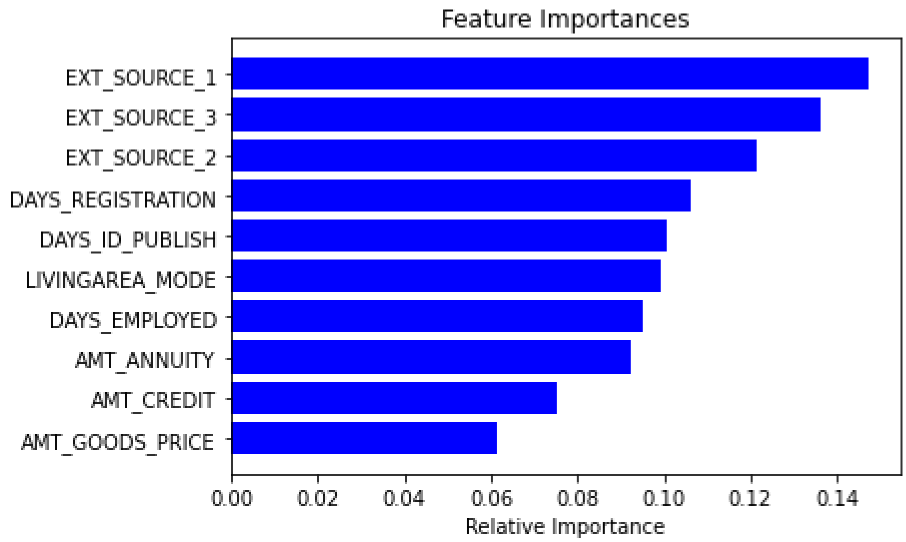
   Accuracy of single decision tree: 0.92543536304123103



2. Single decision tree, unpruned, trained under entropy
   Accuracy of single decision tree: 0.8834410740924913



3. Random forest of length 1000
   Random forest accuracy: 0.9351815017404277



Comparison/Analysis:
We can certainly see that a single decision tree, if its depth is pruned, will not properly classify the correct features. It is additionally prone to overfitting, as seen by the importance values on EXT_SOURCE_3 for the first of the bar graphs above. The high accuracy may be due to the disproportionate training data on the TARGET label.

A single decision tree, even when unpruned, may still suffer from overfitting if we compare the random forest to the decision tree feature importance charts. Certainly there is a marginal improvement in accuracy, and given the convoluted nature of the dataset with regards to accuracy and overfitting, this improvement may in truth be very significant.

These differences are attributed to the very nature of single decision trees vs. forests. The forests allow for the data model to be less biased and have lower variance, leading to better accuracy - especially on larger datasets such as this.

Software Summary:
The code configures some output directories for organization of output files and then reads in the data frame from the training set, since all 10 variables were chosen from the training data set (which may have been short-sighted).

It then proceeds to clean up the data frame to apply only to the top 10 variable columns chosen, drops bad values*, separates a label and feature data frame, and represents DAYS_BIRTH in a way that is more intuitive from an analytics perspective.

Before the data is ready to be fitted into a model, the categorical variables (of which there are many) are separated from the continuous variables and converted into additional variables which are represented as integers. A disadvantage of this approach is that it creates a feature variable list that ends up being very large, and requires quite some time for the data frame to train (especially the organization type). Once this has been performed, the data frames are rejoined.

Then the classification tree is created/loaded and fit to these variables and saved. The tree's accuracy is recorded and the graphics are created and saved for analysis.

Software Summary - Notes:
The main method is largely a continuation of assignment1 main method behavior, however the annotated code submitted here will have most of the plotting behavior stripped clean for cosmetics. Additionally, the imports, software configuration, global variables, and helper functions are added into the appendix for readability.

Main Method:
(note: for most up-to-date code please view the repository
https://github.com/USD-AAI/aai-510-01-su22-new-ianfeekes-sandiego)

```python
# Main Method:
#-------------
# Reads in the data files, plots certain values and creates useful analytical plots and does
# some light data cleaning
#
#
# Parameters:
# -----------
# @param debug:             flag for displaying debugger output
# @param dropMissingValues: true to drop rows with empty values, false to set null values to mean
# @param: savePlots:        true to plot various initial data points
# @param outlierThreshold:  z-value with which to threshold outliers
#
# Returns:
# ---------
# None
#
def main(debug = True, dropMissingValues = False, savePlots = False, outlierThreshold = 3):

   # Create output directories for files and plots to be saved to
   createOutputDirectories(debug)

   # Read the data, assigning independant and dependant variables: x and y respectively
   data, x, y = readData(datasetName, targetColumnName, debug)
   # Assign data to only our top ten variables to save runtime and data cleaning
   data = data[chosenTopTenVariables]
   assert(len(data.columns) == 11)

   # Drop bad values from the variables we care about
   data = data.dropna(axis=0)
   for col in data.columns:
       assert(data[col].isnull().sum() == 0)
   if debug == True:
       print("Null data values properly dropped")

   # Allocate feature and label data frames
   labels = pd.DataFrame()
   labels[targetColumnName] = data[targetColumnName]
   features = data.drop(targetColumnName, axis = 1)

   # Normalize days birth to something reasonable
```

```
yearsBorn = round(abs(features['DAYS_BIRTH'] / (365)))
features['DAYS_BIRTH'] = round(abs(features['DAYS_BIRTH'] / (365)))
plot_column(features.join(labels[targetColumnName]), 'DAYS_BIRTH', ['box'])

# Get sub-data frames that contain variables from each respective data type - we don't want target
dataStrings, dataContinuous = allocateTypes(features, debug)

# One hot encoding for categorical variables
dataStrings = pd.get_dummies(dataStrings)

# Re-merge data types now
features = dataContinuous.join(dataStrings)

# Split dataset into training set and test set
X_train, X_test, y_train, y_test = train_test_split(features, labels, test_size=0.1, random_state=1)

 # Create Decision Tree classifier object
clf = DecisionTreeClassifier()
print("Decision Tree Classifier created")

# Train Decision Tree Classifier
clf = clf.fit(X_train,y_train)
print("clt fitted")

#Predict the response for test dataset
y_pred = clf.predict(X_test)
print("predicted")

# Model Accuracy, how often is the classifier correct?
print("Accuracy:",metrics.accuracy_score(y_test, y_pred))

dot_data = StringIO()
export_graphviz(clf, out_file=dot_data,
            filled=True, rounded=True,
            special_characters=True,feature_names = feature_cols,class_names=['0','1'])
graph = pydotplus.graph_from_dot_data(dot_data.getvalue())
graph.write_png('diabetes.png')
Image(graph.create_png())

# Create random tree forest
model = RandomForestClassifier(n_estimators=50, max_depth=3, random_state=1)
model.fit(X_train, y_train)
predictions = model.predict(X_test)
print("Random tree forest predictions: ", predictions)
```

```
    tree_small = model.estimators_[5]
    export_graphviz(tree_small, out_file = 'small_tree.dot', feature_names = list(features.columns),
rounded = True, precision = 1)
    (graph, ) = pydot.graph_from_dot_file('small_tree.dot')
    graph.write_png('small_tree.png')


    if debug == True:
        debugFd.write("Main Method Cell Completed...\n")
        debugFd.write(lineString+"\n")
```

Appendix:

Helper Functions:

```
# Helper Functions

# If the debugging flag is on, creates directories to store output data
#
# Parameters:
# -----------
# @param debug: flag for displaying debugger output
#
# Returns:
# ---------
# None
#
def createOutputDirectories(debug = False):
    if debug == False:
        return
    if not os.path.exists("images"):
        os.mkdir("images")
    if not os.path.exists("images/initialPlots"):
        os.mkdir("images/initialPlots")
    if not os.path.exists("images/topTenPlots"):
        os.mkdir("images/topTenPlots")
    if not os.path.exists("outputFiles"):
        os.mkdir("outputFiles")
    print("createOutputDirectories...success")


# Reads csv file into data frame and sets independant and dependant variables
#
# Parameters:
# -----------
# @param fileName: string for full relative file path of csv file
# @param dependantVarColumnName: csv file column matching name of column for dependant variable
# @param debug: flag for displaying debugger output
#
# Returns:
# ---------
# data: dataframe object of csv file reading
# independantVars: independant variables (all data that isn't targetColumnName)
```

```python
# dependantVar: dependant variable
#
def readData(fileName, dependantVarColumnName = targetColumnName, debug = False):
    independantVars = []
    dependantVar = []
    data = pd.read_csv(fileName)
    index = None
    for i ,col in enumerate(data.columns):
        if col == dependantVarColumnName:
            index = i
    if index != None:
        dependantVar = data.iloc[:, index]
        independantVars = data.iloc[:]
        independantVars.pop(dependantVarColumnName)
    if debug:
        fd = open(initialDataFileName, "w+")
        fd.write("This file contains the initial data frame without cleaning:\n")
        fd.write(str(data))
        fd.close()
        print("readData...completed")
    return data, independantVars, dependantVar


# Drops rows from dataset which are missing. Prints missing value data for debugging
#
# Parameters:
# -----------
# @param data: dataframe to have missing values dropped and returned
# @param debug: flag for displaying debugger output
#
# Returns:
# ---------
# data: dataframe object with missing values dropped
#
def dropMissingValues(data, debug = False):
    # Drop missing values
    ret = data.dropna(axis=0)
    # Show number of missing values per independant variable
    if debug:
        fd = open(missingValFileName, "w+")
```

```
        fd.write("This data shows the independant variables which contained missing values and the count of each:\n")
        fd.write(str(data.isnull().sum()))
        fd.close()
        fd = open(noMissingValuesFileName, "w+")
        fd.write("This data shows the independant variables which are used for analysis with no mising values:\n")
        fd.write(str(ret.isnull().sum()))
        fd.close()
        print("dropMissingValues...completed")
    return ret


def setMeanValues(data, debug = False):
    return data

# Writes distribution of data frame to text file
#
# Parameters:
# -----------
# @param data: dataframe to have distribution written to text file
# @param debug: flag for displaying debugger output
#
# Returns:
# ---------
# None
#
def writeDistribution(data, debug = False):
    if debug == False:
        return
    numpy_array = data.to_numpy()
    fd = open(initialDistributionFileName, "w+")
    fd.write(str(numpy_array))
    fd.close()
    print("writeDistribution...success")


def doBar(data, column_name, figsize = (18,6),
          percentage_display = True,
          plot_defaulter = True, rotation = 0,
```

```
        horizontal_adjust = 0,
        fontsize_percent = 'xx-small',
        dirName = 'images/initialPlots/'):


   print(f"Total Number of unique categories of {column_name} =
{len(data[column_name].unique())}")

   plt.figure(figsize = figsize, tight_layout = False)
   sns.set(style = 'whitegrid', font_scale = 1.2)

   #plotting overall distribution of category
   plt.subplot(1,2,1)
   data_to_plot = data[column_name].value_counts().sort_values(ascending = False)
   ax = sns.barplot(x = data_to_plot.index, y = data_to_plot, palette = 'Set1')

   if percentage_display:
       total_datapoints = len(data[column_name].dropna())
       for p in ax.patches:
           ax.text(p.get_x() + horizontal_adjust, p.get_height() + 0.005 * total_datapoints,
'{:1.02f}%'.format(p.get_height() * 100 / total_datapoints), fontsize = fontsize_percent)

   plt.xlabel(column_name, labelpad = 10)
   plt.title(f'Distribution of {column_name}', pad = 20)
   plt.xticks(rotation = rotation)
   plt.ylabel('Counts')

   #plotting distribution of category for Defaulters
   if plot_defaulter:
       percentage_defaulter_per_category = (data[column_name][data.TARGET == 1].value_counts() *
100 / data[column_name].value_counts()).dropna().sort_values(ascending = False)

       plt.subplot(1,2,2)
       sns.barplot(x = percentage_defaulter_per_category.index, y =
percentage_defaulter_per_category, palette = 'Set2')
       plt.ylabel('Percentage of Defaulter per category')
       plt.xlabel(column_name, labelpad = 10)
       plt.xticks(rotation = rotation)
       plt.title(f'Percentage of Defaulters for each category of {column_name}', pad = 20)
```

```
    fileName = dirName + column_name + '.png'
    plt.savefig(fileName)


# Plots a column name of the dataframe and saves each plot into a file
#
# Parameters:
# -----------
# @param data:       dataframe to have distribution written to text file
# @param plots:      types of plots for each column to show e.g. "box"
# @param: figsize:   size of figure for matplotlib to plot
# @param: log_scale: flag to log the scale of the plot
#
# Returns:
# ---------
# None
#
def plot_column(data,
               column_name,
               plots = [],
               figsize = (20,8),
               log_scale = False,
               dirName = 'images/initialPlots/'):

    if 'bar' in plots:
        doBar(data, column_name, figsize, dirName = dirName)
        return
    data_to_plot = data.copy()
    plt.figure(figsize = figsize)
    sns.set_style('whitegrid')

    for i, ele in enumerate(plots):
        plt.subplot(1, len(plots), i + 1)
        plt.subplots_adjust(wspace=0.25)
        if ele == 'CDF':
            #making the percentile DataFrame for both positive and negative Class Labels
            percentile_values_0 = data_to_plot[data_to_plot.TARGET ==
0][[column_name]].dropna().sort_values(by = column_name)
            percentile_values_0['Percentile'] = [ele / (len(percentile_values_0)-1) for ele in
range(len(percentile_values_0))]
```

```
            percentile_values_1 = data_to_plot[data_to_plot.TARGET ==
1][[column_name]].dropna().sort_values(by = column_name)
            percentile_values_1['Percentile'] = [ele / (len(percentile_values_1)-1) for ele in
range(len(percentile_values_1))]

            plt.plot(percentile_values_0[column_name], percentile_values_0['Percentile'], color =
'red', label = 'Non-Defaulters')
            plt.plot(percentile_values_1[column_name], percentile_values_1['Percentile'], color =
'black', label = 'Defaulters')
            plt.xlabel(column_name)
            plt.ylabel('Probability')
            plt.title('CDF of {}'.format(column_name))
            plt.legend(fontsize = 'medium')
            if log_scale:
                plt.xscale('log')
                plt.xlabel(column_name + ' - (log-scale)')
        elif ele == 'distplot':
            sns.distplot(data_to_plot[column_name][data['TARGET'] == 0].dropna(),
                         label='Non-Defaulters', hist = False, color='red')
            sns.distplot(data_to_plot[column_name][data['TARGET'] == 1].dropna(),
                         label='Defaulters', hist = False, color='black')
            plt.xlabel(column_name)
            plt.ylabel('Probability Density')
            plt.legend(fontsize='medium')
            plt.title("Dist-Plot of {}".format(column_name))
            if log_scale:
                plt.xscale('log')
                plt.xlabel(f'{column_name} (log scale)')
        elif ele == 'violin':
            sns.violinplot(x='TARGET', y=column_name, data=data_to_plot)
            plt.title("Violin-Plot of {}".format(column_name))
            if log_scale:
                plt.yscale('log')
                plt.ylabel(f'{column_name} (log Scale)')
        elif ele == 'box':
            sns.boxplot(x='TARGET', y=column_name, data=data_to_plot)
            plt.title("Box-Plot of {}".format(column_name))
            if log_scale:
                plt.yscale('log')
```

```python
            plt.ylabel(f'{column_name} (log Scale)')

    fileName = dirName + column_name + '.png'
    plt.savefig(fileName)


def showTargetPlot(data, debug = False):
    class_dist = data[targetColumnName].value_counts()

    if debug == True:
        print(class_dist)

    plt.figure(figsize=(12,3))
    plt.title('Distribution of TARGET variable')
    plt.barh(class_dist.index, class_dist.values)
    plt.yticks([0, 1])

    for i, value in enumerate(class_dist.values):
        plt.text(value-2000, i, str(value), fontsize=12, color='white',
                 horizontalalignment='right', verticalalignment='center')
    plt.show()


def showHeatmap(data):
    corrmat = data.corr()
    top_corr_features = corrmat.index
    plt.figure(figsize=(20,20))
    #plot heat map
    g=sns.heatmap(data[top_corr_features].corr(),cmap="RdYlGn")
    plt.show()


# Allocates data frames for each data type of argument data frame
# @TODO: implement data frame of integer types not being labeled as categorical
#
# Parameters:
# -----------
# @param data:   dataframe to be split into respective types
# @param debug: flag for displaying debugger output of writing columns into respective files
#
```

```
# Returns
# ---------
# strTypes           columns of string type
# continuousTypes    columns of continuous variables
# categorical        columns of categorical types
#
def allocateTypes(data, debug = False):
   strTypes = data.select_dtypes(include='object')
   continuousTypes = data.select_dtypes(include = ['float64', 'int64'])
   if debug == True:
       fd = open(stringVariablesFile, "w+")
       fd.write("String-type variables:\n")
       fd.write(lineString)
       for col in strTypes.columns:
           fd.write(col + "\n")
       fd.close()
       fd = open(continuousVariablesFile, "w+")
       fd.write("Continuous-type variables:\n")
       fd.write(lineString)
       for col in continuousTypes.columns:
           fd.write(col + "\n")
       fd.close()
       print("allocateTypes...success")
   return strTypes, continuousTypes


# Workaround to insert string into file without overwriting contents
#
# Parameters:
# -----------
# @param originalfile: original file name
# @param string:       string to be written to file
#
# Returns:
# ---------
# None
#
def insert(originalfile,string):
   with open(originalfile,'r') as f:
       with open('newfile.txt','w') as f2:
```

```python
            f2.write(string)
            f2.write(f.read())
    os.rename('newfile.txt',originalfile)


# @TODO: figure out a try except for the format of the numpy array printed out
# Prints a data frame
#
# Parameters:
# -----------
# @param data: dataframe to be printed
#
# Returns:
# ---------
# None
#
def printDataFrame(data):
    numpy_array = data.to_numpy()
    numpy_array = [i for i in numpy_array if str(i) != 'nan']

    try: np.savetxt(dataFrameFileName, numpy_array, fmt = "%d")
    except:
        try: np.savetxt(dataFrameFileName, numpy_array, fmt = "%s")
        except:
            try: np.savetxt(dataFrameFileName, numpy_array, fmt = "%f")
            except: print("error in types")

    columnNames = ""
    for i in data.columns:
        columnNames = columnNames + i + " "
    columnNames = columnNames + "\n"
    insert(dataFrameFileName, columnNames)


if debug == True:
    debugFd.write("Helper Functions Cell Completed...\n")
    debugFd.write(lineString+"\n")
```

Imports, configuration, and global variables:

```
# Library Imports
from nis import cat
from re import X
import pandas as pd                    # Used for data frame
import plotly                          # Saves html plots
import plotly.express as px            # Used for displaying plots
import os                              # Allows file manipulation and console debugging for offline jupyter
import numpy as np
from scipy import stats                # Used for outliers
import matplotlib.pyplot as plt        # Used for pyplot heatmap plotting
import seaborn as sns                  # Used for showing heatmap
from statsmodels.stats.outliers_influence import variance_inflation_factor
from sklearn.ensemble import ExtraTreesClassifier
from sklearn.feature_selection import SelectKBest, chi2
from sklearn import linear_model
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import lightgbm as lgb
from sklearn.preprocessing import MinMaxScaler, LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_curve, roc_auc_score, confusion_matrix

import phik

from phik import resources
from phik.binning import bin_data
from phik.report import plot_correlation_matrix

# Module Imports

# Software configurations
pd.options.display.max_rows = 4000  # Allows better debugging analysis

# Global Variables
debug = True                           # Displays additional logging output
saveImages = False                     # Saves plot image files
targetColumnName = "TARGET"            # Name for column denoting dependant variable
outlierThreshold = 3                   # Number of standard deviations from which data will be classified
as an outlier
dropMissingValues = False
datasetName = './dataset/application_train.csv'
initialDataFileName = './outputFiles/initialData.txt'
```

```
missingValFileName = './outputFiles/missingValueSummary.txt'
noMissingValuesFileName = './outputFiles/noMissingValueSummary.txt'
initialDistributionFileName = './outputFiles/initialDistribution.txt'
lineString =
"-------------------------------------------------------------------------------------------------------
--\n"
```