Code Cleanup

So the first step was to merge the student data from 2011-2017 all together before cleaning it, we first created a function to load all the csv files first:

def load\_csv\_files\_from\_folder(path):

files = os.listdir(path)

# Create a list of DataFrames from all files with .csv extension

df\_list = [pd.read\_csv(os.path.join(path, file)) for file in files if file.endswith(".csv")]

# Return concatenated list from the list of DataFrames

return pd.concat(df\_list, ignore\_index = True)

We then imported the student static, financial aid, and student progress data frames and replaced financial aid’s student ID column index (to prevent unnecessary duplicates). We then joined everything together:

# Import student static data DataFrames

static\_df = load\_csv\_files\_from\_folder(path + "Student Static Data")

# Import financial aid DataFrame

financial\_df = pd.read\_excel(path + "fin\_aid\_fasfa\_data.xlsx")

# Replace student ID column index to standardize StudentID

financial\_df.rename(columns = {"ID with leading" : "StudentID"}, inplace = True)

# Import all student progress DataFrames

progress\_df = load\_csv\_files\_from\_folder(path + "Student Progress Data")

# Merging student financial aid, static data, and progress DataFrames into one

merged\_df = dropout\_df.merge(static\_df, on = "StudentID", how = "left")

merged\_df = merged\_df.merge(financial\_df, on = "StudentID", how = "left")

merged\_df = merged\_df.merge(progress\_df, on = "StudentID", how = "left")

# Display the final merged DataFrame

merged\_df

Now getting into how we wanted to clean up the data, we first decided to keep only the last term information that a student attended, using .dropduplicates to keep the order:

# Cleaning dataset and only keep the last term each student attended

last\_term\_attended = merged\_df.groupby("StudentID").tail(1)

# Ensuring ordering of StudentIDs maintained as in original dataset

cleaned\_df = merged\_df[merged\_df["StudentID"].isin(merged\_df["StudentID"])].drop\_duplicates("StudentID")

We then checked for any null values in the csv:

missing = cleaned\_df.isnull().sum()

missing\_percent = (missing / len(cleaned\_df)) \* 100

# Create a DataFrame to view missing values and percentage

missing\_df = pd.DataFrame({"Missing Values" : missing, "Percentage" : missing\_percent})

missing\_df\_sorted = missing\_df[missing\_df["Missing Values"] > 0].sort\_values(

by = "Percentage", ascending = False)

We found that the values for Campus, and 201X Work/Study categories to be almost completely empty (98+%). We ultimately decided on the threshold for throwing out a column to be if the amount of missing values was 60% or higher of the column’s possible values.

threshold = 60

# Drop columns with missing values with greater percentage than these

columns\_drop = missing\_df\_sorted[missing\_df\_sorted["Percentage"] > threshold].index

cleaned\_df = cleaned\_df.drop(columns = columns\_drop)

We decided on the removal of geographical data and birth year, as they didn’t seem particularly relevant to what we were looking for.

# Remove geographical data as well as BirthYear

columns\_remove = ["Address1", "City", "State", "Zip", "BirthYear"]

cleaned\_df = cleaned\_df.drop(columns = columns\_remove)

We also decided to remove the “cohort” related columns, as they seemed potentially redundant, as well as a few others on a similar basis to birth year and geographic information (birth month, registration date, etc.) and decided to consolidate the different “race” columns to be a bit more concise.

# Drop the potentially redundant categorical variables

columns\_drop = ["Cohort\_x", "Cohort\_y", "AcademicYear", "cohort"]

cleaned\_df = cleaned\_df.drop(columns = columns\_drop)

# One-hot encode race variables into one "Race" feature

races = ["Hispanic", "AmericanIndian", "Asian", "Black", "NativeHawaiian", "White", "TwoOrMoreRace"]

# One-hot encoding the race columns into numerical labels

cleaned\_df["Race"] = cleaned\_df[races].idxmax(axis = 1).replace({

"Hispanic" : 1, "AmericanIndian" : 2, "Asian" : 3, "Black" : 4,

"NativeHawaiian" : 5, "White" : 6, "TwoOrMoreRace" : 7

})

# Drop the original Race binary labeled columns

cleaned\_df.drop(races, axis = 1, inplace = True)

# Drop additional columns such as birth month, cohort term, registration date, etc.

columns\_remove = [

"CohortTerm\_x", "BirthMonth", "FirstGen", "HSGPAUnwtd", "HSGPAWtd", "FirstGen", "DualHSSummerEnroll",

"CumLoanAtEntry", "cohort term", "CohortTerm\_y", "Term", "Complete2", "TransferIntent", "DegreeTypeSought"

]

Another important thing to look out for was categories/columns that only have 1 value, such as “DegreeTypeSought”, which all have the same value. These won’t offer any meaningful contribution to our models, they are constants.

# Identify columns that have the same values within all rows

same\_values = [col for col in cleaned\_df if cleaned\_df[col].nunique() == 1]

# Drop columns that all have the same values

cleaned\_df.drop(same\_values, axis = 1, inplace = True)

Exploratory Data Analysis

Feature Engineering

We decided to impute values that either have a somewhat substantial number of missing values under the threshold of 60% (such as: Marital Status, Parent’s Highest Grade Level, and Personal/Parent Gross Income), as well as a few others that didn’t seem to be missing (Why? Please elaborate for NumColCredAttemptTransfer, HighDeg, etc). Our logic for this is that this will help smooth out our missing data values and give us more to work with. Additionally, negative numbers in this data set might complicate some of our calculations, so we believed it would be best to use median values instead.

# Check for missing values in the remaining dataset

remaining\_missing = cleaned\_df.isnull().sum()

remaining\_missing\_df = pd.DataFrame({"Remaining Missing Values" : remaining\_missing})

remaining\_missing\_df = remaining\_missing\_df[remaining\_missing\_df["Remaining Missing Values"] > 0]

# Import additional dependencies

from sklearn.impute import SimpleImputer

# Create a imputer for "most\_frequent" for categorical columns

categorical\_imputer = SimpleImputer(strategy = "most\_frequent")

categorical\_columns = ["Marital Status", "Father's Highest Grade Level", "Mother's Highest Grade Level"]

# Create a imputer for "median" for numerical columns

numerical\_imputer = SimpleImputer(strategy = "median")

numerical\_columns = ["Adjusted Gross Income", "Parent Adjusted Gross Income"]

# Impute the categorical columns

cleaned\_df[categorical\_columns] = categorical\_imputer.fit\_transform(cleaned\_df[categorical\_columns])

# Impute the numerical columns

cleaned\_df[numerical\_columns] = numerical\_imputer.fit\_transform(cleaned\_df[numerical\_columns])

# Impute the "Housing" column with most frequent value

cleaned\_df["Housing"] = categorical\_imputer.fit\_transform(cleaned\_df[["Housing"]])

# Additional Cleaning for Features which Don't Appear to be Missing

columns\_impute = ['NumColCredAttemptTransfer', 'NumColCredAcceptTransfer', 'HighDeg']

median\_values = {}

for col in columns\_impute:

valid\_values = cleaned\_df[(cleaned\_df[col] != -1) & (cleaned\_df[col] != -2)][col]

median\_values[col] = valid\_values.median()

# Replacing -1 values with median values in dataset

for col, median\_val in median\_values.items():

cleaned\_df.loc[cleaned\_df[col] == -1, col] = median\_val

Models General

Besides the loading of our cleaned data csv present in all of our modeling attempts,

For all base dependencies with every model we’ll need:

import numpy as np

import pandas as pd

And import the sklearn.model related dependencies depending on which model we are using.

from sklearn.tree import #Insert Model Related Dependencies Here

The general layout for all of the models is to first remove the dropout variable (as it is the dependent variable, and we wish to use the other variables in the cleaned and merged data frame as predictors) and then to split and train the sets,

# Set the features and target variables for the dataset

X = dropout\_df.drop("Dropout", axis = 1)

y = dropout\_df["Dropout"]

# Split the data into training and test datasets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size = 0.2)

Then create a classifier object,

# Creating a [Insert Model] Classifier object

tree = [Model]Classifier()

We then fit it to the model to the training data,

# Fit the Decision Tree Classifier onto the training data

[model].fit(X\_train, y\_train)

Make predictions with the classifier,

# Make predictions using the Model Classifier

y\_pred = tree.predict(X\_test)

And take accuracy measures.

# Calculate F-beta score with beta = 2 for [model] classifier

f\_beta = fbeta\_score(y\_test, y\_pred, beta = 2)

# Display the F-beta score

print(f"Base [Model] F-beta Score (beta = 2): {f\_beta:.5f}")

Why do we pick a beta of 2/beta scored of .5f?

(Does this matter?)

We then make predictions on the kaggletest DataFrame and make a new column in it for our newly generated predictions, and ultimately submit the trained dataset for evaluation/submission.

# Make predictions on the kaggletest DataFrame using [Model]

y\_pred = tree.predict(kaggletest\_df)

# Create a new column in kaggletest DataFrame for predictions

kaggletest\_df["Dropout"] = y\_pred

# Splice the kaggletest DataFrame to only include StudentID and Dropout label

submission\_df = kaggletest\_df[["StudentID", "Dropout"]]

# Export submission DataFrame to submission folder as submission

submission\_df.to\_csv(path + "submissions/[Sumbission\_name].csv", index = False)

Decision Tree

Specifically for the decision tree and logistic regression models there are 2 accuracy measures, the inbuilt one for the trained model, and the f\_beta score shared with all models.

# Make predictions using the Decision Tree Classifier

y\_pred = tree.predict(X\_test)

# Define the accuracy as well as the inbuilt classification report within the trained model

accuracy = accuracy\_score(y\_test, y\_pred)

report = classification\_report(y\_test, y\_pred)

print(f"Base Decision Tree Classifier Accuracy: {accuracy:.3f}")

Base Decision Tree Classifier Accuracy: 0.736

Base Decision Tree F-beta Score (beta = 2): 0.65939

Specifically for decision trees we chose an inbuilt accuracy of .3f (Again I have no idea if this is relevant or how/why)

This is because the latter 2 submissions already have consistency measures by design, the random forest model and gradient boosted random forest model already is created on multiple decision tree-like models, and thus doesn’t require an inbuilt accuracy measure.

Logistic Regression

Specifically for logistic regression we chose an inbuilt accuracy of .5f (Again I have no idea if this is relevant or how/why)

# Define accuracy score for the trained model

accuracy = accuracy\_score(y\_test, y\_pred)

print(f"Logistic Regression Accuracy Score: {accuracy:.5f}")

Logistic Regression Accuracy Score: 0.65267

Logistic Regression F-beta score (beta = 2): 0.26232

Random Forest

Random Forest F-beta score (beta = 2): 0.77839

Random Forest (Gradient Boosted)

XGBoost Classifier F-beta Score (beta = 2): 0.77980