HUDK4050 Midterm Project: Student Dropout Prediction Report

Nicholas Lin, Ethan Hiew, Mia Li, Carla Silva Hounshell

Pre-Processing and Feature Engineering

```
[ ] # Define a new function to load all CSV files from within a folder
    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)
```

```
[ ] # 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
```

Given the directory of the datasets we first had to merge all of the data together on “StudentID” and then merge the student dropout, static, progress and financial data together. This resulted in a dataset with **52,968 students and 84 features.** Clearly we began with 12,261 student IDs so the number of student data was too great. We then cleaned the dataset so we only keep the last term that the student attended and this resulted in a cleaned dataset with the original **12,261** students represented.

```
[ ] # 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")

    cleaned_df
```

We then focused on removing missing values within the dataset. We first set a threshold of 60% and removed all data that was missing more than 60% of values. We then imputed any missing data in two ways depending on whether the missing data was numerical or categorical. For numerical data we simply imputed the median value for each feature and for categorical data we imputed the most frequent data for each feature.

```
# 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"]
```

Finally we thoroughly examined each variable, determined the range of values, and determined whether some values which hadn't been considered to be missing were in fact missing. For example for a feature such as Number of College Credits Transfer Students Attempted a value of -2, although numerical, symbolized that the value was actually missing in the dataset. Given this we then moved to impute these values with the median values within each feature. Finally we one-hot encoded the race variables into one "Race" feature before dropping additional columns we felt would not contribute to model training significantly.

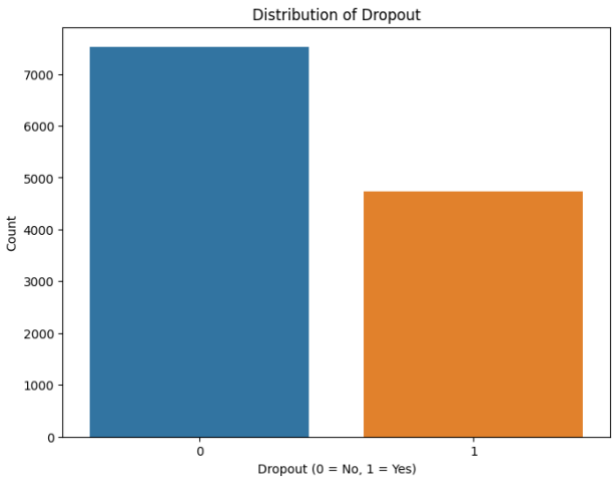
```
[ ] # One-hot encode race variables into one "Race" feature
    races = ["Hispanic", "AmericanIndian", "Asian", "Black", "NativeHawaiian", "White", "TwoOrMoreRace"]

    # 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"
    ]
```

Exploratory Data Analysis

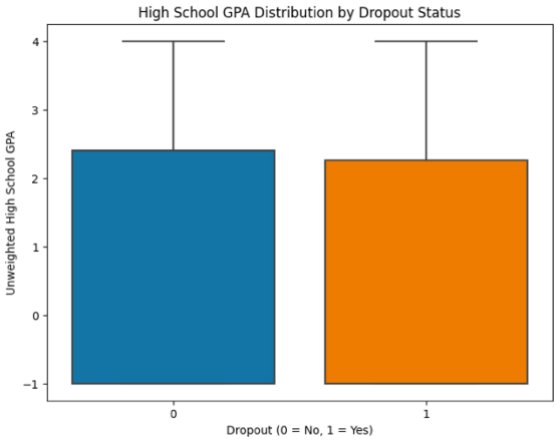
There were a variety of aspects of the data we wanted to explore. First we visualized the imbalance within the target class.

```
# Plotting Target Variable
plt.figure(figsize = (8, 6))
sns.countplot(x='Dropout', data = cleaned_df)
plt.title('Distribution of Dropout')
plt.xlabel('Dropout (0 = No, 1 = Yes)')
plt.ylabel('Count')
plt.show();
```



This allowed us to see if the target class was skewed within the dataset. The degree of the class imbalance would allow us to create strategies that could be more appropriate for model training. For example we could consider resampling strategies or other performance metrics which could specifically be applied to imbalances. Upon analysis though we concluded that although there was a slight imbalance the imbalance within the target class was quite negligible.

We then wanted to see the distribution of the student's high school GPAs to see how high school could relate to student dropout rates in college:

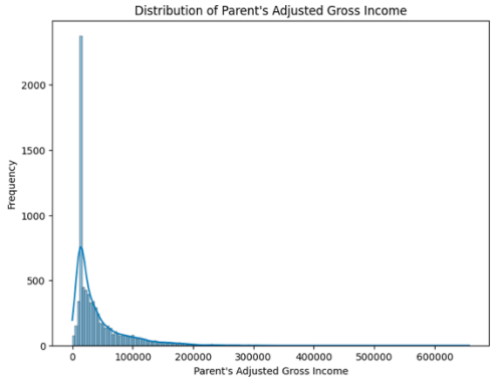



```
# Plotting GPA distributions among students
plt.figure(figsize = (8, 6))
sns.boxplot(x = 'Dropout', y = 'HSGPAUnwtd', data = cleaned_df)
plt.title('High School GPA Distribution by Dropout Status')
plt.xlabel('Dropout (0 = No, 1 = Yes)')
plt.ylabel('Unweighted High School GPA')
plt.show();
```

Variations in GPA could indicate that there was an importance in previous academic performance as a predictor for future dropout. However it seems as if the distribution of high school GPAs were almost identical for students who dropped out and those that didn't. This was quite surprising and served to help us consider other features that could more accurately capture the relationship between features and the target variable.

Finally we wanted to see how parental income affected student dropout rates:

It can be seen that this data is incredibly right-skewed. Of course since it is financial earning data, this is not surprising as income is mostly right-skewed. However there is something to be said that that a large majority of students that dropped out are closest to the bottom end of the spectrum of earning data. Of course as discussed earlier earning data is inherently right-skewed but this could be an interesting relationship to explore.




```
# Plotting distribution of parental gross income through dropout variance in students
plt.figure(figsize = (8, 6))
sns.histplot(cleaned_df[cleaned_df['Parent Adjusted Gross Income'] > 0]['Parent Adjusted Gross Income'],
             kde = True)
plt.title("Distribution of Parent's Adjusted Gross Income")
plt.xlabel("Parent's Adjusted Gross Income")
plt.ylabel('Frequency')
plt.show();
```

Model Training

Logistic Regression

Our first choice of a model was a logistic regression model as it is a standard powerful linear model for binary classification tasks. And considering our task was student dropout prediction, logistic regression made sense as the first step.

```
[ ] # Create a logistic regression object
    lr = LogisticRegression(solver = "liblinear")

    # Fit the logistic regression model onto the training data
    lr.fit(X_train, y_train)
```

```
LogisticRegression(solver='liblinear')
```

```
[ ] # Make predictions using the Logistic Regression model
    y_pred = lr.predict(X_test)

    # 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
```

```
[ ] # Calculate F-beta (beta = 2) score for logistic regression model
    f_beta = fbeta_score(y_test, y_pred, beta = 2)

    print(f"Logistic Regression F-beta score (beta = 2): {f_beta:.5f}")
```

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

However we ran into issues considering how many features were present within the dataset and this made it so the logistic regression fit very poorly to the dropout label prediction. Although dropout prediction was a binary classification task, logistic regression did not result in the robust performance we sought.

Decision Tree

Decision Trees provide clear interpretation of relationships between features. Most importantly decision trees allow for the capture of non-linear relationships within the data.

```
[ ] # Creating a Decision Tree Classifier object
    tree = DecisionTreeClassifier()

    # Fit the Decision Tree Classifier onto the training data
    tree.fit(X_train, y_train)
```

```
DecisionTreeClassifier()
```

```
[ ] # 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
```

```
[ ] # Calculate F-beta score with beta = 2 for base decision tree classifier
    f_beta = fbeta_score(y_test, y_pred, beta = 2)

    # Display the F-beta score
    print(f"Base Decision Tree F-beta Score (beta = 2): {f_beta:.5f}")
```

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

Usually decision trees have issues with overfitting however in this dataset this was not a pressing issue and we also addressed this by utilizing cross-validation to find the optimal hyperparameters. We found that decision trees performed better than our initial logistic regression but not enough to our standard of deploying a robust model.

Random Forest Classifier

Random Forests are an ensemble of decision trees and because of this we decided to implement this next.

```
# Creating a Random Forest object
forest = RandomForestClassifier()

# Train Random Forest model on the cleaned data
forest.fit(X_train, y_train)
```

```
▾ RandomForestClassifier
RandomForestClassifier()
```

```
# Make predictions on the test dataset using Random Forest
y_pred = forest.predict(X_test)

# Calculate F-beta score (beta = 2) of Random Forest predictions
fbeta_scored = fbeta_score(y_test, y_pred, beta = 2)

print(f"Random Forest F-beta score (beta = 2): {fbeta_scored:.5f}")
```

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

Although overfitting wasn't our primary concern, random forests also serve to prevent overfitting. We used cross-validation in order to determine the best hyperparameters to train our model on and resulted in significantly better performance than the lone decision tree had been able to produce. We then performed hyperparameter tuning to improve the performance of the Random Forest Classifier:

```
[ ] # Re-instantiate the Random Forest model
    forest = RandomForestClassifier()

    # Defining a F-beta score as a base metric for hyperparameter tuning
    def custom_fbeta_scorer(y_true, y_pred):
      return fbeta_score(y_true, y_pred, beta = 2)

    f2_scorer = make_scorer(custom_fbeta_scorer)

    # Setting up GridSearchCV to find best hyperparameters for Random Forest
    param_grid = {
        'n_estimators': [100, 200, 300],
        'max_depth': [10, 20, 30],
        'min_samples_split': [2, 5, 10],
        'min_samples_leaf': [1, 2, 4]
    }
    grid_search = GridSearchCV(estimator = forest, param_grid = param_grid,
                               scoring = f2_scorer, cv = 5)
```

```
[ ] # Fitting the model
    grid_search.fit(X_train, y_train)

    # Best parameters and best F-beta score
    best_params = grid_search.best_params_
    best_score = grid_search.best_score_

    best_params, best_score
```

```
({'max_depth': 30,
  'min_samples_leaf': 4,
  'min_samples_split': 10,
  'n_estimators': 300},
 0.7759919956365471)
```

Which resulted in the model with the optimal hyperparameters outputting a F-beta score of approximately **0.776.**

XGBoost Classifier

Our final effort to achieve the highest F-beta score with beta = 2 was by implementing a XGBoost classifier as it is well regarded for high performance, efficiency, as well as customizable hyperparameter tuning.

```
[ ] # Create a XGBoost Classifier object
    xgb_classifier = xgb.XGBClassifier()

    # Training the XGBoost Classifier on the cleaned data
    xgb_classifier.fit(X_train, y_train)
```

```
XGBClassifier(base_score=None, booster=None, callbacks=None,
              colsample_bylevel=None, colsample_bynode=None,
              colsample_bytree=None, device=None, early_stopping_rounds=None,
              enable_categorical=False, eval_metric=None, feature_types=None,
              gamma=None, grow_policy=None, importance_type=None,
              interaction_constraints=None, learning_rate=None, max_bin=None,
              max_cat_threshold=None, max_cat_to_onehot=None,
              max_delta_step=None, max_depth=None, max_leaves=None,
              min_child_weight=None, missing=nan, monotone_constraints=None,
              multi_strategy=None, n_estimators=None, n_jobs=None,
              num_parallel_tree=None, random_state=None, ...)
```

```
[ ] # Make predictions on the test set
    y_pred = xgb_classifier.predict(X_test)

    # Calculate F-beta score (beta = 2) for the XGBoost model
    fbeta_scored = fbeta_score(y_test, y_pred, beta = 2)
```

```
[ ] # Display the F-beta score
    print(f"XGBoost Classifier F-beta Score (beta = 2): {fbeta_scored:.5f}")

    XGBoost Classifier F-beta Score (beta = 2): 0.76492
```

This turned out to be a challenge however as the model's complexity made it so we had to conduct extensive hyperparameter tuning using GridSearchCV as well as RandomizedSearchCV in order to optimize our model. However this did result in slightly higher performance than our optimized Random Forest classifier so it was a positive development:

```
({'subsample': 0.7,
  'n_estimators': 400,
  'min_child_weight': 2,
  'max_depth': 3,
  'learning_rate': 0.2,
  'colsample_bytree': 0.6},
 0.7869556390357294)
```

Training Optimal XGB Classifier

```
[ ] # Instantiate the optimal XGBoost classifier with optimal hyperparameters
    xgb_model = XGBClassifier(
        subsample = 0.7,
        n_estimators = 400,
        min_child_weight = 2,
        max_depth = 3,
        learning_rate = 0.2,
        colsample_bytree = 0.6,
        use_label_encoder = False,
        eval_metric = "logloss"
    )

    # Fit the model to the training data
    xgb_model.fit(X_train, y_train)
```

Essentially the hyperparameter-optimized XGBoost Classifier resulted in the best performance in determining student dropout rates. However it was only slightly better in terms of F-beta score when compared to the hyperparameter-optimized Random Forest Model. Going into the model training, I had assumed that either the Random Forest or the XGBoost Classifier would result in the best performance. XGBoost did have the slight edge because of how easy and complex the hyperparameter tuning could be in the model training as well as evaluation. Overall though the F-beta score achieved by both the hyperparameter-optimized Random Forest Model and XGBoost Classifier was around 80%.