# **CS 634 Data Mining Final Project**

Dina Lahham

11/17/24

## **Background**

In this project, we will analyze a dataset that categorizes participants into seven groups: Insufficient Weight, Normal Weight, Overweight I, Overweight II, Obese I, Obese II, and Obese III. For the purposes of this study, we reclassed these groups into two umbrella categories: Normal (combining "Insufficient" and "Normal Weight") and Overweight (combining "Overweight" and "Obese"). This reclassification allowed us to create a binary classification problem.

The aim of this project is to use three machine learning models - Random Forest, KNN, and GRU - to predict the likelihood that a participant is classified as Normal or Overweight based on a set of attributes. To evaluate the performance of these models, we will use several metrics, including recall, precision, accuracy, F1 score, error rate, AUC, ROC, TSS, and HSS.

The original dataset contains the following attributes: id, gender, age, height, weight, family\_history\_overweight, freq\_high\_cal\_food, freq\_veg\_consump, meals\_per\_day, alch\_in\_excess, smoker, water\_consump, calorie\_monitor, freq\_physical\_activity, time\_using\_tech, alch\_consump, mode\_of\_trans, label\_cat, and label. In this project, we will explain how we selected relevant features for the model, dropped attributes that were overly specific or irrelevant, and converted categorical attributes into numerical values.

## **Source Data & Links**

* This data is from Kaggle’s Multi-Class Prediction of Obesity Risk dataset
  + <https://www.kaggle.com/competitions/playground-series-s4e2/data>
* This analysis can also be viewed on GitHub:
  + https://github.com/dina-git-njit/dina\_lahham\_finalproj

## **Requirements**

* Python 3.11 or higher

## **Setup**

1. **Download the zipped file** titled ‘dina\_lahham\_finalproj.zip’.
2. **Unzip the file** by right-clicking and pressing ‘Extract All’ and choose a directory to place it in. Please keep track of this directory as we will be using it again.
3. **Go to the cmd shell**. An easy way to get here is by pressing the Windows button and typing in: “cmd” (no apostrophes needed), then press enter.
4. **Copy the address from your directory where the zip file has been extracted** then type this into your command shell. I have an example below of my directory for reference. The command line will now reflect your new file location for the next command.

“cd C:\ADDRESS\EXAMPLE” (no apostrophes needed).

C:\Users\15514>**cd C:\Users\15514\OneDrive\Documents\Fall 2024\dina\_lahham\_finalproj**

C:\Users\15514\OneDrive\Documents\Fall 2024\ dina\_lahham\_finalproj>

1. **Start a virtual environment** using these commands

C:\Users\15514\OneDrive\Documents\Fall 2024\ dina\_lahham\_finalproj> **python -m venv venv**

C:\Users\15514\OneDrive\Documents\Fall 2024\ dina\_lahham\_finalproj> **venv\Scripts\activate**

1. **Install Jupyter Notebook** if you do not already have it installed. This is as simple as typing the following into your command line.

You may also need to upgrade pip.

C:\Users\15514\OneDrive\Documents\Fall 2024\ dina\_lahham\_finalproj**> pip install jupyter**

C:\Users\15514\OneDrive\Documents\Fall 2024\ dina\_lahham\_finalproj**> python.exe -m pip install --upgrade pip**

1. **Run Juptyer** **Notebook** with this command. This command will automatically run jupyter in your default web browser using the local host link (http://localhost:8888/tree)

C:\Users\15514\OneDrive\Documents\Fall 2024\ dina\_lahham\_finalproj **> jupyter notebook**

1. **Navigate to the main.ipynb file** in the Jupyter environment by double clicking on the file

A screenshot of a computer

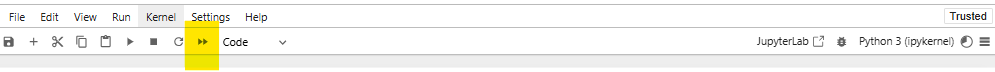
Description automatically generated

## **Running the Program**

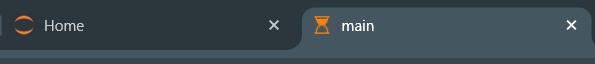
1. **Press the Restart Kernel button** highlighted in yellow. This will begin the program.

Please note that this may take about an hour to complete.

**The kernel may appear to not be loading while it is installing all the libraries. Please be patient with this step. After a few minutes, you will see a loading symbol appear in the tab name.**







### Step 1: Prepare Data

1. The program will install all packages and import the applicable libraries.

A screenshot of a computer

Description automatically generated

1. The program will read the .csv



1. **Dummy Variables:** Later in this analysis (Step 2), we will explain the criteria used to create dummy variables for certain columns. The goal was to convert categorical attributes into numerical ones. Since these attributes have only 2-3 unique values, this transformation adds only a small number of additional columns to our dataset.
2. **Label:** We will also transform the "label" attribute to a 1 or 0.

**A white background with red text

Description automatically generated**

1. **Remove irrelevant data:** We removed ID, label, the label\_cat attribute which categorized the weight label 7 ways. Weight was removed as it would skew the outcome. Although height would not influence the outcome it was removed so that this study can focus on behavioral lifestyle attributes.



1. **Prepare K-10 Fold:** Lastly, we will use the Stratified Fold command to split the data into 10 training folds.



### Step 2: Summary Stats

This step will help us understand the original data set.

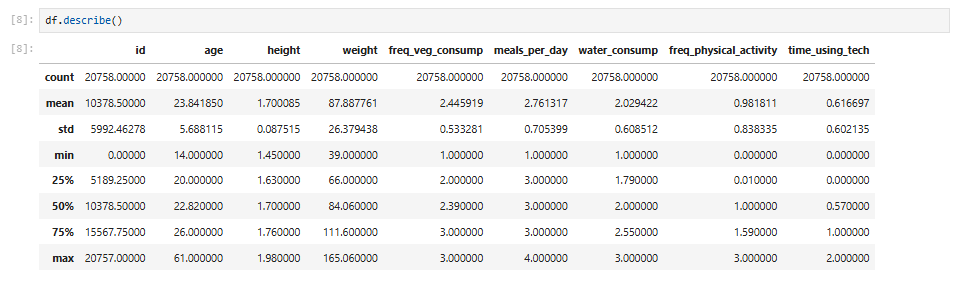
1. The first step is to read the .csv as a data frame.



1. The **describe function** will give us some top line stats to help understand the numerical attributes.

There appears to be 20,758 rows of data. The average age of a participant in this study is 23.8 years old. Although the eldest participant is 61, 75% are 26 or younger. Height is in meters and weight is in kilograms. The average respondent claims to eat around 2.5 servings of vegetables per day. Respondents are also averaging about 2.8 meals per day, with the lower end at 1 meal per day and the highest at 4. We see that physical activity is right under about an hour per day per respondent. Lastly, the time spent using technology ranges from 0 to 2 hours, with an average at a little over half an hour.

This data was collected from Mexico, Peru, and Colombia which explains why some of these answers might slightly differ from the average American's response.



1. The **info function** will allow us to understand the data types available

Aside from the numerical attributes above, we also have categorical ones.

**A screenshot of a computer

Description automatically generated**

1. This loop will show us all the **unique values per attribute¶**

Here, we will print all the unique column options in the categorical attributes.

This data was initially separated into 7 different label categories with one class for "Insufficient Weight", another for "Normal Weight", 2 classes for "Overweight", and 2 classes for "Obese".

For the purposes of this study, we classed Insufficient and Normal into a new label called Normal. Then, the Overweight and Obese categories were grouped into one value called Overweight.

A screenshot of a computer program

Description automatically generated

1. Finally, the **head function** will give us a small glimpse into the raw data

A screenshot of a computer

Description automatically generated

1. **Visualize the distribution of Normal vs Overweight categorization in the data using the count plot function**

The data is composed of 27% labeled as "Normal" and 72% as "Overweight."

Although it may seem that normalization is necessary given this distribution, it had little to no effect on this data.

In Random Forest, normalization isn't usually needed because it’s a tree-based model that focuses on data structure and splits rather than feature scales.

For KNN and GRU, normalization usually helps because it’s a distance-based model, but in this case, the feature ranges are already very close that scaling did not impact results.

In summary - normalization helps smooth the data when the distance between points varies widely, but in this case the data points already hovered in a smaller range, so normalization is less critical here.

A screenshot of a computer

Description automatically generated

1. **Identify Correlation**

To identify and reduce redundancy in our data, we created a correlation matrix to find relationships between features.

In the correlation matrix below, we included all dummy variables for categorical data, resulting in a large and complex matrix that’s difficult to interpret. Additionally, binary attributes (such as Yes/No variables) show perfect correlations with each other, which can clutter the matrix. To create a cleaner version, we will drop the first dummy variable for each category in the following matrix.

A screen shot of a computer

Description automatically generated

1. In this **refined correlation matrix**, there are still strong correlations among some dummy variables with three or more unique values. However, we can now clearly observe that the other features have low correlations. This confirms that the current set of attributes is **sufficiently distinct**.

A screenshot of a computer

Description automatically generated

1. **Histograms of Attributes**

Again, since some of our attributes are categorical and some are numerical, the following code will allow us to understand how the data is distributed per attribute.

Histograms are blue and bar charts are red.

The most notable findings from the histograms are that this data appears to be collected from participants mainly in the 20-30 age range.

There is an even split of male to female participants.

Height is the only attribute that follows a bell curve structure.

The family history overweight attribute follows a similar pattern to the label we saw a few steps earlier.

Most participants are not smokers, do not monitor calories, and report eating 3 meals a day.

Like mentioned prior, since this data was collected from Colombia, Peru, and Mexico we see that the main mode of transport is Public Transport.

A computer code on a white background

Description automatically generated

A screenshot of a graph

Description automatically generated

1. **Pair plots**

Pair plots visualize relationships between pairs of features in a dataset by displaying scatter plots for each feature pair. They help identify clusters and class separations. Here we will visualize the pair plots for our numeric attributes.

As expected, weight perfectly clusters with "Normal" to one side and "Overweight" to the other side.

Height, vegetable consumption, water consumption, and time spent using technology do not seem to follow any clear patterns.

On the other hand, meals per day saw clusters across all variables for "Normal" in the 3-4 range.

A screenshot of a computer screen

Description automatically generated

### Step 3: Parameter Tuning

#### Random Forest

For Random Forest, we will optimize the number of trees in the forest as well as the minimum number of splits in a node. This is different to the number of splits in the k-fold discussed earlier. The purpose of it is to control the growth of individual trees.

We use the GridSearchCV function to test different combinations of estimators and minimum sample splits. After evaluating each combination by calculating the ROC-AUC metric, this code prints the best combination.

After running the optimization below, we find that a higher number of trees/estimators and a higher sample split threshold yields the highest AUC score, specifically 200 trees, and a minimum split threshold of 10.

A screenshot of a computer

Description automatically generated

#### KNN

Like the random forest fine tuning, this code looks for k-values in the range of 1 to 39. The metric it uses to choose the optimal k-value is 10-fold average AUC. In this case we found that k-value to be 28.

A screenshot of a computer program

Description automatically generated

#### GRU

First, what is GRU? GRU stands for Gated Recurrent Unit. It is a type of neural network designed for sequential data. It works by remembering important information from earlier steps while "forgetting" less relevant details. Fine-tuning GRU helps the model decide the right balance between what to remember and what to forget. While our data isn’t naturally sequential, we can reshape it to treat features as sequential steps. Although GRU isn’t traditionally used for this type of data, reshaping it into vectors has proven to be a reasonable approach. It is likely that the attributes in this study heavily rely on one another in a time-series like dependency which enabled the algorithm to work as well as it did, as you will see in Step 4.

For our GRU model, we’ll focus on optimizing three key parameters: units, dropout, and learning rate.

Units represent the number of neurons in the GRU layer. More units allow the model to capture complex patterns, while fewer units make the model simpler and faster but may lead to underfitting.

Dropout refers to the rate at which the model “forgets” certain connections during training. A higher dropout rate can help prevent overfitting but may also risk underfitting if set too high.

Learning rate controls the size of each adjustment the model makes during training. A smaller learning rate leads to slower, more stable learning, while a larger one speeds up training but risks overshooting the optimal solution.

By tuning these parameters, we aim to build a GRU model that works well even with our reshaped non-sequential data.

Running the code below will give us **256 units**, **dropout rate of 0.4**, and **learning rate: 0.0022692999466626704.**

A screenshot of a computer program

Description automatically generated

### Step 4: Algorithms

In the last step, we will run the algorithms with their optimization parameters built in. In the screenshots below, you will see the scores for all the metrics across each fold as well as a mean score.

#### Random Forest

A screenshot of a computer

Description automatically generated

A graph of a curve

Description automatically generated

#### KNN

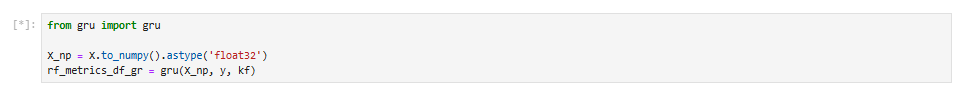
A screenshot of a computer

Description automatically generated

A graph of a curve

Description automatically generated

#### GRU



A screenshot of a computer

Description automatically generated

A graph of a curve

Description automatically generated

## Results & Conclusion

Below is a list of all the metrics used in this analysis and what to look for. In almost every case, a higher score would denote a more accurate algorithm. The exceptions are of course FPR, FNR, Error Rate, and Brier Score.

**TP:** True Positives

**TN:** True Negatives

**FP:** False Positives

**FN:** False Negatives

**P:** Total Positives

**N:** Total Negatives

**TPR:** True Positive Rate (TP/P)

**TNR:** True Negative Rate (TN/N)

**FPR:** False Positive Rate (FP/N)

**FNR:** False Negative Rate (FN/P)

**Recall:** Also referred to as the True Positive Rate. It measures the ability to identify positive cases.

**Precision:** the proportion of True Positives from the pool of both True and False Positives (TP/(TP+FP))

**F1 Score:** the balance between recall and precision. The closer this score is to 1, the better the model. (2\*(Precision \* Recall)/ (Precision + Recall))

**Accuracy:** the proportion of correct predictions (TP+TN)/(P+N)

**Error Rate:** the proportion of incorrect predictions (the inverse of accuracy: 1-accuracy)

**BACC:** the balanced accuracy is the average of the TRP and TNR

**TSS:** the difference between TPR and FPR. The closer this score is to 1, the better the model.

**HSS:** measures how much better the model is at guessing than random chance.

**Brier Score:** measures the accuracy of the predictions’ probabilities. This score measures how confident the model was before it gave its final prediction. A Brier Score closer to 0 indicates that the probabilities are more accurate.

**ROC Curve:** plots TPR vs FPR to visualize how well the model can balance these two metrics above random chance – or the line of non-discrimination.

**AUC:** the Area Under the Curve measures the area under the ROC curve. The closer this number is to 1, the better the model.

Here is how all the models did on average for the 10-Cross Fold Method:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Metric** | **RF** | **KNN** | **GRU** | **Winner** | **Runner Up** | **Third** |
| **TP** | 14,066 | 14,250 | 14,007 | KNN | RF | GRU |
| **TN** | 4,533 | 3,943 | 4,007 | RF | GRU | KNN |
| **FP** | 1,072 | 1,662 | 1,598 | RF | GRU | KNN |
| **FN** | 1,087 | 903 | 1,146 | KNN | RF | GRU |
| **P** | 15,153 | 15,153 | 15,153 |  |  |  |
| **N** | 5,605 | 5,605 | 5,605 |  |  |  |
| **TPR** | 0.928266 | 0.940408 | 0.924372 | KNN | RF | GRU |
| **TNR** | 0.808740 | 0.703473 | 0.714901 | RF | GRU | KNN |
| **FPR** | 0.191260 | 0.296527 | 0.285099 | RF | GRU | KNN |
| **FNR** | 0.071734 | 0.059592 | 0.075628 | KNN | RF | GRU |
| **Recall** | 0.928266 | 0.940408 | 0.924372 | KNN | RF | GRU |
| **Precision** | 0.929246 | 0.895594 | 0.897967 | RF | GRU | KNN |
| **F1 Score** | 0.928722 | 0.917432 | 0.910751 | RF | KNN | GRU |
| **Accuracy** | 0.895992 | 0.876433 | 0.867810 | RF | KNN | GRU |
| **Error Rate** | 0.104008 | 0.123567 | 0.132190 | RF | KNN | GRU |
| **BACC** | 0.868503 | 0.821941 | 0.819636 | RF | KNN | GRU |
| **TSS** | 0.737006 | 0.643881 | 0.639272 | RF | KNN | GRU |
| **HSS** | 0.736361 | 0.672530 | 0.655739 | RF | KNN | GRU |
| **Brier Score** | 0.074288 | 0.090850 | 0.095423 | RF | KNN | GRU |
| **AUC** | 0.956627 | 0.929346 | 0.923745 | RF | KNN | GRU |

Overwhelmingly, Random Forest won in TNR, FPR, Precision, F1 Score, Accuracy, Error Rate, BACC, TSS, HSS, Brier Score, and AUC. These results suggest that Random Forest is exceptional at a range of classification tasks – particularly in minimizing false positives and striking a balance between recall and precision. It maintains accuracy and reduces error, making it a highly reliable classifier.

While KNN won in TPR (recall) and FNR, it also won as runner up in nearly every metric that Random Forest won first place in. It excelled at identifying true positives and minimizing false negatives. It could be more beneficial in cases where the analysis at hand is particularly concerned with detecting relevant information is more important than avoiding false positives.

GRU’s wins as 2nd place came in TNR, FPR, and Precision. As stated earlier in this analysis, GRU is best fitted for time-series analysis. Since this was a thought experiment in model-building, it was worth exploring how well it would perform on non-sequential data. While its performance never won in any of the metrics, it came as a close third place.

Please note that running the tensor flow library will not give you the same results each time. Deep learning algorithms like GRU rely on some level of variability in how they initialize weights and shuffle data. We tried to control some of the variability by stabilizing dropout rate and GRU layers. So, please note that when the user runs the last algorithm, GRU may win in a few metrics like TPR and Precision. This does not diminish from the overall analysis put forth about the well-roundedness of Random Forest.

Ultimately, Random Forest emerges as the most well-rounded algorithm for this set of data. If the goal were only to maximize true positives and minimize false negatives, KNN would have a strong case. GRU would likely have performed better if the data was structured sequentially.

In future model selection, it is important to understand your use case, the data at hand, and consider which metrics best suit your end goal.