**Authors:**

Saw William Joseph

Park Kunyoung

Brent Kuan Jia Yu

Yu Ying Cheng

|  |  |
| --- | --- |
| NTU Logo [Nanyang Technological University] png    Group Project: Costa Rican Household Poverty Level Prediction (Kaggle Competition) |  |

# Contents

[Contents 1](#_Toc88767986)

[1. Introduction 3](#_Toc88767987)

[2. Data Preprocessing 4](#_Toc88767988)

[2.1. Overview of Data 4](#_Toc88767989)

[2.2. Exploratory Data Analysis 6](#_Toc88767990)

[2.3. Filling up discrepancies in data 8](#_Toc88767991)

[2.3.1. Filling up Nan values 8](#_Toc88767992)

[2.3.2. Replacing “Yes” and “No” 9](#_Toc88767993)

[2.3.3. Rectifying erroneous target labels 10](#_Toc88767994)

[2.4. Feature Engineering 11](#_Toc88767995)

[2.4.1 Binary to Ordinal 11](#_Toc88767996)

[2.4.2. Aggregating Features 12](#_Toc88767997)

[2.4.3. Removing redundancy and decreasing dimensionality 13](#_Toc88767998)

[2.5. Preparing dataset for training 16](#_Toc88767999)

[5. 3. Experiments & Results 18](#_Toc88768000)

[3.1. Models 18](#_Toc88768001)

[3.1.1 Bagging classifier 18](#_Toc88768002)

[3.1.2. Random Forest classifier 18](#_Toc88768003)

[3.1.3. Extra Trees classifier 19](#_Toc88768004)

[3.1.4. LightGBM classifier 19](#_Toc88768005)

[3.2. Training 20](#_Toc88768006)

[3.2.1. Bagging classifier 20](#_Toc88768007)

[3.2.2. Random Forest classifier 22](#_Toc88768008)

[3.2.3. Extra Trees classifier 23](#_Toc88768009)

[3.2.4. LightGBM classifier 25](#_Toc88768010)

[3.3. Predicting/Classifying 27](#_Toc88768011)

[3.3.1. Bagging classifier 27](#_Toc88768012)

[3.3.2. Random Forest classifier 28](#_Toc88768013)

[3.3.3. Extra Trees classifier 29](#_Toc88768014)

[3.3.4. LightGBM classifier 29](#_Toc88768015)

[4. Discussion 31](#_Toc88768016)

[4.1. Submission 31](#_Toc88768017)

[4.2. Results using aggregated dataset 32](#_Toc88768018)

[4.3. Learning Points 33](#_Toc88768019)

[5.Bibliography 34](#_Toc88768020)

[6. Appendix 35](#_Toc88768021)

# Introduction

As social programs have limited funding, it is crucial that resources are allocated to those that need them the most. However, many organizations are having a hard time achieving this as programs targeted to help the poorest segment of the population sometimes struggle to identify them. This problem arises as the poorest segments often lack the income and expense records to prove that they qualify for certain social programs thus being disqualified from aid despite being the ones that need them the most. As a workaround to this problem, the Inter-American Development Bank has come up with a Proxy Means Test (PMT) that aims to use other non-conventional rubrics such as materials of walls, ceilings etc. to supplement their income qualification assessments. This sets the premise for our project that is based upon the Kaggle competition “Costa Rican Household Poverty Level Prediction” which aims to improve performance and reliability of PMTs to classify households into different income groups with the purpose of assessing a household's need for social assistance.

The objective of the Kaggle competition is to accurately classify households into 1 of 4 categories of poverty level where 4 are households that are relatively well to do whereas 0 are households in the lowest poverty level in need of social assistance.

# Data Preprocessing

## 2.1. Overview of Data

By exercising some assumption and preliminary judgement, we have assigned all variables into their respective categories manually. We will mainly be focusing on the individual and household variables in further analysis. The breakdown of different features is as such:

1. Individual variables: features of each member in a household

a. Boolean:

b. Ordered

2. Household variables: features that represent all members in the household

a. Boolean

b. Ordered

c. Continuous

3. Squared variables: squared values of other features given in the dataset

4. Identification variables: variables to identify household/individual/target label

|  |  |  |
| --- | --- | --- |
| Column Type | Description | Number of Columns |
| Identification variables | Identifies the individual/household and is not useful as a feature | 2 |
| Individual boolean | Boolean attributes describing characteristics of the specific individual | 33 |
| Individual Continuous | Continuous numbered attributes describing characteristics of the specific individual | 2 |
| Household boolean | Boolean attributes describing characteristics of the specific household | 70 |
| Household ordinal | Ordered categories describing characteristics of the specific household | 20 |
| Household Continuous | Continuous numbered attributes describing characteristics of the specific household | 6 |
| Squared values | Values derived by taking a squared value from other previous features | 9 |
| Target | Ordinal Target value to be predicted | 1 |

Table 1: Feature columns in train.csv

The data is provided in the form of 2 csv files: train.csv and test.csv. For train.csv, there are a total of 143 different features falling into a few distinct categories as indicated in Fig 2.1.1. The only difference between the train and test.csv provided is the absence of a Target column in test.csv which is meant as a judgement rubric for the competition. While the dataset provides for both individual and household features, it should be noted that the final assessment criteria for the competition is for the accuracy of prediction for each household and not on each individual.

The train.csv includes a total of 9556 rows of inputs while the test.csv includes 23856 rows of inputs. As per competition format, the final prediction accuracy is evaluated by comparing only those individuals who are labelled as household heads. Within the datasets, household heads make up 31.1% and 30.7% of all individuals for train and test.csv respectively.

The dataset provided uses acronyms to arbitrarily name the features and also provides a separate csv file to provide the corresponding feature description. An example of such acronyms include “v2a1” to represent “Monthly rent payment” and “hacapo” to represent “=1 Overcrowding by rooms”. To improve interpretability, we have opted to replace the acronyms with the actual feature description.

## Exploratory Data Analysis

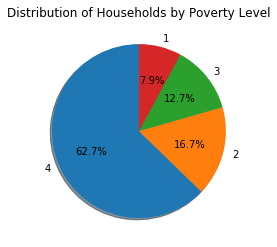


Figure : Distribution of households by poverty level (train.csv)

Where 0 represents the “poorest” poverty level while 4 represents those that are the most “well-off”, most households fall in the poverty level group of 4 and a total of 75.4% of all households are classified as a poverty level of 3 and above. Hence, it is important to account for this imbalance in classes as a large proportion of poverty level 4 will skew model predictions otherwise. This imbalance is addressed by adjusting model parameters as highlighted in the report in section 3.2.

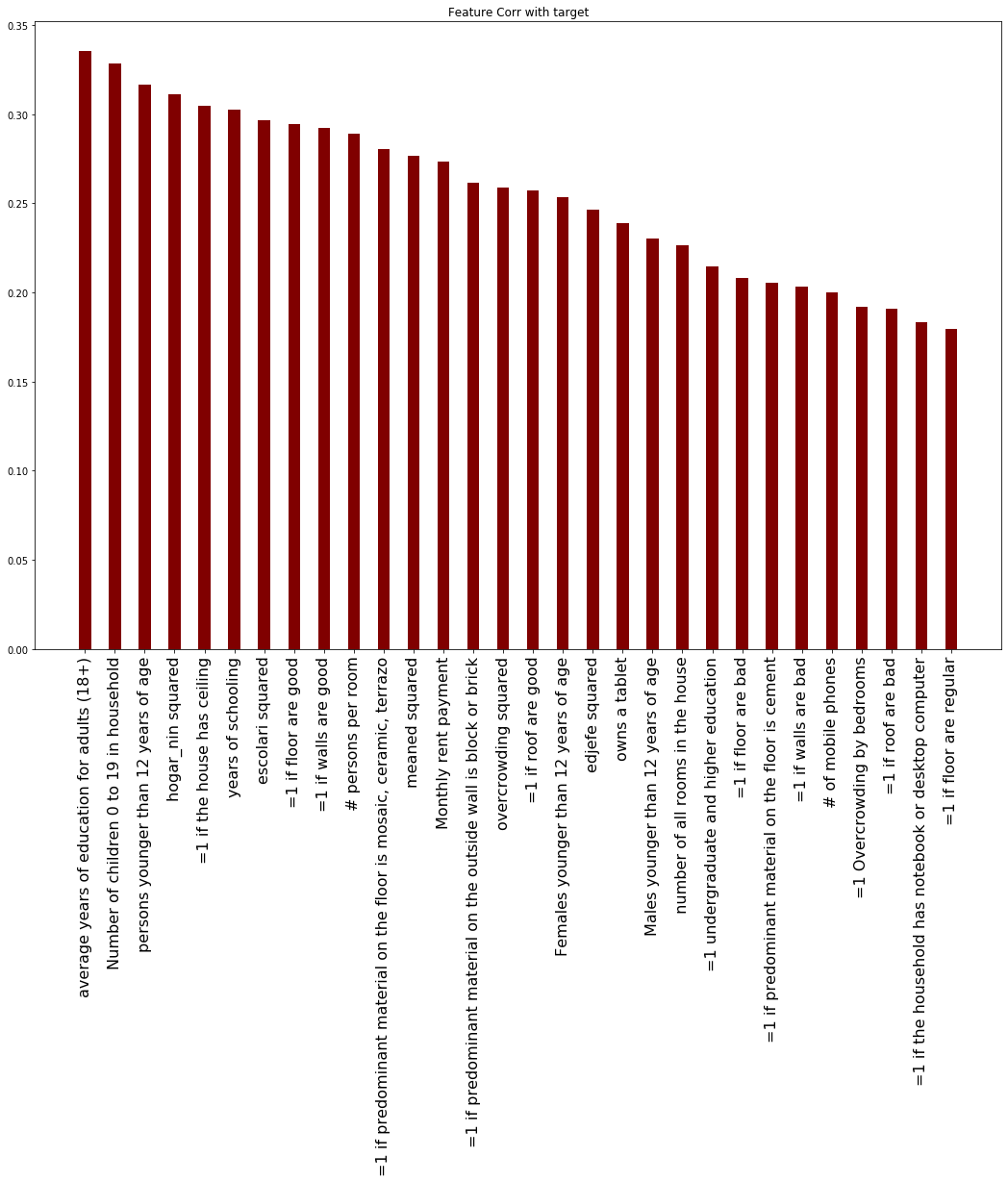


Figure : Top 30 features by absolute correlation with target variable

By taking the absolute correlation of every feature with the target variable, we are able to get the top 30 features by absolute correlation as seen in Fig 2.2.2. The feature with the highest correlation is “average years of education for adults (18+)” with a value between 0.30-0.35.

## 2.3. Filling up discrepancies in data

### 2.3.1. Filling up Nan values

As the dataset comes with missing values and a few discrepancies, they have to be rectified and filled as the machine learning models are unable to use those input rows with Nan values. In total, 5 columns had missing values and 3 columns had discrepancies. As the total number of rows with missing values accounted for 98.4% of all rows, it was not feasible to simply drop nan values as it would drastically reduce the number of datapoints available for training. The missing values have to be addressed in order to train the machine learning model and it is not feasible to arbitrarily assign a value as it will skew with model training later on. Hence, to tackle this problem, each feature with missing values is looked into to determine what is the most logical way to fill them.

#### 2.3.1.1. Monthly rent payment

**“**Monthly rent payment” is a household continuous attribute that represents the value of rent to be paid monthly and the column had a total of 71.8% of all rows having a missing value. Upon closer inspection, the reason for which there exists a large number of missing values is due to the fact that individuals who stay in a house that they own are not obliged to pay rent and thus have a missing value. Using the “=1 own and fully paid house” column, we can identify this group of individuals which account for 86.2% of all missing values. As these missing values represent individuals who do not pay rent, they are thus replaced with the value 0. The remaining values which are missing are unaccounted for and cannot be reasonably replaced with another value and are thus removed from the dataset.

#### 2.3.1.2. Number of tablets owned

“Number of tablets owned” is a household continuous attribute and the column had a total of 76.8% of all rows having a missing value. It can be seen that the existing values range from 1 tablet per household to 6 tablets owned in each household. Thus, it is implied that the missing values represent households with no tablets owned and the missing values are replaced with the value 0.

#### 2.3.1.3. Years behind in school

“Years behind in school” is an individual continuous attribute and the column had a total of 82.9% of all rows having a missing value. Upon deeper analysis, it can be seen that the individuals with missing values are either too young to have start school or too old such that they are no longer in school. For these individuals, their level of education can be represented through other columns. Hence the nan values for this column is filled with 0.

#### 2.3.1.4. Average years of education for adults (18+)/meaned squared

“average years of education for adults (18+)” is an individual continuous attribute while “meaned squared” is simply the squared value of the prior. Both columns share the same rows that have missing values and they account for 0.0005% of all existing rows. By comparing the values in this column with that of ‘years of schooling’ we can see that they are the same and hence the values can be taken from that of ‘years of schooling’. Due to the fact that this column can be represented by another column, they will be taken out later on as described in the report in section 2.4.3.

### 2.3.2. Replacing “Yes” and “No”

'Dependency rate', ‘'years of education of male head of household' and 'years of education of female head of household' are continuous household attributes which have discrepancies within the columns. Instead of numerical values, some rows have a string, “Yes” or “No”, value in the column of numerical continuous attributes.

#### 2.3.2.1. Dependency Rates

Looking at the features provided, it can be seen that there is a ‘dependency squared’ variable. This feature is simply the squared value of ‘Dependency rate” and it has no missing values. Thus, dependency rate can just be derived by taking the square rooted value from ‘dependency squared’ and used to fill the missing values.

#### 2.3.2.2. ‘'years of education of male/female head of household'

These features describe similar information regarding the years of education possessed by the head of household. The only difference between them is that one references female heads while the other references male. Hence, when the household head is male, 'years of education of male head of household' should be filled while

'years of education of female head of household' should be 0 and vice versa. The values in these columns can also be cross referenced with the column “years of schooling”. From this, we can see the following patterns emerging:

1. Both columns have ‘no’ values when the head of household had 0 years of schooling. Hence they will be replaced with 0.
2. In the case where one feature is 'years of education of male head of household' is ‘yes’ while 'years of education of female head of household' is “no’, all the head of households had 1 year of schooling. This is also true for the inverse. Hence these values will be replaced with 1 and 0 respectively.
3. For the remaining cases, one column will be represented by a number while the other will have a “no”. Hence, these cases will be replaced with 0.

Hence, the ‘yes’ and ‘no’ can be replaced according to which of the 3 scenarios listed above they fall into. This ensures that the values in the columns remain coherent to not affect the credibility and information provided in the column. To reduce dimensionality, these 2 columns are also condensed to a single column “Years of education of head of household” as the information pertaining to the gender of the household head is already fully captured by other columns.

### 2.3.3. Rectifying erroneous target labels

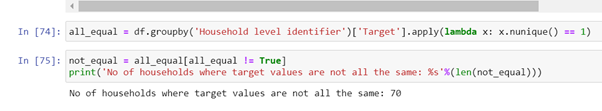


Figure 3 Checking for households with different target labels

We realised that in our dataset of >8000 individual entries, there are 70 data points which are invalid. These data points are identified when we group all entries by their Household Level Identifier and check if all members in the household have the same target label.

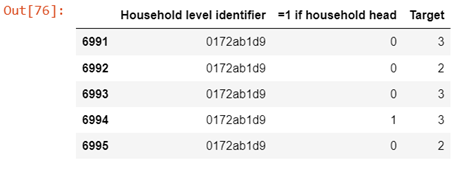


Figure 4 : Example of household with different target labels

The above is a specific instance, where all members in the household have the common identifier of ‘0172ab1d9’, yet they have differing Target labels of 2 or 3. In order to correct this, we will change the target labels of members in the household to follow that of the Household head. We have checked for the possibilities of households with differing target values and no identified household head and fortunately, there is none, so we do not have to account for this.

## 2.4. Feature Engineering

### 2.4.1 Binary to Ordinal

Upon examining the dataset in finer detail, we realise that we can condense several variables as they have a clear ordering and are mutually exclusive with the other variables in the same group. For instance, we can create an ordinal variable ‘electricity’ which can represent the information of cost of electricity where 0 represents low cost while 3 represents the highest cost. Thus, households that can afford electricity with cost 3 suggest higher affluence and likewise households with cost 0 suggest greater poverty. While it might still be possible for the model to learn the information despite leaving the variable in its nominal one hot encoded form, it will result in trickier parameter tuning, potentially longer training time, and possible loss of information. Hence, it makes sense to provide this information to the model by converting it to an ordinal variable during preprocessing and it allows us to reduce the dimensionality as well.

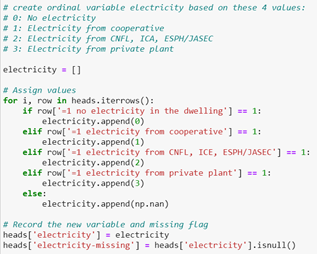


Figure 5 : Creating "electricity" ordinal variable

After which, the original columns can be dropped while we insert the new ‘electricity’ column. The above is just one example and we have created other variables including:

1. ‘walls’: representing the quality of the walls in household (good/regular/bad)
2. ‘roof’: representing the quality of roofs in household (good/regular/bad)
3. ‘floor’: representing the quality of flooring in household (good/regular/bad)

Like household variables, we can condense several variables which are mutually exclusive into 1 ordinal variable which give the same representation. In this case, the ‘highest education level’ variable can be created to replace other redundant variables like '=1 no level of education','=1 incomplete primary','=1 complete primary','=1 incomplete academic secondary level','=1 complete academic secondary level','=1 incomplete technical secondary level','=1 complete technical secondary level','=1 undergraduate and higher education','=1 postgraduate higher education'.

### 2.4.2. Aggregating Features

While the most intuitive method to train on the given dataset will be to consider each data entry on an individual level, we will also be experimenting on a different dataset that will be preprocessed furthermore following the steps taken earlier. Given that the assessment criteria and objective of this classification task will be to compute predictions on each household and not each individual, there is justification to consolidate data on a household level first before feeding the data into the classifiers. As elaborated earlier, the original dataset contains both individual level features, as well as household level features. The following steps will illustrate the methodical procedures to summarize the data of all individuals in each household, and also to perform an aggregation on individual variables. The initial dataset will be the preprocessed dataframe from the earlier segment.

#### 2.4.2.1. Creating of Per Capita Features

We can create some additional features that represent the per capita statistics of each household. These variables include:



Figure 6 : Creating 'schooling/age' feature

In this case, we can create a ‘schooling/age’ variable which appropriately captures the amount of schooling an individual has received relative to his age:

#### 2.4.2.2. Aggregating Individual level data for each Household

After performing the necessary steps to condense the individual level data earlier, we can group all the individual data points by their Household level identifier. The individual level data will then be transformed into household level data by taking the ‘min’, ‘max’, ‘sum’, ‘count’, ‘standard deviation’ and ‘range’. This is performed by using the ‘groupby’ and ‘agg’ functions available in pandas.

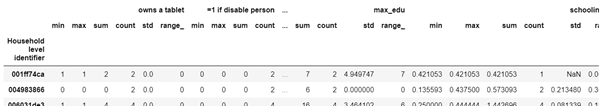


Figure 7: Aggregating individual features on household level

#### 2.4.2.3. Combining all Data

After performing the aggregation on the individual level data, we can combine the dataframe with the household level data. This can be achieved by merging the two dataframes on the Household Level Identifier. We perform a final round of feature selection by finding pairs of variables who have a correlation of more than 0.95. We will then drop 1 out of every pair of variables.

### 2.4.3. Removing redundancy and decreasing dimensionality

The curse of dimensionality refers to the potential issue of having too many features. When the dataset has a huge number of dimensions, it increases the risk of overfitting the model and thus resulting in the model to have poor out of sample performance. An example of this can be seen in the paper “Implications of the curse of dimensionality for supervised learning classifier systems: theoretical and empirical analyses”. The paper tested the impacts of increasing dimensionality by using the hyper-rectangular rule representation to depict the relationship among the dataset and found that when using a fixed number of training instances, its representation is exponentially decreasing as a function of dimensionality. Thus, severely affecting the ability of the representation to generalize the data (Debie & Kamran, 2019) Hence, we will first remove redundant features to reduce the dimensionality and prevent overfitting.

In deciding which features to drop, features that provide little or no additional information for the model were considered as they risked skewing the model negatively. The feature “=1 if rubbish disposal mainly by throwing in river, creek or sea” was removed due to the fact that it had a variance of 0 as every individual/household had a value of 1. As such, the feature added no predictive value and was thus removed.

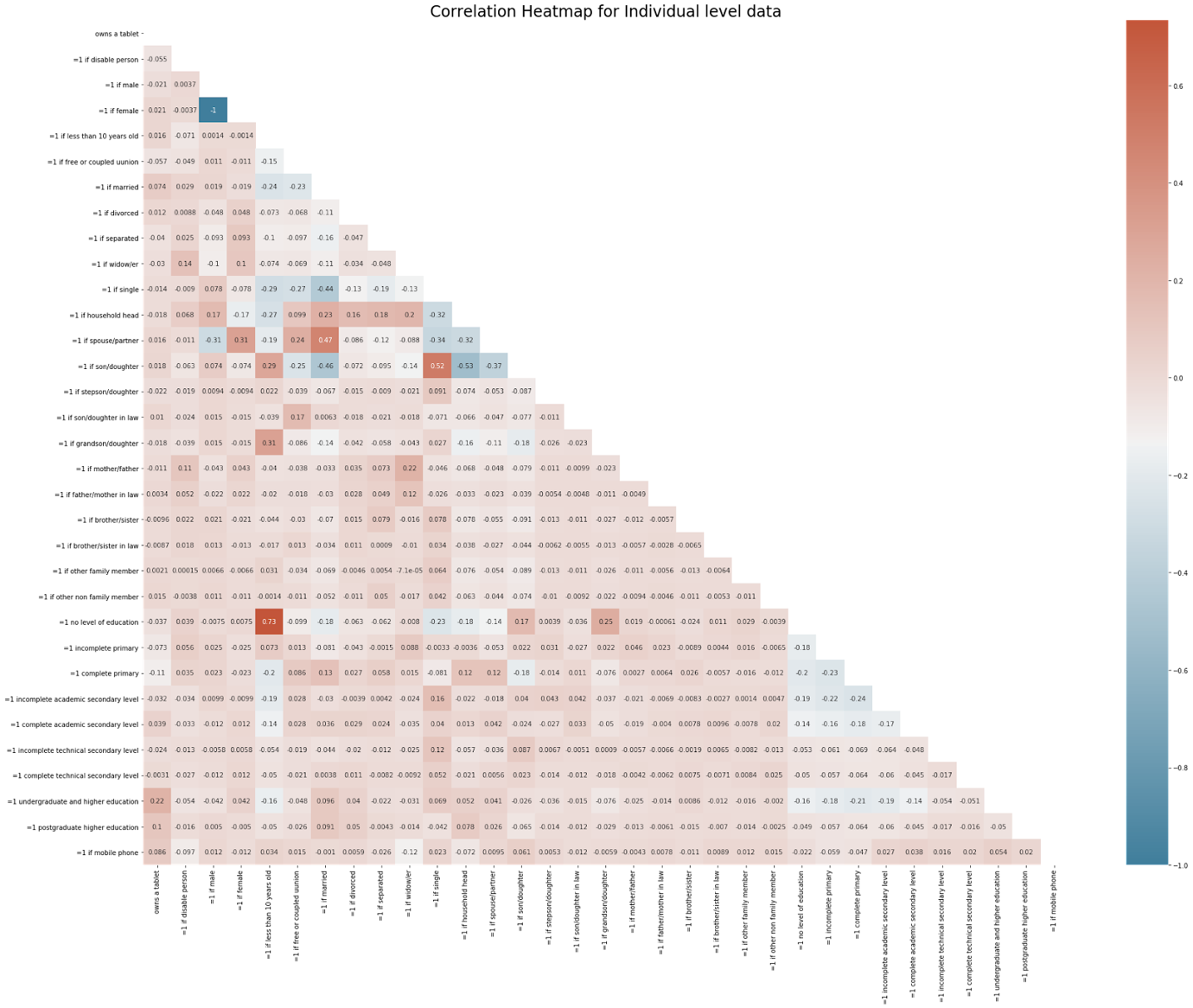


Figure 8 : Correlation heat map for individual level features

Features that have high levels of correlation as they do not provide additional information, but they increase model complexity and increase risk of overfitting. Features are tested for their correlation with one another , as per Fig 2.3.2.1, and features with correlation of >0.95 are removed as they generally can be represented by existing features. An example of such a feature is “=1 if female” and “=1 if male” which have a correlation of -1. Hence, one of the features is obsolete and was removed. By doing so we can remove the following columns:

|  |  |
| --- | --- |
| Columns removed | Reason for removal |
| 'Total persons in the household',  'size of the household',  'TamViv',  'household size' | Can be represented by household total variable instead |
|
|
|
| '=1 has toilet in the household' | Can be resented by “ =1 no toilet in the dwelling” |
| '=1 if predominant material on the outside wall is other',  '=1 if predominant material on the floor is other',  '=1 if no water provision',  '=1 no main source of energy used for cooking (no kitchen)',  '=1 if predominant material on the roof is other',  '=1 toilet connected to other system',  '=1 if rubbish disposal mainly other',  '=1 if single',  '=1 if other non family member',  '=1 other(assigned, borrowed)',  '=1 region Central',  '=1 zona urbana',  '=1 if female' | High Correlation with existing one-hot variables |
| 'escolari squared',  'age squared',  'hogar\_total squared',  'edjefe squared',  'hogar\_nin squared',  'overcrowding squared',  'dependency squared',  'meaned squared',  'Age squared' | Can be represented by their non-squared counterparts |
| ‘=1 if household head’ | Only household heads are used in final dataset hence variable has no variance |

Table 2: Columns to be removed



Figure 9 : Examples of highly correlated variables

From above, we see that there are several pairs of variables who are strongly correlated to each other. Since 'household size' has almost perfect correlation with 'size of household' and 'total individuals' in household, remove the latter 2. We also remove 'total persons in household' also as it has near perfect correlation of 1 with 'household size'.

## 2.5. Preparing dataset for training

Before we fit the training data into the models, it is very important that we normalise our data. We do this because by analysing the dataset, we see that the features are all of different ranges. A feature with a larger range will have a stronger influence on the model’s learning and thus, the classification results. However, this does not mean that it is a more significant predictor. Therefore, we will normalise the data to ensure that the machine learning models can learn best and churn out classification results of the highest possible accuracy.

Most of the columns in our dataset consist of binary values of 0 and 1. However, there are some with continuous values having different ranges. Therefore, we wrote a function to filter out these columns.

|  |
| --- |
| def selectUnnormalised(df):  normCol = df.columns[df.isin([0,1]).all()] #getting the binary columns  dfCols\_list = list(df.columns)  unnormCols = list(set(dfCols\_list)-set(normCol))    #remove target because we shouldn't normalise that  unnormCols.remove('Target')  return unnormCols |

Figure 10 : Function for getting columns with unnormalised data

Once we get these columns, we fit it into a function called *normalise* along with the *xTrain* and *xTest* data gathered from *train\_test\_split()*. In this function, we check for those columns with unnormalised data in both *xTrain* and *xTest* sets and normalise their values. The results returned are *xTrain* and *xTest* dataframes with all continuous values normalised.

|  |
| --- |
| def normalise(unnormCols, xTrain, xTest):  #normalise training data  toBeNorm\_train = xTrain[[i for i in unnormCols]]  ss = StandardScaler()  std\_scale = ss.fit(toBeNorm\_train)  xTrain\_norm = std\_scale.transform(toBeNorm\_train)    #covert numpy array to df  xTrain\_normCol = pd.DataFrame(xTrain\_norm, index = toBeNorm\_train.index, columns = toBeNorm\_train.columns)  xTrain.update(xTrain\_normCol)    #normalise test data using mean and SD of training set  toBeNorm\_test = xTest[[i for i in unnormCols]]  xTest\_norm = std\_scale.transform(toBeNorm\_test)  xTest\_normCol = pd.DataFrame(xTest\_norm, index = toBeNorm\_test.index, columns = toBeNorm\_test.columns)  xTest.update(xTest\_normCol)    return xTrain, xTest |

Figure 11 : Normalising Function

Once all these functions to preprocess the data are ready, we write a final function to return the ideal *xTrain, xTest, yTrain* and *yTest* data to train our models. In this function, we incorporate both the *selectUnnormalised* and *normalise* functions.

|  |
| --- |
| def prepData(df):  unnormal\_cols = selectUnnormalised(df)  x = df.iloc[:, :-1]  y = df.iloc[:, -1:]  xtrain, xtest, ytrain, ytest = tts(x,y,test\_size = 0.3, random\_state = 42)  xTrain, xTest = normalise(unnormal\_cols, xtrain, xtest)  yTrain, yTest = ytrain.values.ravel(), ytest.values.ravel()  return xTrain.values, xTest.values, yTrain, yTest |

Figure 12 : Function to prepare the training and test data

# 3. Experiments & Results

## 3.1. Models

For our classification, we will be exploring 4 types of models as follows.

### 3.1.1 Bagging classifier

A bagging classifier is an ensemble meta-estimator consisting of multiple versions of a base estimator. These versions of the base estimator are trained with variant forms of training data using a technique known as bagging sampling. Bagging sampling essentially collects samples of the training data using replacement. This results in a training set made up of duplicate or unique collections of data. The final predictor, the bagging classifier, combines the predictions made by each version of the base estimator by voting (classification). This way, it reduces overfitting. However, it could lead to an increase in bias.

We chose to use this model among many others because bagging allows many weak learners to combine their efforts to outperform a single, strong learner. Bagging also reduces the variance thus preventing the model from overfitting during training.

### 3.1.2. Random Forest classifier

Random Forest is an ensemble of many decision trees. It works through a concept known as the wisdom of crowds. This concept is illustrated by how the different decision trees churn out a class prediction and the class with the highest vote becomes the model’s prediction.

Random Forest incorporates additional randomness to the model while growing the decision trees. Rather than using the most important feature to split a node, it uses the best feature among a random subset of features. This gives the model a larger diversity and makes it better.

A few reasons why we decided to include Random Forests are that it can handle large data with plenty of variables running into the hundreds, as with our dataset, and because it can autonomically balance data sets when a class is more infrequent than other classes in the data. This is such a case with our dataset where the 4 classes were not uniformly distributed across the dataset. Lastly, among all the available classification methods, Random Forest usually provides the highest accuracy.

### 3.1.3. Extra Trees classifier

The Extra Trees classifier is very similar to the Random Forest classifier. The difference is that in Extra Trees, the features and splits are randomly chosen and therefore, it is less computationally expensive than Random Forest.

In Extra Trees, each decision tree is built following certain criteria. Firstly, all the data in the training set is used to build each tree. Secondly, to form a node, the best split is made through searching in a subset of randomly chosen features that is the size of the square root of the total number of features. Lastly, the maximum depth of each decision tree is one.

Extra Trees, though remarkably similar to Random Forest, does have a tad bit of difference compared to it. While Random Forest chooses the optimum split while Extra Trees does this randomly. However, once the split is done, the two choose the best points between all the subset of features. Since, the Extra Trees splits randomly, it is much faster and less computationally expensive.

### 3.1.4. LightGBM classifier

LightGBM is a gradient boosting framework that uses tree-based learning algorithms. It also generally performs better than the XGBOOST gradient boosting framework. This is because the LightGBM framework splits the tree leaf-wise with the best fit. The leaf-wise algorithm can reduce more loss than level-wise algorithm, thus providing better accuracy than existing boosting algorithms. It is designed to be distributed and efficient with the following advantages: Faster training speed and higher efficiency, lower memory usage, better accuracy, support of parallel, distributed, and GPU learning, capable of handling large-scale data.

LGBM was employed for a variety of good reasons. Firstly, it produces much more complex trees by following leaf wise split approach rather than a level wise approach. This results in it is achieving higher accuracy. Secondly, it is equally competent in handling a large dataset with a significant reduction in training time compared to XGBOOST. Lastly, it trains extremely fast because it uses a histogram-based algorithm that buckets the continuous feature values into discrete bins.

## Training

### Bagging classifier

To achieve the best model possible, we will have to vary three important parameters in the classifier. They are:

1. *n\_estimators:* The number of base estimators in the ensemble. In our case, our base estimator is a decision tree classifier.
2. *max\_features:* The number of features to draw from training data to train each base estimator
3. *max\_samples:* The number of samples to draw from training data to train each base estimator.

|  |
| --- |
| num\_estimators = [500,800,1000,1250,1650] max\_feature = [20,50,75,100,129] max\_sample = [100,200,225,275,300] |

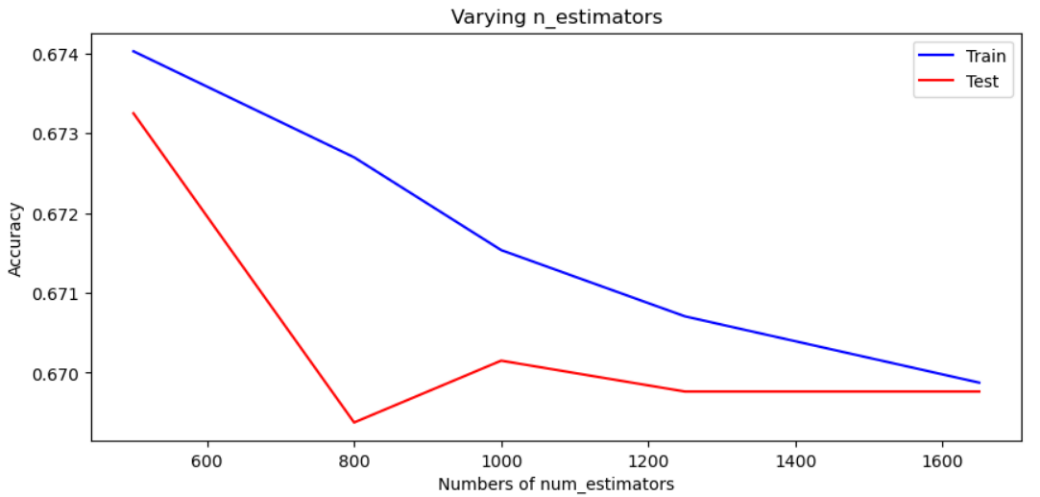
Figure 13 : Various arguments for each parameter

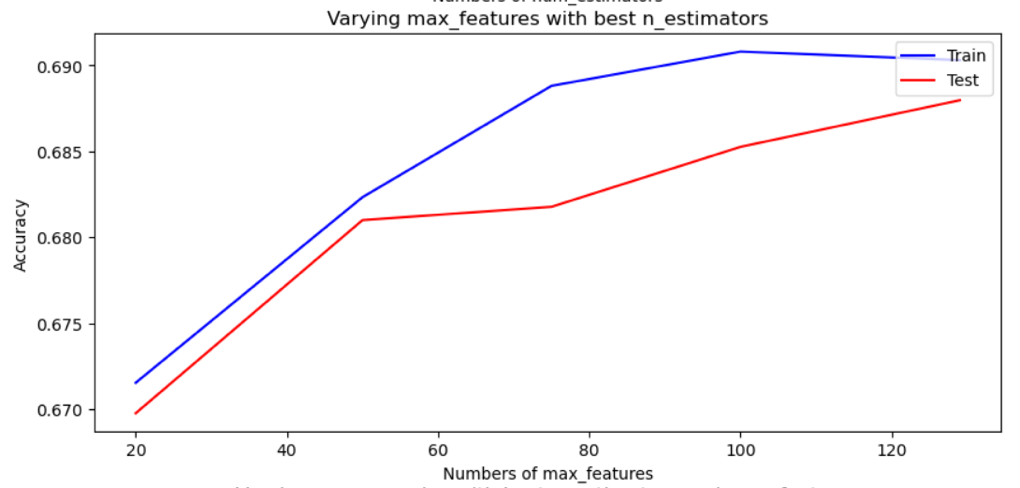
To achieve the best model, we carried out the following steps:

1. *Vary the n\_estimators values using those in num\_estimators while setting the max\_features and max\_samples at their lowest values of 20 and 100, respectively.*
2. *Run the model through a loop and collect the test accuracies achieved.*
3. *Using the n\_estimators value that gave us the highest test accuracy, train the model again by varying max\_features.*

Repeat these steps to get the best arguments for *max\_features* and *max\_samples*. The code for this is illustrated in **Fig 1** in Annex A.

As we continue to use the best argument for each parameter, we notice that the accuracy of classification keeps increasing and this is illustrated by the graphs shown below.





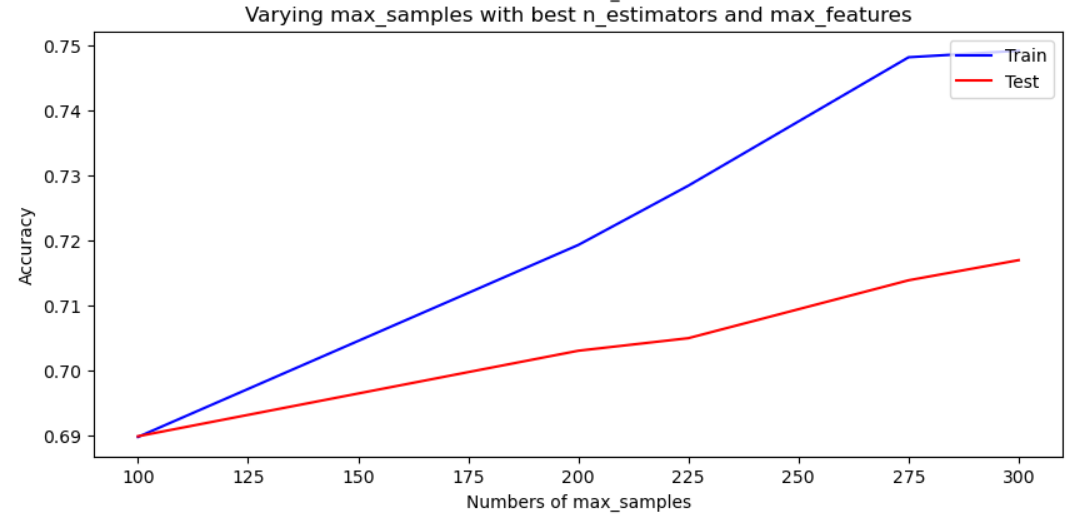


Figure 14 : Hyperparameter tuning to obtain optimal accuracy (bagging classifier)

We conclude that the best bagging classifier has the following parameters:

1. n\_estimators: 500
2. max\_features: 129
3. max\_samples: 300

### Random Forest classifier

In Random Forest, there are 3 parameters to be concerned with:

1. *n\_estimators:* The number of trees in the Forest
2. *max\_depth:* The maximum depth of the tree
3. *class\_weight:* Weights associated with classes

While assessing the distribution of the classes in the dataset, we noticed that the classes are unequally distributed. Therefore, we shall set the *class\_weight* as ‘balanced’ and this will automatically adjust weights inversely proportional to class frequencies in the input data.

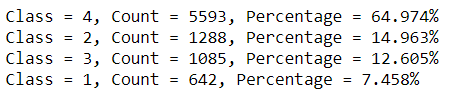
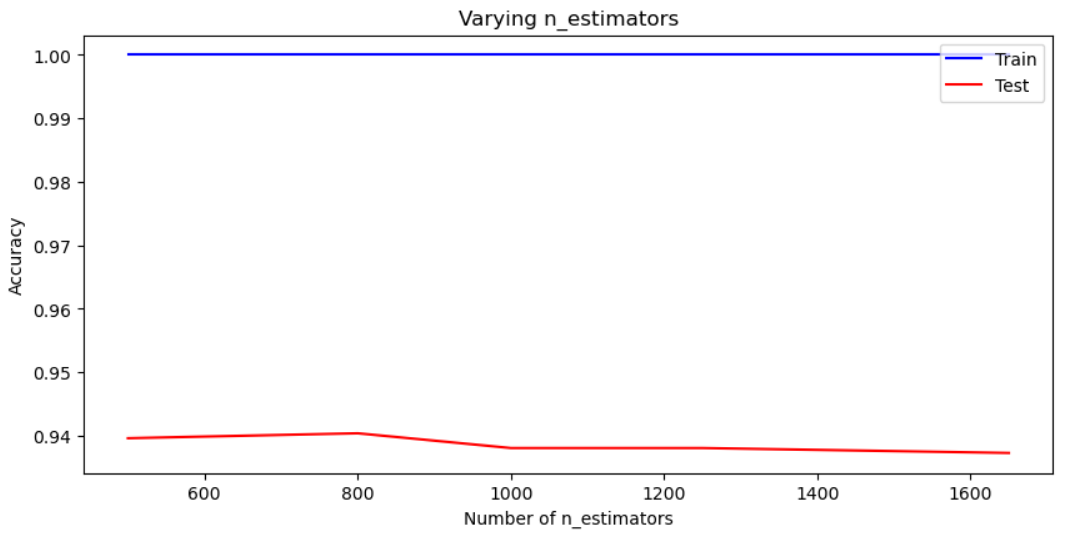


Figure 15: Distribution of classes

To achieve the best model, we follow the same procedure as the one in the bagging classifier where we varied the values for the parameters. The steps involved were essentially finding the best argument for *n\_estimators* followed *max\_depth.* The code to retrieve these values can be seen in **Fig 2** of Annex A.



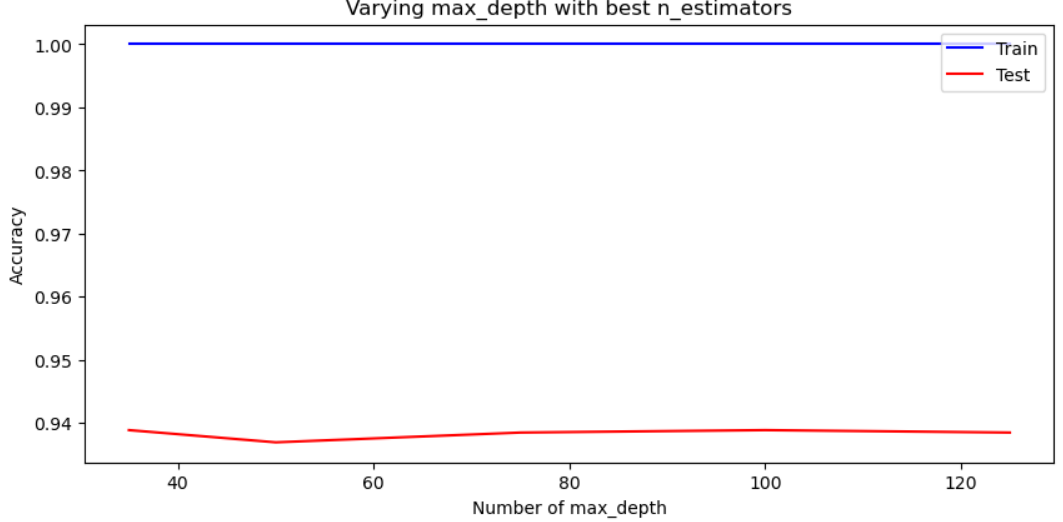


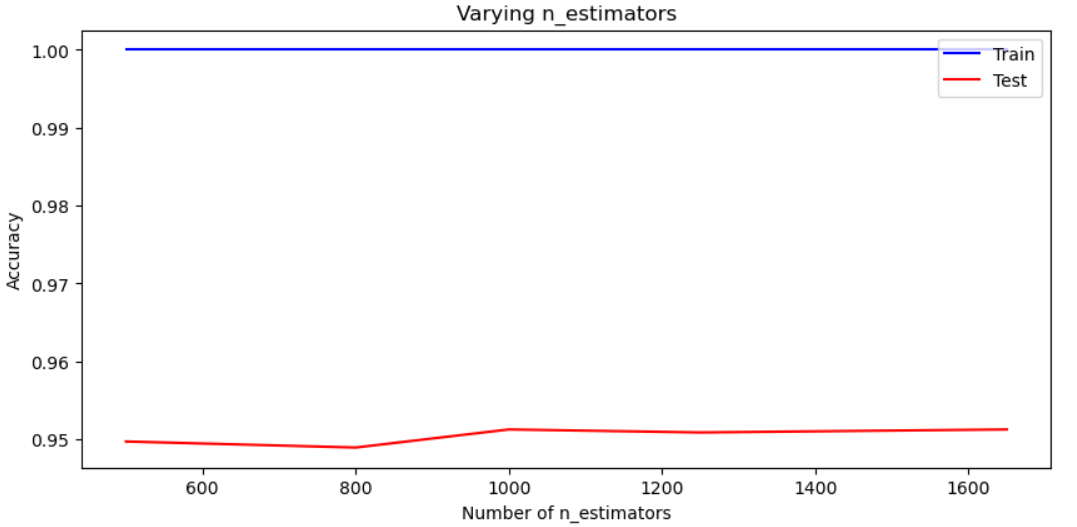
Figure 16 :HYPERPARAMETER TUNING TO OBTAIN OPTIMAL ACCURACY (random Forest)

We conclude that the best Random Forest classifier has the following parameters:

1. n\_estimators: 800
2. max\_depth: 35

### Extra Trees classifier

With Extra Trees classifier, the parameters are the same as those of Random Forest classifier. The different values for each parameter are also set to the same. Therefore, we shall dive right into assessing the performance of the model in the bid to seek the best values for *n\_estimators* and *max\_depth.* The code to attain these values are the same as that in **Fig 2** of Annex A.



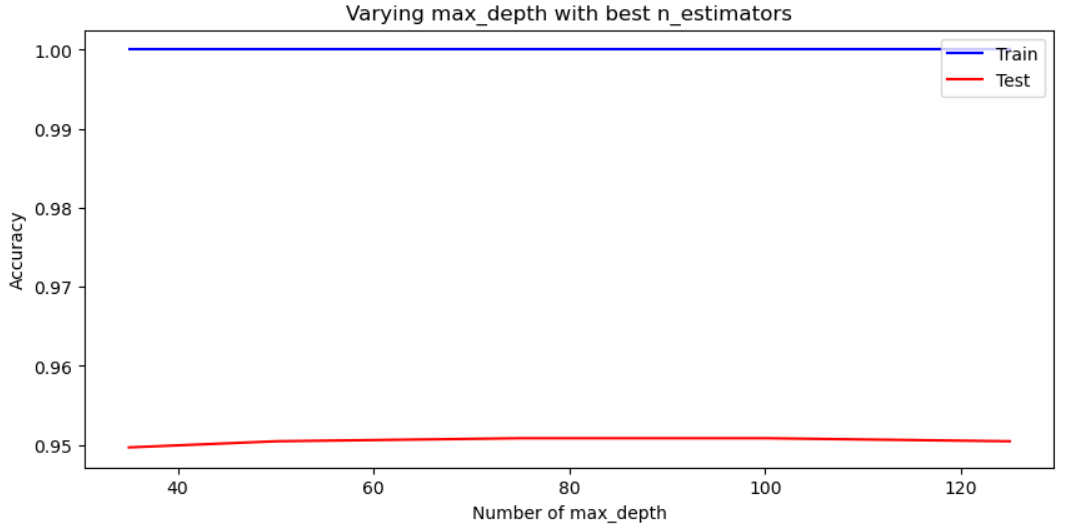


Figure 17 : Hyperparameter tuning to obtain optimal accuracy (extra classifier)

We conclude that the best Random Forest classifier has the following parameters:

1. n\_estimators: 1000
2. max\_depth: 75

### LightGBM classifier

To achieve the best model possible, we will have to vary two important parameters in the classifier. They are:

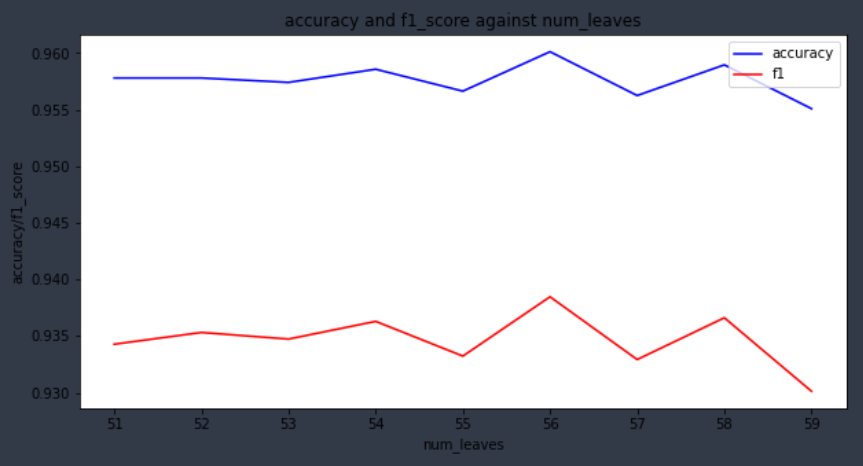
*1. num\_leaves: The number of base estimators in the ensemble. In our case, our base estimator is a decision tree classifier.*

*2. learning\_rate: The number of samples to draw from training data to train each base estimator.*

The num\_estimators will be constant as early stopping rounds = 400 is in place.

|  |
| --- |
| num\_leaves = [51,52,53,54,55,56,57,58,59]  learning\_rate = [0.01,0.02,0.03,0.04,0.05,0.06,0.07,0.08,0.09,0.1] |

Figure 18 : Various arguments for each parameter



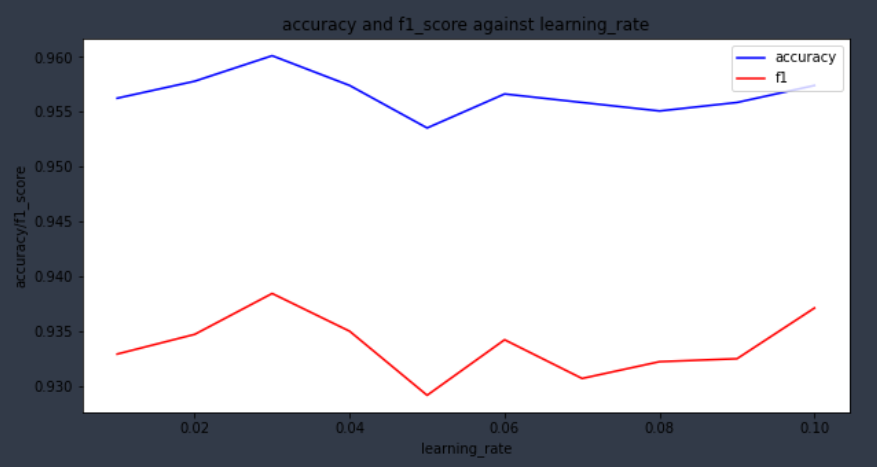


Figure 19 : Hyperparameter tuning to obtain optimal accuracy (LGBM)

The parameters for the model was set as following:

1. *n\_estimators* = 5000 (early stopping rounds = 400)
2. *max\_depth* = -1 (default)
3. num\_leaves = 56
4. learning\_rate = 0.03
5. *metric* = ‘multi\_logloss’

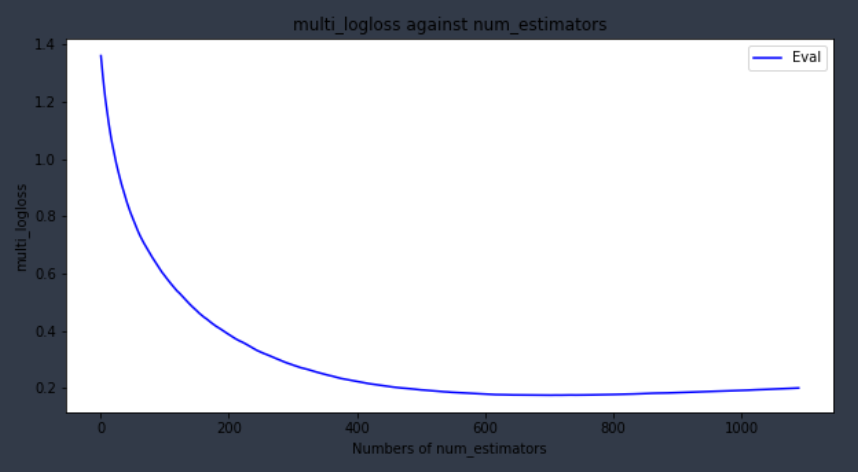


Figure 20 : Multi-logloss against num\_estimators

The best iteration was 689 with lowest multi\_logloss = 0.17455.

## Predicting/Classifying

### 3.3.1. Bagging classifier

Using the optimal values of the parameters, we predicted the labels of the test set as follows.

|  |
| --- |
| bag = bc(n\_estimators = 500, max\_features = 129, max\_samples = 300) bag.fit(xTrain, yTrain) y\_pred = bag.predict(xTest) print("Classification accuracy: {:.2f}".format(bag.score(xTest, yTest))) print("F1 score: {:.2f}".format(f1\_score(yTest, y\_pred, average = 'macro'))) |

Figure 21 : Optimal parameters for bagging classifier



Figure 22 : Prediction result of bagging classifier

Despite a decent classification accuracy of 72%, the f1 score for this model is pretty low at 0.43. Therefore, this model may not be the best option for our classification task.

### Random Forest classifier

Using the optimal values for the parameters, we predicted the labels of the test set as follows.

|  |
| --- |
| rff = rf(n\_estimators = 800, max\_depth = 35, class\_weight = 'balanced') rff.fit(xTrain, yTrain) y\_pred = rff.predict(xTest) print("Classification accuracy: {:.2f}".format(rff.score(xTest, yTest))) print("F1 score: {:.2f}".format(f1\_score(yTest, y\_pred, average = 'macro'))) |

Figure 23: Optimal parameters for random forest classifier



Figure 24 : Prediction result for random forest

The Random Forest classifier clearly trumps bagging classifier as it has a very high score for both classification accuracy (94%) and f1 score (0.91). Therefore, the Random Forest classifier is a model we can seriously consider for accurate classification of data in the test dataset.

### Extra Trees classifier

Using the optimal values for the parameters, we predicted the labels of the test set as follows.

|  |
| --- |
| ett = et(n\_estimators =1000, max\_depth = 75, class\_weight = 'balanced') ett.fit(xTrain, yTrain) y\_pred = ett.predict(xTest) print("Classification accuracy: {:.2f}".format(ett.score(xTest, yTest))) print("F1 score: {:.2f}".format(f1\_score(yTest, y\_pred, average = 'macro'))) |

Figure 25 : Optimal parameters for extra trees classifier



Figure 26 : Prediction results for extra trees

The Extra Trees classifier performed slightly better than Random Forest with a classification accuracy of 95% and an f1 score of 0.93. Therefore, we can also use this classifier to predict the classes accurately in the test dataset.

### LightGBM classifier

Using the optimal values for the parameters, we predicted the labels of the test set as follows.

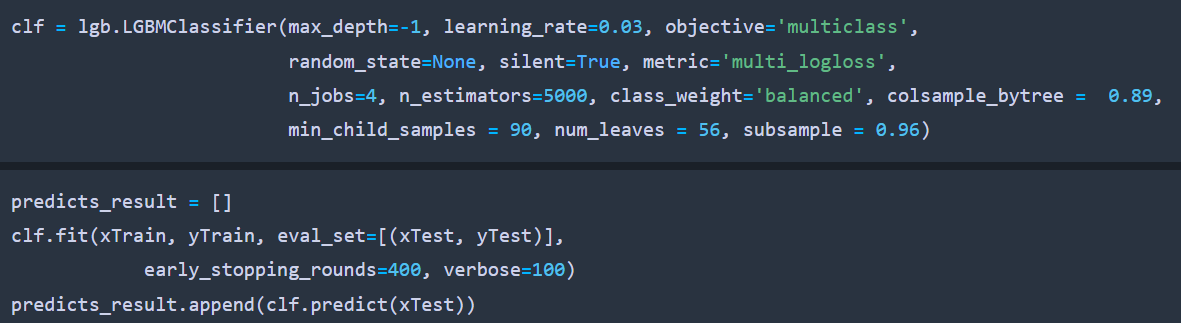


Figure 27 : Optimal parameters for lgbm

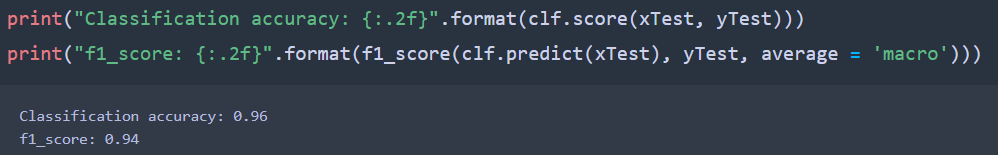


Figure 28 : Prediction results for lgbm

The LightGBM classifier performed similarly to the extra tree classifier with a classification accuracy of 96% and an f1 score of 0.94. Hence, we can also use this classifier to predict the classes accurately in the test dataset.

# Discussion

## 4.1. Submission

The LightGBM was chosen as our final classifier as it showed the highest accuracy and f1 score out of all the classifiers. The final results were submitted to the Kaggle competition and the evaluation score and ranking were as follows.

Text

Description automatically generated

Figure 29 : Highest Attained Score (LGBM, non-aggregated dataset)

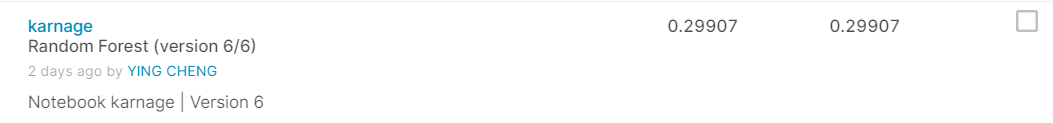


Figure 30 : Score with Random Forest classifier (non-aggregated dataset)

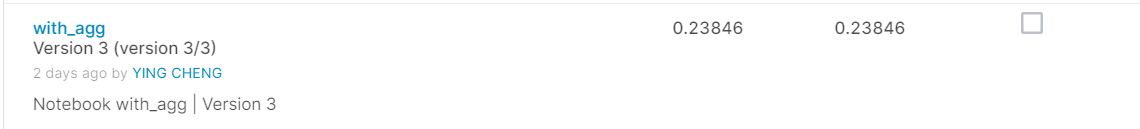


Figure 31 : Score with LGBM classifier (aggregated dataset)

The highest evaluation score returned was 0.40802 and our result will place us at 231/616 position on the leaderboard (top 37.5%).

## Results using aggregated dataset

Comparing the results in the previous section (figure 22 - 28), we can see that the accuracy scores achieved using the dataset with aggregated statistics across households actually are marginally worse than the scores using the default dataset (individual level). This is contrary to our expectations. These are some of our analysis and explanations as to why the aggregated dataset performed poorly:

* The dataset comprising of individual level data contained a lot more data samples and much more robust compared to the aggregated dataset (about 4 times more training samples). More training data will add more diversity to the model training and decreases the generalization error.
* The creation of new household level statistical features like ‘min’, ‘max’, ‘sum’, ‘count’, ‘standard deviation’ and ‘range’ may not have improved the model performance. In fact, these extra features could have added more irrelevant noise to the model training, which in turned reduced the model accuracy as the model may not have correlated well with the other original features that the dataset contained. Another explanation is that overfitting might have occurred with the addition of the new created features.
  + It is worth noting that number of features that are household aggregated essentially multiplied by 5 (5 statistical methods used).

## 4.3. Learning Points

In this project, we learnt to carry out data exploration and data preprocessing on large-scale real dataset, which helped us to understand the dataset better and improve the model accuracy by improving the quality of the dataset. Also, we explored various classifiers: Bagging classifier, Random Forest Classifier, Extra Trees Classifier and LightGBM Classifier, and compared their prediction accuracy to obtain the best performing classifier. In addition, we also carried out model training and model evaluation on each classifier to identify the optimal parameter to be used.

Ultimately, we conducted empirical testing using 4 models representing 4 different approaches to the problem:

1. Bagging Classifier (Ensemble Learning using bagging to train base classifiers)
2. Random Forest (Ensemble Learning using a subset of optimal features to train base classifiers)
3. Extra Trees (Ensemble Learning using a subset of random features to train base classifiers)
4. LightGBM (Ensemble Learning using Gradient Boosting)

The decision to use ensemble learning was mainly due to the fact that ensemble learning methods generally performs better than their individual counterparts (Dietterich, 2002). Furthermore, using an individual classifier (Decision Tree) increases the risk of a model with higher variance as decision trees make a cascade of choices where one incorrect choice would then result in an impact on the subsequent decisions. Hence, our decision to use mainly ensemble learning techniques was justified as model performance (as measured by accuracy) remained strong and model variance was low as seen by the strong and consistent performance across both train and test sets.

In comparison between our various bagging and boosting approaches, it can be seen that boosting (LightGBM) outperformed bagging in our test scenario. LightGBM performed strongly in both accuracy and computational speed making it suitable to be used in our context where we have sufficient data entries to enjoy the benefits of a faster while ensuring that the model does not overfit. In Bagging, each model receives an equal weight and it merges the same type of predictions thereby decreasing variance. In contrast, in boosting, models are weighted based on their performance where models that can predict “hard to predict” samples have higher weightage. This in turns decreases bias.

# 5.Bibliography

Dietterich, T. G. (2002). Ensemble Learning. *The MIT Press.*

Debie, E. S., & Kamran, S. (2019). Implications of the curse of dimensionality for supervised learning classifier systems: theoretical and empirical analyses. *Mathematics, Computer Science Pattern Analysis and Applications.*

# Appendix

|  |
| --- |
| def trainBG(xtrain, xtest, ytrain, ytest):  #get list of accuracies  accEst\_train = []  accEst\_test = []  accFeat\_train = []  accFeat\_test = []  accSam\_train = []  accSam\_test = []    #to vary the parameters  num\_estimators = [500,800,1000,1250,1650]  max\_feature = [20,50,75,100,129]  max\_sample = [100,200,225,275,300]        #train with varying num\_estimators  print("Training with varying num\_estimators...")  print("Completed run: \t")  for i in range(0,len(num\_estimators)):    bc\_clf = bc(n\_estimators = num\_estimators[i],  max\_features = max\_feature[0], max\_samples = max\_sample[0], n\_jobs = 5)  bc\_clf.fit(xtrain, ytrain)  accEst\_train.append(bc\_clf.score(xtrain, ytrain))  accEst\_test.append(bc\_clf.score(xtest,ytest))  print(str(i+1), end = "\t")    best\_est = num\_estimators[np.argmax(accEst\_test)]    #train with varying max\_features  print("\nTraining with varying max\_features...")  print("Completed run: \t")  for i in range(0,len(max\_feature)):    bc\_clf = bc(n\_estimators = best\_est,  max\_features = max\_feature[i], max\_samples = max\_sample[0], n\_jobs = 5)  bc\_clf.fit(xtrain, ytrain)  accFeat\_train.append(bc\_clf.score(xtrain, ytrain))  accFeat\_test.append(bc\_clf.score(xtest,ytest))  print(str(i+1), end = "\t")    best\_numFeat = max\_feature[np.argmax(accFeat\_test)]    #train with varying max\_sample  print("\nTraining with varying max\_sample...")  print("Completed run: \t")  for i in range(0,len(max\_sample)):    bc\_clf = bc(n\_estimators = best\_est,  max\_features = best\_numFeat, max\_samples = max\_sample[i], n\_jobs = 5)  bc\_clf.fit(xtrain, ytrain)  accSam\_train.append(bc\_clf.score(xtrain, ytrain))  accSam\_test.append(bc\_clf.score(xtest,ytest))  print(str(i+1), end = "\t")    best\_numSam = max\_sample[np.argmax(accSam\_test)]    print("\n\nBest parameters for...\nnum\_estimators: %d\tmax\_features: %d\tmax\_samples: %d" %(best\_est, best\_numFeat, best\_numSam))    return accEst\_train, accEst\_test, accFeat\_train, accFeat\_test, accSam\_train, accSam\_test |

**Fig 1**

|  |
| --- |
| def trainRF\_or\_ET(modelName, xtrain, xtest, ytrain, ytest):  accEst\_train = []  accEst\_test = []  accDepth\_train = []  accDepth\_test = []  #to vary the parameters  num\_estimators = [500,800,1000,1250,1650]  max\_deep = [35,50,75,100,125]    #train for varying num\_estimators  print("Training with varying num\_estimators...")  print("Completed run: \t")  for i in range(0,len(num\_estimators)):  if(modelName == "Random Forest"):  model = rf(n\_estimators = num\_estimators[i],  max\_depth = max\_deep[0], class\_weight ='balanced')  elif(modelName == "extra tree"):  model = et(n\_estimators = num\_estimators[i],  max\_depth = max\_deep[0], class\_weight = 'balanced')  model.fit(xtrain,ytrain)  accEst\_train.append(model.score(xtrain, ytrain))  accEst\_test.append(model.score(xtest, ytest))  print(str(i+1), end = "\t")  best\_est = num\_estimators[np.argmax(accEst\_test)]  #train for varying max\_depth  print("\nTraining with varying max\_depth...")  print("Completed run: \t")  for i in range(0,len(max\_deep)):  if(modelName == "Random Forest"):  model = rf(n\_estimators = best\_est,  max\_depth = max\_deep[i], class\_weight ='balanced')  elif(modelName == "extra tree"):  model = et(n\_estimators = best\_est,  max\_depth = max\_deep[i], class\_weight = 'balanced')  model.fit(xtrain,ytrain)  accDepth\_train.append(model.score(xtrain, ytrain))  accDepth\_test.append(model.score(xtest, ytest))  print(str(i+1), end = "\t")  best\_depth = max\_deep[np.argmax(accDepth\_test)]  print("\n\nBest parameters for...\nnum\_estimators: %d\tmax\_depth: %d" %(best\_est, best\_depth)  return accEst\_train, accEst\_test, accDepth\_train, accDepth\_test |

**Fig 2**