Machine Learning Engineer Nano Degree – Capstone Project

Household Poverty Level Prediction using Machine Learning Classification Algorithms

Definition

Project Overview

This project's high level goal is for fighting poverty.

While the specific data set given in this project was introduced by the Inter-American Development Bank (IADB) for Costa Rica, I strongly believe that many of the techniques used here, can be re-used for similar analysis for other populations in other countries. Poverty is a problem that concerns many governments in many places around the world, and the need to analyze in advance which families are more likely to require the aid of social welfare is something that can help a lot to improve the quality of life for many people.

From Wikipedia: "**Poverty** is the scarcity or the lack of a certain (variant) amount of material possessions or money. Poverty is a multifaceted concept, which may include social, economic, and political elements. Absolute poverty, extreme poverty, or *destitution* refers to the complete lack of the means necessary to meet basic personal needs such as food, clothing and shelter".

Poverty has been, and still is, a major concern for many countries around the world, affecting the lives of billions of people. According to www.dosomething.org, "Nearly 1/2 of the world's population — more than 3 billion people — live on less than $2.50 a day. More than 1.3 billion live in extreme poverty — less than $1.25 a day". And even the World Bank (www.worldbank.org) admits that "Despite the progress made in reducing poverty, the number of people living in extreme poverty globally remains unacceptably high. And given global growth forecasts, poverty reduction may not be fast enough to reach the target of ending extreme poverty by 2030".

Thus, it is still crucial for many countries to find & aid those families in need. One way for social welfare to find the population in need is by the use of the **PMT** (**P**roxy **M**eans **T**est) https://olc.worldbank.org/sites/default/files/1.pdf:

The main idea behind PMT is that in many situations we cannot have clear and accurate income reports on these populations. Even the household members themselves cannot report on their income and\or spending. And so, we have to use other, alternative metrics, in order to evaluate their economy status:

1. The type of wall that is surrounding their house (brick or clay)
2. Does the family have livestock ownership (e.g. cattle)
3. Etc…

Naturally, there can be relatively a very large number of parameters that can make up a PMT for making the evaluation of a family's welfare, and doing such an evaluation manually can be a very long and cumbersome work for social welfare organizations.

This is where machine learning can come into play.

About the Dataset

the dataset to be used for this project is taken from Kaggle recent competition on this subject, contributed by the Inter-American Development Bank. Link to the competition in Kaggle: https://www.kaggle.com/c/costa-rican-household-poverty-prediction

The specific data set was given for Costa Rica, though the trained models can probably be re-used for other Latin American countries, and probably for other countries in the world.

The data itself consists of a CSV file containing ~ 10,000 labeled records. Each record (line) is composed of 142 (!) features and the target variable which is a whole number in the range 1-4 (1 = Extreme poverty, 2 = Moderate poverty, 3 = vulnerable household, 4 = non vulnerable household).

Problem Statement

The challenge of this project is to train and find the best classifier for prediction of poverty level of households according to the given PMT input data. Note that because this is a **supervised multi-class classification** problem (there are 4 target classifications), we can take into account several algorithms for solving it.

In this project I will evaluate the following classifiers that fit this problem, testing each one with various set of hyper parameters, and compare the results between them, denoting the best classifier found:

Gaussian Naïve Bayes
Decision Trees
AdaBoost
K-Nearest Neighbors
SVC
Random Forrest
XGBoost
LightGBM

One of the ***major*** challenges of applying any of the models above, is that the dataset itself requires ***extensive feature analysis &*** **preprocessing**:

- There are relatively many features in the initial data
- The goal of the project is to predict the poverty level **per household**. Some of the features are **per household** while others are per **individual**. As part of the feature preprocessing we need to take that into account, as well as combining (**via aggregations**) all individual features to be household features, solving any incompatibilities found along the way.

- The dataset at hand contained several “duplicate” features, data skewed features, and its target classification values **are very unbalanced**.

After finding the best classifiers, I will try to use PCA in order to reduce the feature complexity, thus having the best classifier(s) train on a less complex problem and see how this affects the final results of those classifiers.

As a benchmark (to compare to the ML classifiers), I will use a ***naïve predictor***, just to proof that classification model is possible and compare the algorithms in my solution statement to the naïve predictor. The naïve predictor will simply make a prediction of value ***4 (Non-vulnerable)*** regardless of the feature vector of a given record.

My expectation is that while the naïve predictor will give low accuracy compared to the other classifiers, as well low F1 score results (See more information on that in the “Metrics” section).
I will compare both the accuracy as well as the balanced accuracy scores of the naïve predictor to the other algorithms.

Metrics

I will use the ***weighted*** **F1 score** and **balanced accuracy score** as evaluation metrics for the classifiers being tested, as the input data is greatly imbalanced (there are very few extreme poverty records compared to the other record labels), so using the regular F1 score here is not good enough. I will also denote the **confusion matrix**, as well as the **running time** for each classifier.

Accuracy Score
Accuracy score is the ratio between the number of our correct predictions to the total prediction count, so it can be defined as: ***(TP + TN) / (TP + TN + FP + FN)*** (TP = True Positive, TN = True Negative, FP = False Positive, FN = False Negative)

F1 Score
F1 score is the weighted average of ***Precision*** and ***Recall***.

Precision is the ratio of correctly predicting a specific label out of all the times we predicted that label, so it can be defined as: ***TP / TP + FP*** (TP = True Positive, FP = False Positive)

Recall is the ratio of correctly predicting a specific label out of all the times this label exists in the observations, so it can be defined as: ***TP / TP + FN*** (TP = True Positive, FN = False Negative)

And since F1 Score is the weighted average between them, it is defined as:
F1 score = **2 * (Precision * Recall) / (Precision + Recall)**

Note that because this is a ***multiclass*** classification problem, the final F1 score is the **average** of F1 scores per each class.

Why using F1 score and Accuracy score is not good enough for this problem?

Because this problem is **highly un-balanced** (you will see it in the initial exploratory visualizations), using

the regular equations denoted above may yield incorrect results, since the predictions should take into account also the **weight** of each class (e.g. there are much more observations of target class = 4 compared to the others).

So I will use instead the **weighted** F1 score which weights the calculated average by the ratio of number of true positives for each class (label).

For the same reason I will use the **balanced** accuracy score, which weights the calculated average by the inverse prevalence of the true positives. If the classifier performs equally well on either class, this term reduces to the conventional accuracy (i.e., the number of correct predictions divided by the total number of predictions). In contrast, if the conventional accuracy is above chance only because the classifier takes advantage of an imbalanced test set, then the balanced accuracy, as appropriate, will drop to **1 / (Number of classes)**. We will see exactly that in the naïve predictor.

Analysis

Data Exploration

The dataset which was used is a single CSV file from Kaggle competition which can be downloaded from their site:
https://www.kaggle.com/c/costa-rican-household-poverty-prediction

When loading and looking at sample observations of the data set, we get the following results:

```
Total rows*columns of the dataset:  (9557, 143)
```

	Id	v2a1	hacdor	rooms	hacapo	v14a	refrig	v18q	v18q1	r4h1	r4h2	r4h3	r4m1	r4m2	r4m3	r4t1	r4t2	r4t3	tamhog	tamviv	escola
0	ID_279628684	190000.0	0	3	0	1	1	0	NaN	0	1	1	0	0	0	0	1	1	1	1	1
1	ID_f29eb3ddd	135000.0	0	4	0	1	1	1	1.0	0	1	1	0	0	0	0	1	1	1	1	1
2	ID_68de51c94	NaN	0	8	0	1	1	0	NaN	0	0	0	0	1	1	0	1	1	1	1	1
3	ID_d671db89c	180000.0	0	5	0	1	1	1	1.0	0	2	2	1	1	2	1	3	4	4	4	
4	ID_d56d6f5f5	180000.0	0	5	0	1	1	1	1.0	0	2	2	1	1	2	1	3	4	4	4	1

(I did not output here the entire column set for the samples, because it is too large. The full output can be seen in the Jupyter notebook)

So we can see this dataset contains ~ 10,000 records, with 143 features. For ML perspective, it is a **relatively small** data set, with large number of features.

Let's look at the statistics summary of each feature in this data set (Again, I've omitted all the printout, and put here just "samples" of the features, due to the large output of the table):

```
data.describe(exclude = [ 'object'])
```

	v2a1	hacdor	rooms	hacapo	v14a	refrig	v18q	v18q1	r4h1	r4h2	r4h3
count	2.697000e+03	9557.000000	9557.000000	9557.000000	9557.000000	9557.000000	9557.000000	2215.000000	9557.000000	9557.000000	9557.000000
mean	1.652316e+05	0.038087	4.955530	0.023648	0.994768	0.957623	0.231767	1.404063	0.385895	1.559171	1.945066
std	1.504571e+05	0.191417	1.468381	0.151957	0.072145	0.201459	0.421983	0.763131	0.680779	1.036574	1.188852
min	0.000000e+00	0.000000	1.000000	0.000000	0.000000	0.000000	0.000000	1.000000	0.000000	0.000000	0.000000
25%	8.000000e+04	0.000000	4.000000	0.000000	1.000000	1.000000	0.000000	1.000000	0.000000	1.000000	1.000000
50%	1.300000e+05	0.000000	5.000000	0.000000	1.000000	1.000000	0.000000	1.000000	0.000000	1.000000	2.000000
75%	2.000000e+05	0.000000	6.000000	0.000000	1.000000	1.000000	0.000000	2.000000	1.000000	2.000000	3.000000
max	2.353477e+06	1.000000	11.000000	1.000000	1.000000	1.000000	1.000000	6.000000	5.000000	8.000000	8.000000

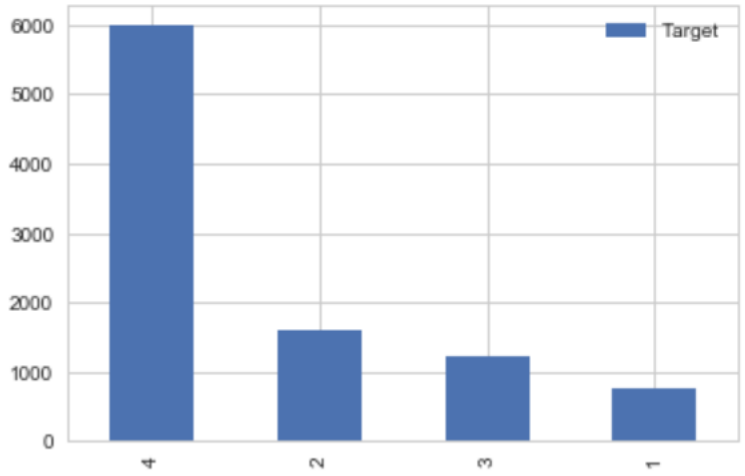
r4h3	r4m1	r4m2	r4m3	r4t1	r4t2	r4t3	tamhog	tamviv	escolari	rez_esc	hhsize
9557.000000	9557.000000	9557.000000	9557.000000	9557.000000	9557.000000	9557.000000	9557.000000	9557.000000	9557.000000	1629.000000	9557.000000
1.945066	0.399184	1.661714	2.060898	0.785079	3.220885	4.005964	3.999058	4.094590	7.200272	0.459791	3.999058
1.188852	0.692460	0.933052	1.206172	1.047559	1.440995	1.771202	1.772216	1.876428	4.730877	0.946550	1.772216
0.000000	0.000000	0.000000	0.000000	0.000000	1.000000	1.000000	1.000000	1.000000	0.000000	0.000000	1.000000
1.000000	0.000000	1.000000	1.000000	0.000000	2.000000	3.000000	3.000000	3.000000	4.000000	0.000000	3.000000
2.000000	0.000000	1.000000	2.000000	0.000000	3.000000	4.000000	4.000000	4.000000	6.000000	0.000000	4.000000
3.000000	1.000000	2.000000	3.000000	1.000000	4.000000	5.000000	5.000000	5.000000	11.000000	1.000000	5.000000
8.000000	6.000000	6.000000	8.000000	7.000000	11.000000	13.000000	13.000000	15.000000	21.000000	5.000000	13.000000

age	SQBescolari	SQBage	SQBhogar_total	SQBedjefe	SQBhogar_nin	SQBovercrowding	SQBdependency	SQBmeaned	agesq
557.000000	9557.000000	9557.000000	9557.000000	9557.000000	9557.000000	9557.000000	9557.000000	9552.000000	9557.000000
34.303547	74.222769	1643.774302	19.132887	53.500262	3.844826	3.249485	3.900409	102.588867	1643.774302
21.612261	76.777549	1741.197050	18.751395	78.445804	6.946296	4.129547	12.511831	93.516890	1741.197050
0.000000	0.000000	0.000000	1.000000	0.000000	0.000000	0.040000	0.000000	0.000000	0.000000
17.000000	16.000000	289.000000	9.000000	0.000000	0.000000	1.000000	0.111111	36.000000	289.000000
31.000000	36.000000	961.000000	16.000000	36.000000	1.000000	2.250000	0.444444	81.000000	961.000000
51.000000	121.000000	2601.000000	25.000000	81.000000	4.000000	4.000000	1.777778	134.560010	2601.000000
97.000000	441.000000	9409.000000	169.000000	441.000000	81.000000	36.000000	64.000000	1369.000000	9409.000000

We can initially already see that the data needs to be preprocessed:
Some feature value are missing (all those features with count < 9557, e.g. **rez_esc**)
Some features are duplicates and\or 100% correlated (e.g. **age** vs **SQBage** vs **agesq**)
Some features are suspected to have data skew (v2a1).

We will deal with all of that during the preprocessing phase.

We can also see in the target distribution that this dataset is highly imbalanced:

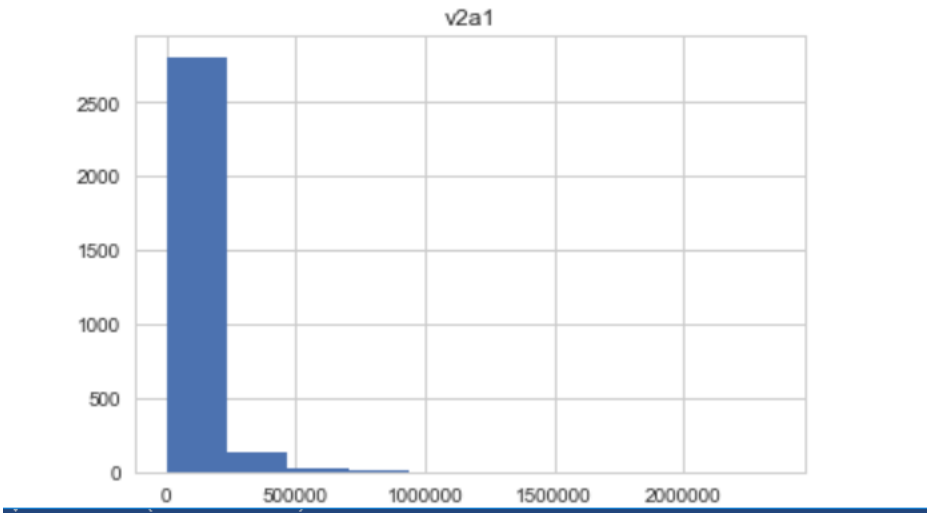


(This is the reason for using the weighted F1 score and accuracy metrics)

Exploratory Visualization

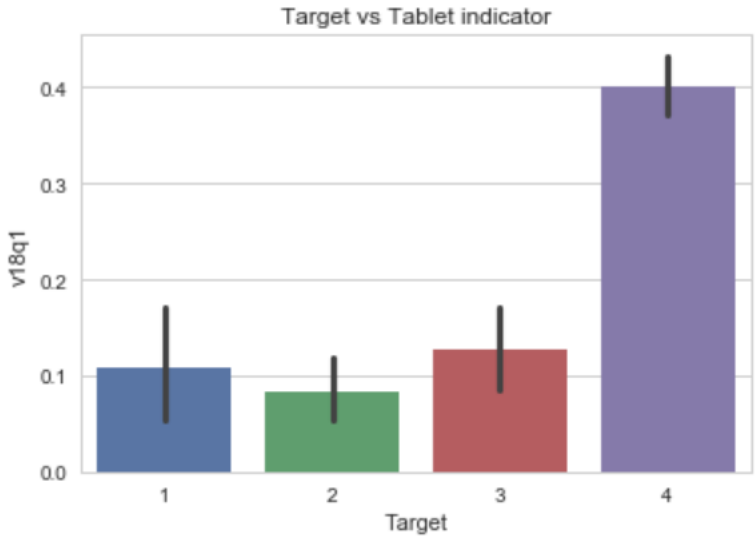
Some of the features that appear to be interesting just by initial looking of the samples and the problem at hand, so I plotted some visualizations of them. Some of the findings were a bit surprising.

```
count       2973.000000
mean       46389.874201
std       112755.371100
min            0.000000
25%            0.000000
50%            0.000000
75%        40000.000000
max      2353477.000000
Name: v2a1, dtype: object
```



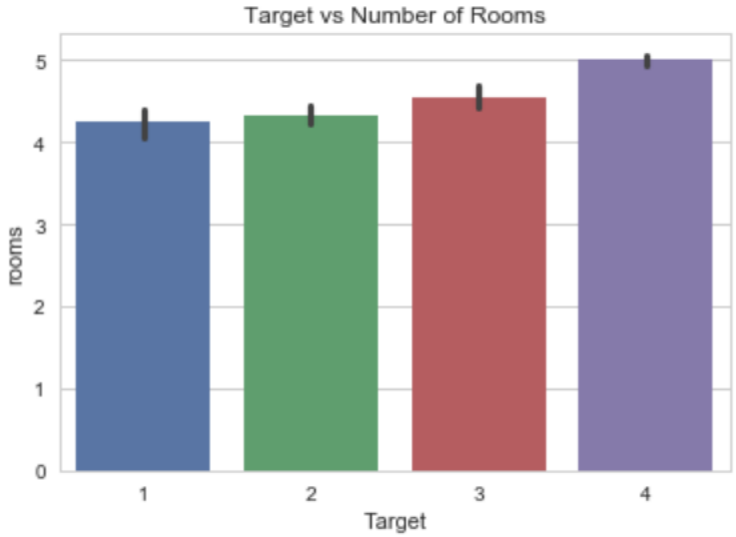
The **v2a1** feature represents the monthly rent payment, and this value can be anything from 2,353,477 to… zero. The average is ~ 50,000, and the standard deviation is much higher than that. I will normalize the values as part of the data preprocessing to avoid data skewness.

One of the features is **v118q1**, which indicates the number of tablets a family owns. I expected that this will have relatively high correlation to the predicted target:



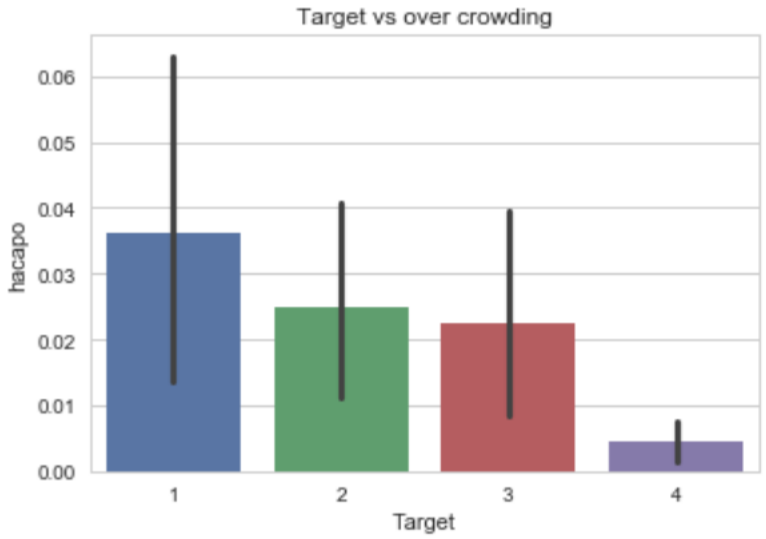
And indeed we can see that for extreme poverty cases (target = 1), there are less number of tablets in the households. However, it is still hard to distinguish between targets 1 \ 2 \ 3.

The next feature I looked at was the number of rooms a family lives in:

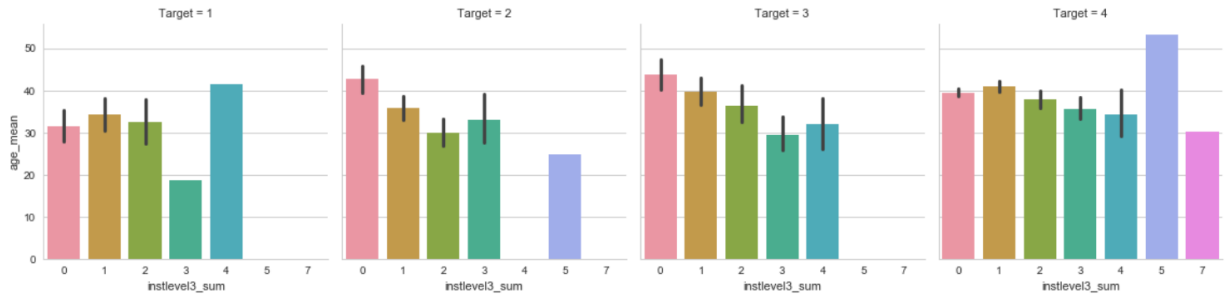


My initial expectations was that in extreme poverty cases the average number of rooms will be smaller, but the difference between the targets is lower than I expected. So this feature would not help much when trying to predict the target class.

Next I checked the overcrowding indicator flag. Maybe this differentiates better between the poverty classes:



It looks like a better predictor. The extreme poverty class are definitely more overcrowded.
One of the key factor I thought can be useful for predicting poverty is ***education***.
Here is a plot that show the connection between the average age and number of completed years in household, per target:



It makes sense that if we look at target 1, we have smaller number of people that completed primary school: only up to **4** people per household, and also we can see that in households that have less than 4, the average age is relatively young (less than 40). Comparing that to the other targets you can see that

Other targets have more people in a household that completed primary school (in target 2 there are also 5 people, in target 4 there are up to **7** people in a family that completed primary school) and\or they are of older age (over 40).

So as a rough statement, the extreme poverty cases seem to be younger and less educated compared to the other targets.

Algorithms and Techniques

Because this problem is a **multiclass classification** problem, I chose to use several known classification algorithms and compare between them. Since I did not know in advance which algorithm would give the best result, I decided to use the following approach:

1. Choose a wide range of algorithms. The list of algorithms below is known to be used for classification problems:

   Gaussian Naïve Bayes
   Decision Trees
   AdaBoost
   K-Nearest Neighbors
   SVC
   Random Forrest
   XGBoost
   LightGBM

2. For each algorithm tune some of its hyper parameters, in order to find a best estimator for a given algorithm. **I used here a "trial and error" approach, since I was not sure in advance which combination of parameters would give the best result**.

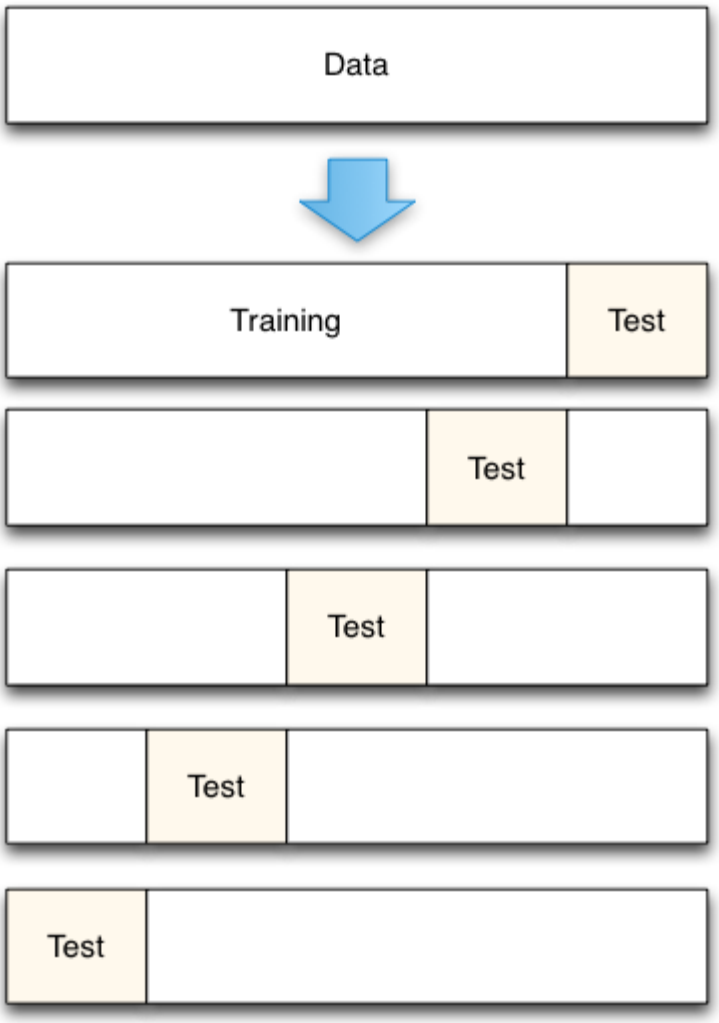
3. I've created a skeleton code which accepts the training set, testing set, labels, classifier and scorer that performs the actual training and logs the training time, accuracy score and F1 score. I've used the logged output to generate the confusion matrix for each algorithm to get a clearer visibility on its accuracy.

4. In order to be able to run the naïve predictor using the same framework above, I've implemented it by creating a **classifier class** (that inherits from **BaseEstimator**), and implemented the required interface methods for it.

5. Because the data set is relatively small, I've used **GridSearchCV** which uses internally **K-Fold** on the input (note that the input here to GridSearchCV is the **train data**, which is separated from the **test data**) → On every fold the data is split into K (in my case I've used the default of K=3 as I saw no substantial difference between different values of K), which is used then as K-1 sets for training and 1 set for testing.

   This way we ensure all the inputs are used for both training and validation and reduce the risk of overfitting. At each iteration, a different sub set of the data is used for training and for testing.

   Here is an illustration of how K Fold cross validation technique is used (again, in my case it's done within scikit learn function)


Data
Training
Test
Test
Test
Test
Test

Benchmark

I've used the ***naïve predictor*** as benchmark model, just to proof that classification model is possible and give substantially better results.

The predictor implementation is quite simple: predict target = 4 (non vulnerable household) for **any** record.
Here is the implementation of this classifier:

The Naive Predictor

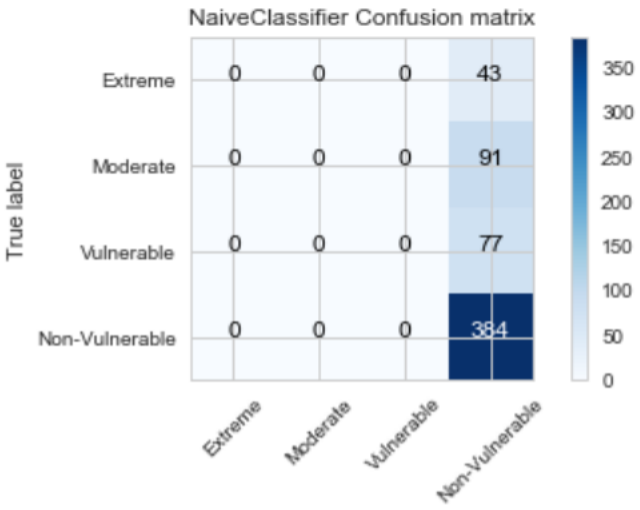
```
In [95]: from sklearn.base import BaseEstimator
         class NaiveClassifier(BaseEstimator):

             def fit(self, X, y):
                 return self

             def predict(self, X):
                 return np.full(X.shape[0], 4, dtype=np.int64)
```

The results I got for this predictor were:

```
completed training using NaiveClassifier classifier
train_time: 0.01999664306640625
pred_time: 0.0
accuracy: 0.25
f1_score: 0.5062823495077295
Confusion matrix, without normalization
[[  0   0   0  43]
 [  0   0   0  91]
 [  0   0   0  77]
 [  0   0   0 384]]
```



Note the **balanced** accuracy score (I will discuss it more in the "Refinement" section below): It is **exactly** 0.25 since this predictor predicts exactly 1 class out of the 4 classes. Needless to say, it is hardly useful for predicting whether a family suffers from extreme \ moderate poverty…

Methodology

Data Preprocessing

I found out that the dataset had to undergo a lot of preprocessing steps in order to make each train-able by the ML classifiers. Because the original data is already very small, I made a hard effort to correct **all** the abnormalities I found, because anything being "left behind" in the dataset could have major influence on the results!

One important observation that I had to take into account is that there are ***different "types" of features in this dataset: features that represent individual data (e.g. person ID, age, number of years in school, etc…) and household data (e.g. number of persons in the household, does the house has refrigerator, etc…)***.

So the data pre-processing has to take this into account and apply the preprocessing separately before aggregating the individual data into household data.

All the pre-processing actions I've made are detailed below.

Separation of data into household and individual

the **parentesco1** feature denotes whether this is the head of a household or not. So I used this feature as the "pivot" to split the data into 2, keeping the household related features in the household data and the individual related features in the individuals' data.

Fixing object data type values

While the "ID" fields should be defined as object type ("Id" is person ID, "idhogar" is household ID), the rest of the fields that are of data type object have mixed numeric and string values:

	Id	idhogar	dependency	edjefe	edjefa
0	ID_279628684	21eb7fcc1	no	10	no
1	ID_f29eb3ddd	0e5d7a658	8	12	no
2	ID_68de51c94	2c7317ea8	8	no	11
3	ID_d671db89c	2b58d945f	yes	11	no
4	ID_d56d6f5f5	2b58d945f	yes	11	no

According to the definition of **dependency** (ratio of too old\too young people to adults), **edjefe** (years of education for male predictor) and **edjefa** (years of education for female predictor), I can replace those string values with 1 for yes and 0 for no.

Removal of entries that have no household head

There were **15** such rows in the dataset. I removed them because according to Kaggle, their labels cannot be resolved, and we are dealing here with supervised learning.

Handling columns with missing values

There were 5 such columns:

	total missing
rez_esc	7921
v18q1	7319
v2a1	6843
meaneduc	5
SQBmeaned	5

The feature **rez_esc** is years behind in school. According to the competition (and real life), this field is relevant only for people whose age is between 7 and 19, so I set the value to 0 for all other people. Also I've set the value to 5 (since it was defined as maximum value in the competition) for people who had higher value.
For the rest of the people with value null, I created a new synthetic field on the data set which is **12-escolari** (escolari is number of years in school, so for anyone who is behind, e.g. completed by now only 9 years, the new field would give a positive number which denotes the years behind in school) and used it to set the value for rez_esc.

The feature **v18q1** is the number of tablets a family owns. Since we have a feature called **v18q** which indicates whether a family has a tablet, I checked whether these 2 features are correlated:

```
data.groupby('v18q')['v18q1'].apply(lambda x: x.isnull().sum())
```

```
v18q
0    7319
1       0
Name: v18q1, dtype: int64
```

So it's an exact match → I can set 0 value in the missing entries.

The feature **v2a1** indicates the monthly rent payment. I used the **tipovivi_xxx** features to verify that maybe this feature can be set to 0, in case rental is not relevant (e.g. the household owns the house).

```
owner_column_names = [x for x in data if x.startswith('tipovivi')]

# check ownership variables values for home missing rent payments
data.loc[data['v2a1'].isnull(), owner_column_names].sum()
```

```
tipovivi1    5897
tipovivi2       0
tipovivi3       0
tipovivi4     163
tipovivi5     783
dtype: int64
```

You can see that it **sums up exactly to 6843, and tipovivi3 (which is for rented) is 0**, which means indeed all the cases can be set to 0 since they don't pay rent.

The last feature is **meanedu** which represents the average years of education for adults (Note that the 5th feature is simply the squared value of this feature).

I first checked if these values are because people did not go to school at all because they are younger than 7, but that was not the reason. **I did find that in all 5 cases these are young people (age 18 or 19) and living on their own or with someone their own age** → no adults in the household!
So I assumed these people did not go to school and set the value to 0.

Feature removal

I dropped all the "squared" features along with my synthetic features used for missing values ('12-escolari').

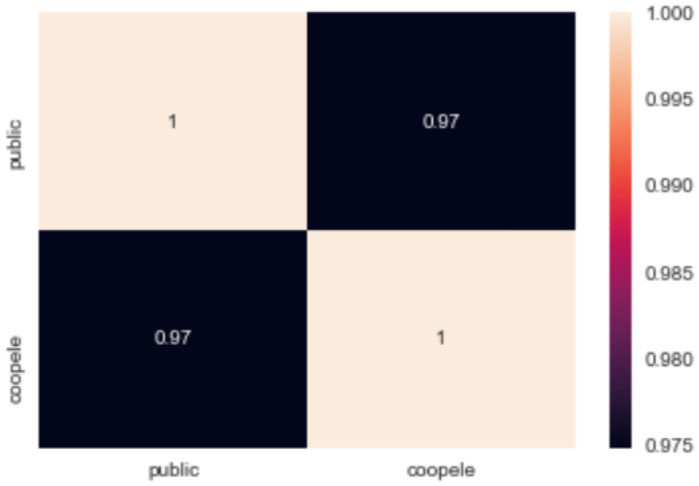
Next I used **correlation matrix** in order to find and drop highly correlated features, starting with the **household** rows. I first checked which features are highly correlated (above 95% correlation), and found the following:

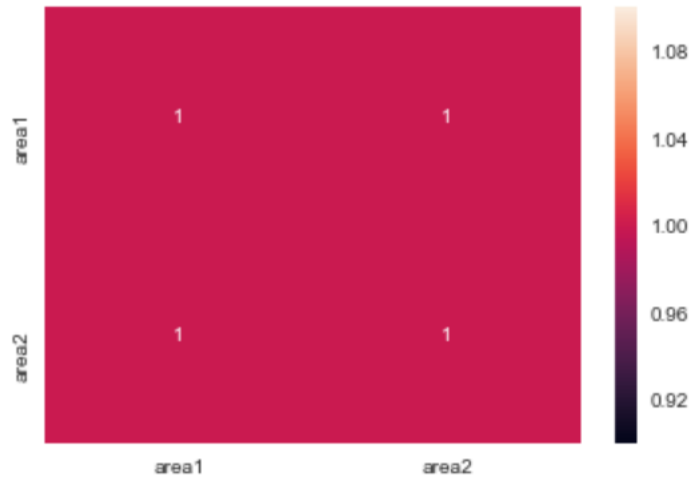
tamhog
coopele
area2
hhsize
hogar_total

Starting with **tamhog**:

Most of these features are indeed identical and represent the number of family members that live in a household, except for **tamviv** which represent the number of persons that live in the household, which may be different (larger). So I dropped **hhsize**, **hogar_total** and **r4t3**.

In the same method I found that I can drop **public** (since coopele and public represent public electricity in almost all cases) and **area2** (since area1 and area2 are exact opposites: rural vs. urban location):





The next step was to apply the same method on the **individuals' data**, and I found that the columns **female** and **male** have 100% correlation (not a surprise…), so I dropped the male column.

I saw this redundant column right by looking at the initial data. Surprisingly, there was no other correlated features in the individuals' data which I hadn't removed already.

Aggregation of individuals' data into household data

Because the individuals features are composed of both **Boolean** (e.g. estadocivil1 = "is less than 10 years old") and **continuous** features (e.g. rez_esc = "years behind in school"), there was no single aggregation function I could use.

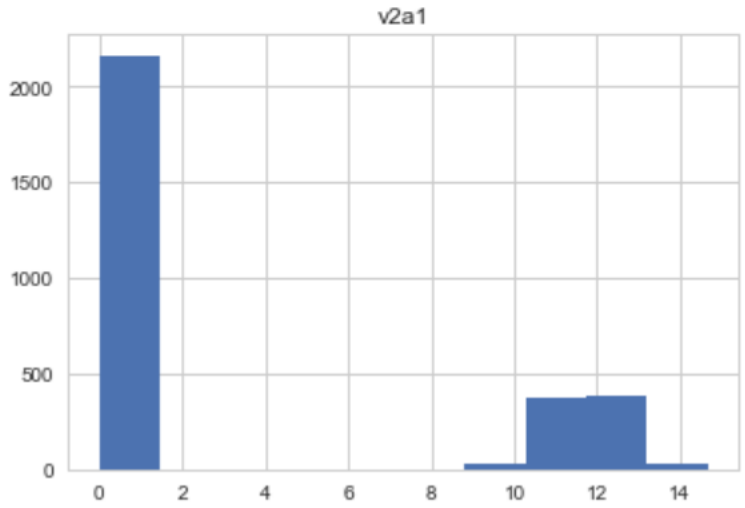
For Boolean features I used the sum function (so I would get for example the number of people less than 10 years old in a household), and for continuous features I used mean, min and max.

I tried to use STD but did not use it because I got NaN value for households that have only 1 person (row) in the dataset, due to a limitation in Pandas:
https://stackoverflow.com/questions/50306914/pandas-groupby-agg-std-nan

Data skew correctness

As seen before, **v2a1** had to be corrected, so I applied logarithmic function on it to gain a more reasonable range of values:

```
count    2973.000000
mean        3.200900
std         5.243638
min         0.000000
25%         0.000000
50%         0.000000
75%        10.596660
max        14.671405
Name: v2a1, dtype: object
```



Data scaling and pipeline for classifiers

I've removed the ID columns and applied the **MinMax** scaler on all the features (now that they are all numeric).

As a final step, the data was split into training data and labels, and I've used 80% of the data for training and 20% for testing.

From experiments I've done using higher percentage for testing ***gave in most cases bad results***, and I suspect this is because the dataset is very small.

After all the steps above the dataset was reduced to 2973 rows with 132 features

Implementation

The following table depicts the relevant software and versions I've used for my implementation

Software Name	Software Version
Anaconda	4.4.10
Python	3.6.4
Jupyter notebook	5.4.0
Numpy	1.15.4
Scikit-Learn	0.20.1

I've applied the following main as part of my implementation:

```
def train_predict(clf, scorer, hyper_parameters, X_train, y_train, X_test, y_test):

    results = {}

    grid_obj = GridSearchCV(estimator=clf, param_grid=hyper_parameters, scoring=scorer)

    # training and finding best model using grid search
    start = time()
    grid_fit = grid_obj.fit(X_train, y_train)
    end = time()
    results['train_time'] = end - start

    # extract the best classifier and use it to make the predictions
    best_clf = grid_fit.best_estimator_
    start = time()
    y_pred = best_clf.predict(X_test)
    end = time()
    results['pred_time'] = end - start

    # calculate accuracy, weighted F1-score and confusion matrix
    results['accuracy'] = balanced_accuracy_score(y_test, y_pred)
    results['f1_score'] = f1_score(y_test, y_pred, average='weighted')
    results['cm'] = confusion_matrix(y_test, y_pred)

    print("completed training using {} classifier".format(clf.__class__.__name__))

    # Return the results
    return results
```

The code accepts as input the estimator, hyper parameters and scorer, and uses Grid search in order to find the best estimator for the given classifier type. Then this estimator is applied on the testing set, and finally, the accuracy, F1 score and confusion matrix are calculated.

The next function I used to output the results:

```
def output_classifier_results(clf_name, results):

    print('train_time: {}'.format(results['train_time']))
    print('pred_time: {}'.format(results['pred_time']))
    print('accuracy: {}'.format(results['accuracy']))
    print('f1_score: {}'.format(results['f1_score']))
    plot_confusion_matrix(results['cm'], clf_name)
```

(The **plot_confusion_matrix** was taken from https://scikit-learn.org/stable/auto_examples/model_selection/plot_confusion_matrix.html)

Using the functions above, running a specific classifier is pretty straight forward. The following code snippet exhibits how to use them:

```
# Random Forest

from sklearn.ensemble import RandomForestClassifier

rf_hyper_parameters = {'n_estimators': [50, 150],
                       'criterion': ['gini', 'entropy'],
                       'max_depth': [2, 4, 8],
                       'min_samples_split': [2, 4, 8],
                       'min_samples_leaf': [2, 4, 8]}

# using class_weight = 'balanced' produced better accuracy score because of the data imbalance we have
rf_clf_res = train_predict(RandomForestClassifier(random_state = 42, class_weight = 'balanced'),
                           f1_weighted_scorer,
                           rf_hyper_parameters, X_train, y_train, X_test, y_test)

output_classifier_results('RandomForestClassifier', rf_clf_res)
```

Refinement

There were several refinements that I had to make throughout the implementation of running the classifiers:

The 1st refinement was that I switched between using **accuracy score** to **balanced accuracy score**: when running the naïve predictor, the accuracy score I got was **0.6453781512605042**. This was relatively high, and then I realized that the classifier was getting this high score because it was not "penalized" for not predicting correctly for the other 3 classes! Switching to the balanced accuracy score made more sense in the final output.

I also had to change some hyper parameter values for several algorithms to reach the best results (per algorithm):

Random Forest: I tried at the beginning small number of estimators (<50) and improved the results only when working with higher numbers. I also found that using **balanced class weight** produced better results compared to the default → this makes sense because the data is unbalanced.

Decision Tree: I used several values for hyper parameters here, but the only thing that seem to make any difference was using the **balanced class weight** (like Random Forest)

SVC: I tried using different decision function shapes but only “ovo” worked out. I also tried different kernels (including ‘linear’, ‘sigmoid’), but none of them made any difference in the results.

AdaBoost and LightGBM: I’ve started with number of estimators<100, and managed to improve the results only when working with over 150 estimators. I did find, however, that using too many estimators also degrades the results. I think that the best value is somewhere around ~ 300 estimators for both of them.

XGBoost: I did not tune this model much, because I found its execution time VERY long, and each hyper parameters tuning took a very long time. So at the end I used only the objective parameter for tuning.

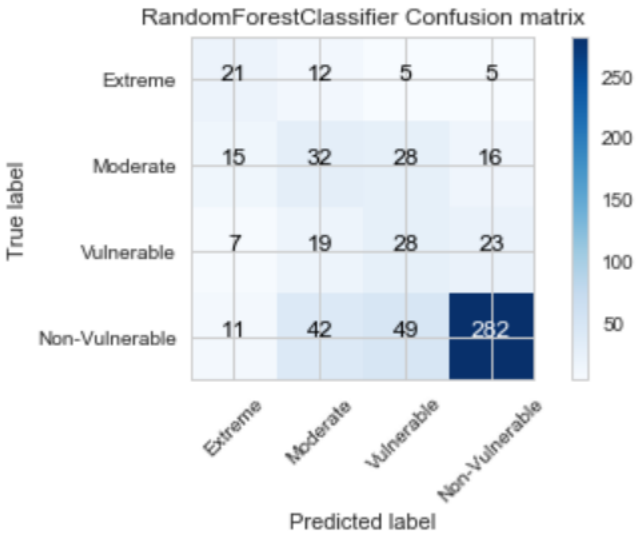
Results

Model Evaluation and Validation

I’ve found out that the best models that solved this problem were **Random Forest** and **LightGBM**. Here are their results:

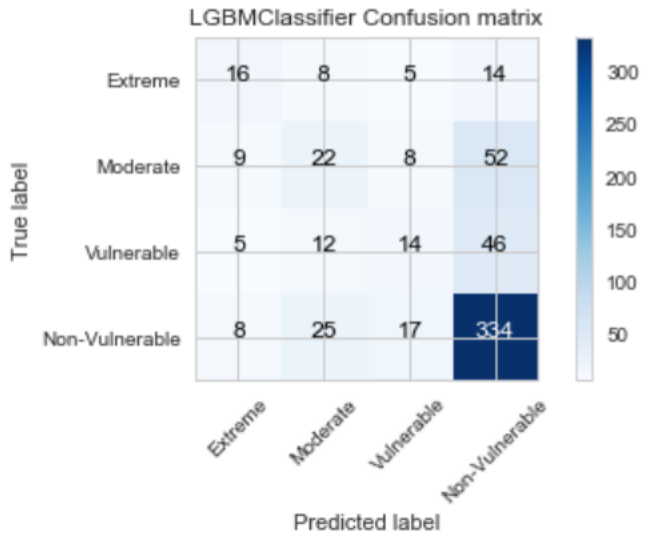
Random Forest

```
completed training using RandomForestClassifier classifier
train_time: 123.3260543346405
pred_time: 0.043999433517456055
accuracy: 0.4845079520769928
f1_score: 0.6326525809462828
Confusion matrix, without normalization
[[ 21  12   5   5]
 [ 15  32  28  16]
 [  7  19  28  23]
 [ 11  42  49 282]]
```



LightGBM

```
completed training using LGBMClassifier classifier
train_time: 52.446016788482666
pred_time: 0.023005008697509766
accuracy: 0.41636527837472603
f1_score: 0.6205011944549694
Confusion matrix, without normalization
[[ 16   8   5  14]
 [  9  22   8  52]
 [  5  12  14  46]
 [  8  25  17 334]]
```



The best parameter values for each of the models were:

Random Forest

Parameter name	Parameter value
Criterion	Gini
Max_depth	8
Min_samples_leaf	2
Min_samples_split	2
N_estimators	150

LightGBM

Parameter name	Parameter value
Learning_rate	1.5
N_estimators	250

I think that the main thing that influenced the values of the parameters I got for the best models is the fact ***the size of the data set***. Because the dataset was very small, the learning rate and number of estimators in LigthGBM had to be relatively large because lower values could not achieve good results.

The same goes with Random Forest: when I initially started with 20, 30 and 50 estimators, the results were poor. I had to increase the number of estimators to gain better results. Also you can see that the minimal number of samples per leaf and the minimum samples required for a split are low values: higher values would produce a "shorter" decision tree and with such small inputs it would cause the prediction to be inaccurate. In order to gain "higher" tree, we have to settle for small values in leafs and splits.

Principal Component Analysis
As a final step I wanted to apply PCA on the given dataset, and apply the best models on the reduced feature set, seeing how\if the results were changed.

I found out that **35** components explain **~ 90%** of the features variance, so I reduced the feature set from 131 features to 35. The following table summarizes the results:

	Train time	Accuracy score	F1 score
Random Forest	78.5	0.48	0.63
Random Forest PCA	208.3	0.40	0.62
LightGBM	56.5	0.41	0.62
Light GBM PCA	74.4	0.37	0.61

I think that because of the dataset size, the training time was not improved (but in fact there are worse). I think that with a larger dataset these values may be different. Regarding the quality metrics, there is a medium change to the accuracy score (larger in random forest, smaller in LightGBM) and a very small change to the F1 score.

Due to the results above, I think it **makes sense to apply PCA on the original data**, especially if the algorithms would be applied in the future on a **larger** dataset (in terms of number of rows).

Justification

Comparing the results on the original dataset of the 2 best classifiers to the classifier used for the benchmark:

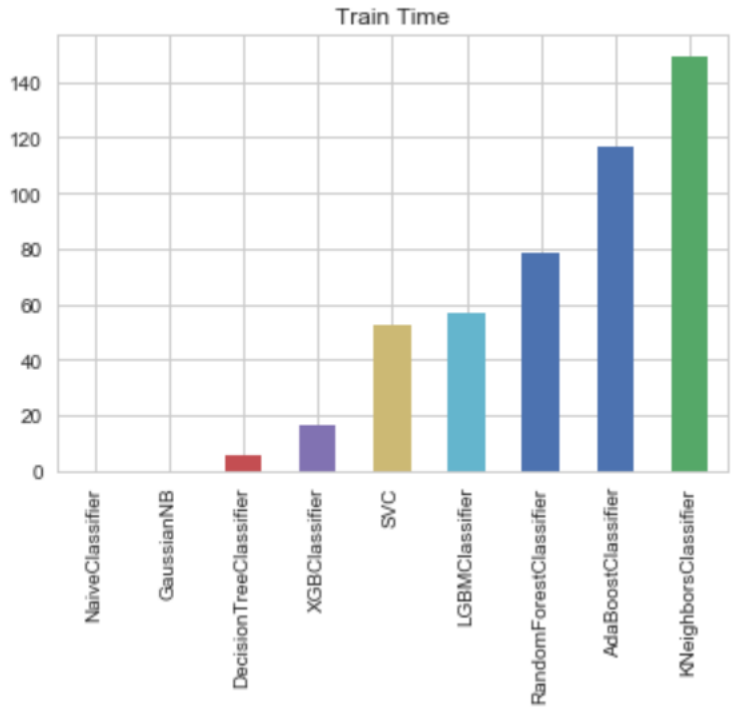
	Train time	Accuracy score	F1 score
Random Forest	**78.5**	**0.48**	**0.63**
LightGBM	56.5	0.41	0.62
Naïve classifier	**0.020**	**0.25**	**0.51**

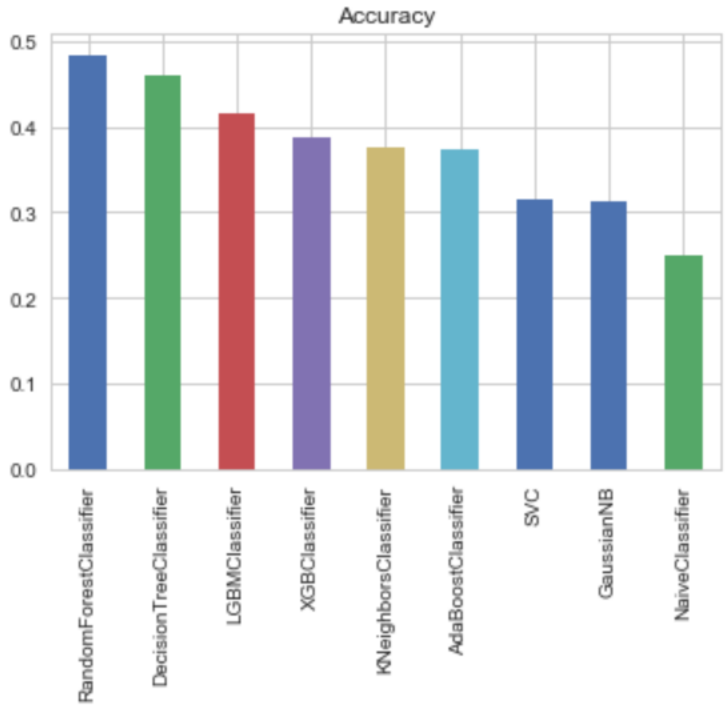
The results are quite self-explanatory: The best classifier is almost **2 times (!)** accurate compared to the benchmark model, and has **~ 20%** improvement in its F1 score. I believe that both the accuracy and F1 score can even be further improved (see "Improvement" section below) to even further make ML classifiers more appealing in this case.

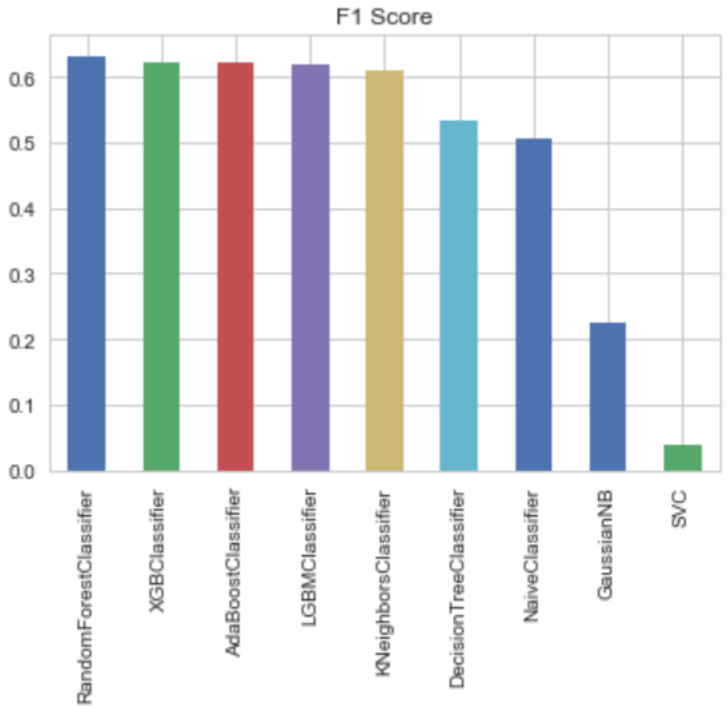
Conclusion

Free-Form Visualization

The following graphs summarize all the ML classifier algorithms I tried out throughout the project and the results per each evaluation metric:




Accuracy
0.5
0.4
0.3
0.2
0.1
0.0
RandomForestClassifier
DecisionTreeClassifier
LGBMClassifier
XGBClassifier
KNeighborsClassifier
AdaBoostClassifier
SVC
GaussianNB
NaiveClassifier


F1 Score
0.6
0.5
0.4
0.3
0.2
0.1
0.0
RandomForestClassifier
XGBClassifier
AdaBoostClassifier
LGBMClassifier
KNeighborsClassifier
DecisionTreeClassifier
NaiveClassifier
GaussianNB
SVC

I think that there are some interesting conclusions that can be drawn from these graphs:

1. The more learners\estimators being used, the better results I got: all the algorithms that are in the top of the graphs in terms of quality metrics (random forest, XGB, Ada Boost, LGB) share a common attribute which is that they combine **multiple** learners (each one with a different way of aggregating the learners knowledge: bagging, boosting, etc…).

2. The algorithms that are a single learner (SVC, Gaussian, etc…) produced weaker results. I think that specifically for GaussianNB the results were even poorer than expected, because the features likelihood in the dataset does not follow Gaussian distribution.

3. In terms of training time, the results are naturally opposite: single learners are faster compared to multiple ones. However, because the dataset was small, I don't think there should be much emphasis on this issue, as the differences are not that large.

Reflection

The goal of this project was to find and test the best classification algorithms for detecting poverty and extreme poverty.

The solution implementation had to address the following issues:
- initial analysis of the data
- data pre-processing
- creating a framework which can then be re-used for running multiple classifiers
- extracting and outputting the results

To me the most challenging and interesting part of this project was the **data pre-processing** part. I was not aware at the time I started to work on this project how much work is required, and I found myself handling many subtle end cases, before reaching the point where I felt I was ready to actually execute the algorithms on the data.

I think that the results I got can serve as a basis from which algorithms can be improved and revised (more on that in the "Improvement" section below)

Another important aspect of this project was dealing with evaluation metrics for **imbalanced** data set: using the **balanced** accuracy score gave me more real life results compared to the regular accuracy score: Using the "regular" accuracy, all models got ~ .60 accuracy, but when looking at the confusion matrix, you saw that this is because the data is imbalanced. When switching to balanced accuracy, the results made more sense, and no model even passed the ~ 0.50 accuracy.

Improvement

There are several aspects in which the work done in this project can be further improved:

1. **Feature engineering**: I did not apply any feature engineering at all. Because the dataset contains many feature, it makes sense that some features can be removed while new features can be constructed from the existing features.

2. **Further hyper parameters tuning**: I applied several combinations of parameters, but of course there are far more possibilities than outlined here. Since the execution of the code was done on a local laptop, I had to take into account the long executions time. Running on a GPU (for example), would enable testing much more combinations in lesser time, probably yielding even better results.

References

https://www.dosomething.org/us/facts/11-facts-about-global-poverty
https://en.wikipedia.org/wiki/Poverty#Poverty_reduction
https://www.worldbank.org/en/topic/poverty/overview
https://olc.worldbank.org/sites/default/files/1.pdf
https://www.kaggle.com/c/costa-rican-household-poverty-prediction
https://chrisalbon.com/machine_learning/feature_selection/drop_highly_correlated_features/
https://scikit-learn.org/stable/auto_examples/model_selection/plot_confusion_matrix.html (for plotting confusion matrix)
https://www.kaggle.com/willkoehrsen/a-complete-introduction-and-walkthrough (for splitting columns to 'household' ones vs. 'individual' ones in a nice manner using column group variables)