Poverty Identification in Nepal

Comparative analysis of machine learning
algorithms applied to poverty prediction.

by
Atit Bashyal

A report presented for the degree of
Masters of Science in Data Engineering

Prof. Dr. Adalbert F. X. Wilhelm
Jacobs University Bremen
Prof. Dr. Stefan Kettemann
Jacobs University Bremen

Contents

Abstract

Poverty eradication has been a significant aspect of the development goals of developing and developed countries for a long time. Methods of eradicating poverty have been a pressing topic of research in the past decade. To figure out a strategic plan for eradicating poverty, it is essential to understand the causes of poverty while measure its changing levels and predicting its trends.In recent years the advancements in computational power and development of new statistical methods have enabled diverse applications of data-driven decision making. These advancements have been able to break the barrier of limitations that researchers had to face previously in the task of estimating and predicting poverty.

This project aims to compare the performance of several machine learning classification algorithms, in the task of predicting the poverty status of households in Nepal. The target variable to be predicted by each algorithm is a label "Poor" or "Non-Poor". The algorithms are trained and predictions are evaluated Using national-level data from the Nepal Household survey 2011. The project does not aim to build a best possible model for poverty prediction but compares existing open source algorithms across multiple metrics.

1 Introduction

The field of Machine Learning has seen a rapid growth in the recent years which has changed the way the world engages with data especially in terms of using data for decision making. The trend of using machine learning algorithms for data driven decision making has found its way in the field of economics and social development, where data serves to be an input for improving and understanding the consistency of theoretical frameworks of economic and development policies.[1] Following the trend of utilizing open source ML algorithms to aid the process of evaluating social development this project aims to Understand the mathematical framework utilized by ML algorithms in evaluating such socioeconomic dataset. In particular the objective of this project is to explore and investigate statistical structures present in the dataset obtained from Nepal Living Standards Survey (NLSS) 2011 and use the dataset to train several machine learning classification algorithms in predicting Poverty.

Using Machine learning algorithms enables various setups of the poverty prediction problem, this opens up ways for framing the poverty prediction problem as a problem which utilizes the probability distribution of variables present in a survey dataset to estimate the poverty level of households present in the dataset. The most common and widely used method is to set up poverty prediction as a simple regression problem to predict the household per ca pita consumption and comparing it with existing poverty lines. The poverty prediction problem can also be framed as a classification problem where the aim is to predict,for each household present in the dataset a class label value of ”poor” or ”non-poor”. New advanced ML classification algorithms allow poverty prediction to be setup not just as a binary classification problem but as a multi-class problem by stratifying the households into various class groups of ”extremely poor”, ”medium poor”,”poor” and ”not poor”.[2] In this project as at looking at poverty prediction problem as an binary classification problem and compares the performance of several open source classification algorithms in predicting Poverty. The software used during this project for data preprocessing, data visualisation and algorithm training/predicting is Python. The python scripts for all of the processes implemented in the project can be found in Appendix 2. Additionally all of the open-source python packages used can be referenced from Appendix 3.

Section 2 of this report focuses on defining and describing in mathematical formalism the original dataset obtained from the NLSS 2011 survey, to understand the variables that will be used for classification and how they came into existence. Section 3 discusses the steps taken in preprocessing and cleaning the dataset before using it for training the classification algorithms. Section 4 explores graphically the processed dataset to look into structures present in the dataset that differentiate between 'poor' and 'non-poor' households. Section 5 focuses on discussing in mathematical formalism the general mechanisms of classification algorithms followed by discussion of the various classification algorithms used in the project. Section 6 of this discusses the different evaluation metrics used in the project to compare the performance of the classification algorithms used. Finally section 7 presents the results of poverty predictions

and performance comparison of the different classification algorithms.

2 The NLSS Dataset

The data for this project includes data and metadata from the Nepal Living Standard Survey(NLSS) conducted in the year 2011.The NLSS survey is aimed at collecting data with the objective of measuring the living standards of the people in Nepal. The data is often utilized to determine the level of poverty in Nepal. The survey covers a wide range of topics related to household welfare which include demographic structure, monthly and weekly consumption, income, access to facilities, health, education etc. Results published from the NLSS survey has been of prime importance for government agencies and other organisations to assess the impact of policies and programs on socioeconomic changes in Nepal.

2.1 NLSS 2011 Data

The NLSS 2011 data obtained for this project included:

- The survey questionnaire
- Stata (version 14) data files: household data and individual data. These data contain a subset of the full survey data.

Each of the Stata data files contain:

- The binary variable to be predicted, named “poor”. The variable is a dummy variable indicating if a household or an individual falls into the category poor/non-poor.
- The sample weight Variables for the household weight ”wt_hh” and the population weight (household weight * household size) ”wt_ind”
- Set of 258 response variables present in the survey data

The data file used in the project to build the training/test dataset is the household level data file. The reason to do so is because poverty prediction is done at the household level i.e. either all members or no member of a particular household are categorized as poor[1]. The individual level dataset allows us to create additional variables to be added to the household dataset which might improve the prediction accuracy of the algorithms used however exploration of such variables are not done in this project.

2.2 Mathematical Formalism of the Survey Data set

When data is collected, the data collection process can be described using different mathematical components. Understanding these components of the data collection process helps us understand the dataset better and generate a mathematical abstraction of the dataset. The components that we will explore are as follows:

- **Universe and Elementary Events :** The Universe is a part of the real world from which we collect data. In mathematical terms the universe is represented by a set Ω. In the particular case of considering the NLSS dataset the Universe set is the collection of all the Households of Nepal in the year 2011.

  Elementary events are elements of the universal set from which the observation data is collected. Mathematically the elementary events are represented by ω are elements of the universe set Ω. Following the definition of the universe set provided for the NLSS data, the elementary events from which the NLSS data was collected are the households present in the universal set Ω. [3]

- **Data Value Space:** The data value space mathematically is also set which contains all the possible outcomes of an observation procedure acting on elementary events, which we also can call the "observation act".[3] In particular when considering the NLSS data, each question present in the survey questionnaire is an observation procedure and a household members answering the questions during the survey is the observation act on an elementary event(the household). A review of the survey questionnaire, shows that data values generated as answers to the questions are of three different types:

  - **categorical value :** The survey questions that output a categorical value produces a data value space of finite sets. These finite sets can further be divided in to two types depending on the type of values the elements of the sets have.

    * **Nominal valued elements:** These sets are finite and have elements that have no natural order and no numeric value. For the purpose of this project we will represent sets with nominal valued elements by c_i where i represents the index of questions in the survey questionnaire that output Nominal values. One such question in the questionnaire asked which ethnic group in Nepal does the household head identify with. The data value space generated by this particular question will be the finite set of 11 different ethnic groups recognized by the government of Nepal. Figure 1 below shows the total count of households that belong to these 11 different ethnic groups, based on the NLSS dataset. Next to make our task of defining the data value space easier we create a single nominal data value space represented by $\mathcal{C}$ by defining it to be the set product of c_i

$$\mathcal{C} = \times_i c_i \tag{1}$$

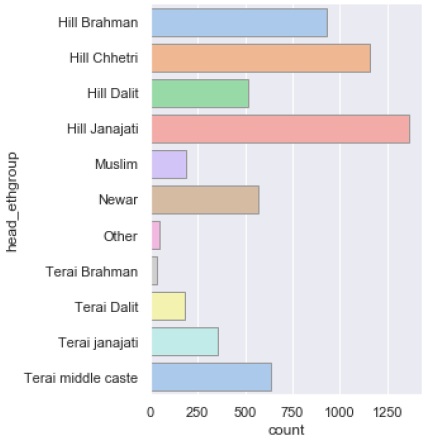


Figure 1: total count of households belonging to different ethnic groups in Nepal

* **Ordinal valued elements:** These sets are finite and have elements that have a natural order and a numeric value. The survey questions that output such values are questions that have yes/no answer. The natural ordering given to the answer of such a question is most commonly yes=1 and no =0. we represent such sets for the purpose of this project by $d_j = \{0, 1\}$ where j is is the index representing questions in the survey questionnaire that have a yes/no answer. One such question in the questionnaire asks if the household gets access to hospital facilities within 30 minutes. The data value space generated by this particular question will be the finite set {0,1}. Figure 2 below shows the total count of households according to access of hospital within 30 minutes, based on the NLSS dataset. Similar to defining the nominal data value space, we can define a ordinal data value space represented by $\mathcal{D}$ as the set product of d_j.

$$\mathcal{D} = \times_j d_j \tag{2}$$

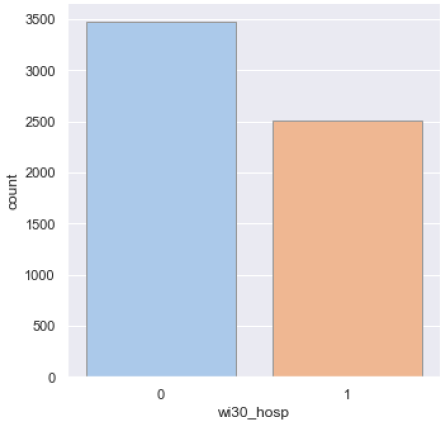
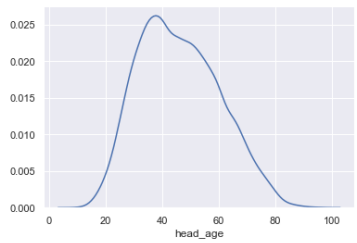


Figure 2: total count of household based on access to hospital within 30 minutes.

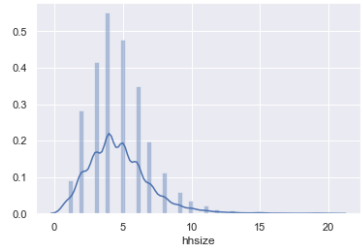
– **Numerical values:** Some survey questions output a numeric value. These questions will create either a discrete data value space or a continuous data value space. The discrete valued data value space are again sets of various lengths that contain elements belonging to the real numbers. We will represent these sets by g_k where k is the index of question that output a discrete data value space. Then we can further define a discrete valued data space such that,

$$\mathcal{G} = \times_k g_k \tag{3}$$

on the other hand the continuous data value space contains different intervals of the real number line. To make the task of defining this continuous data value space we can merge the real line interval $[0, \infty]^n$ where n is the number of questions that require a continuous numerical answer. I have defined the upper limit of the line interval to be ∞ because, the data value space must contain all the possible outcome values, defining these intervals for each questions will require a long section and detailed discussion of the particular questions therefore to make the task simpler, I have defined the interval with an upper bound of ∞. Most of the numerical values resulting from the questions are discrete and a few of them are continuous. One question which outputs a discrete numerical value is the household size. Similarly the age of the household head outputs a continuous numerical value. Figure 3 below shows plots of the distribution of the household size and age of the household head, based on the NLSS data



(a) distribution of household head age (continuous numerical)



(b) distribution of household size(discrete numerical)

Figure 3: distribution of the variables household head age and household size the y axis has been normalized to make the area under the distribution curve 1

These different data value spaces created depending on the type of answer each question of the questionnaire requires can be merged together into a big data value space $\mathcal{S}$ which is mathematically represented as

$$\mathcal{S} = \mathcal{C} \times \mathcal{D} \times \mathcal{G} \times [0, \infty]^n \tag{4}$$

- **Random Variables:** are functions which map the elementary events from the universe set into data values in a data value space. The random variables are the mathematical formalism of representing the observation procedure.[3] This follows that each question of the questionnaire, $q_l \mid l \in \{1, ...n\}$ where n=258 is the total number of question in the questionnaire, is a random variables that maps the elementary events into the data value space. Some examples of the data value space mapped by these individual random variables (questions) have been discussed above. These individual random variables q_l can also be expressed as a compound random variable $\mathcal{Q}$, where:

$$\mathcal{Q} := \bigotimes_{i=1}^{258} q_l \tag{5}$$

This compound random variable $\mathcal{Q}$ is thus the mathematical representation of the survey questionnaire. Following these definitions of the random variable and data value space the process of collecting the household data is then represented in mathematical formalism by:

$$\mathcal{Q} : \Omega \rightarrow \mathcal{S}^N \tag{6}$$

where p is the total number number of questions in the who were included in the survey.

Mathematically formalizing the data value space and the random variables gives us an mathematical description of how the data values of 258 columns of the dataset are generated from the elementary events in the universe, and defines a data-value space where

the household characteristics have been defined mathematically. There are two other columns in our dataset namely the binary variable "poor" and the survey weight variable "wt_hh". The values that these two variables take are not a direct result of the random variable $\mathbb{Q}$ acting on the elementary events. These variables have been generated using information from the mathematical representation of the households and household characteristics in the data value space generated by $\mathbb{Q}$ and the design/modelling choices made by the data collection agency, the Central Bureau of Statics Nepal (CBS).

Survey weights are typically a value assigned to each case in the data file. These values are used to make statistics computed from the data more representative of the population. In general, the survey weight is calculated for each case of the data file (in our case each household) using the following formula:

$$wt\_hh = 1/(sampling fraction) \tag{7}$$

where sampling fraction for each household in a particular strata of the sample is calculated using:

$$Sampling fraction = n/N \tag{8}$$

where n represents the sample size of a particular strata and N is the population size. The division of households in a particular strata is a consequence of how the CBS defines each particular stratum. The metadata does not discuss the process of choosing the sampling strata and the sample size of each strata so formalizing this process further is out of scope of this project.

On the other hand the variable "poor" which categorizes each household as "poor" or "non-poor" has been computed using a method called the Proxy Mean Test. In formal mathematical terms, the Proxy Mean test is an estimator which estimates the level of household income based on the different household characteristics described by the dataset. as a further step this income estimate is then compared with a predetermined standard income level to categorize the household as "poor" or "non-poor". In formal mathematical definition the Proxy mean estimator (which we represent by ρ for discussion in this project) is a function, such that it computes the household consumption based on the answers from the survey questionnaire, this predicted consumption level is then used to label a household poor or non-poor. Mathematically,

$$\rho : \mathcal{S}^N \to \mathbb{R} \tag{9}$$

finally the whole process of collecting data and applying the proxy mean test to label households as poor and non-poor can be represented mathematically by the composition of the proxy mean estimator ρ and the Survey Questionnaire by,

$$\rho \circ \mathbb{Q} : \Omega \to \{0, 1\} \tag{10}$$

where a household categorized by a label of 0 is "non-poor" and with a label of 1 "poor".

3 Data Preprocessing

This section introduces the NLSS 2011 data and metadata that will be used for the project and will also explain the process of cleaning and transforming the responses into a data format suitable for applying the machine learining algorithms. The process of transforming survey responses into a dataset that is fed into Machine learning algorithms requires transforming each survey response into a pair of feature vectors $x_i \in \mathbb{R}^m$ and and class label y_i. where, $i \in N$ represents a particular household included in the survey process and n represents the number of questions in the survey questionnaire .

Although, the survey data used in this project was cleaned and stored in useful format in the Stata files, there were small issues to deal with before spliting the data into training and test datasets and using them. The issues that addressed include:

- The data is loaded into the python environment using the read_stata function present in the pandas library of python. There is a difference in how python panda environment and stata deal with categorical variables. In order to remedy this, data must be loaded in python without converting the categorical variables. In the second step the labels must be loaded saperately using pandas StataReader function to create a python dictionary for mapping of the categorical variables and the labels. In the next step the values of the original categorical variables are replaced with labels using the dictionary mapping created and finally converted to categorical variables.

- Households records in the household dataset have missing values, the exact number of households that have missing values is 302. Since this is a small number of households compared to the total 5988 households present in the survey dataset, these records are dropped.

- Since preprocessing of the data is done once before splitting the dataset in to train/test in order to start training and tuning the different algorithms using the same dataset we must consider the fact that some classification algorithms are unable to handle categorical variables, i.e. they require all input to be numerical[2]. To deal with this problem, the easiest method is to create dummy variables where a categorical feature are taken and binary columns are made for each value of the categorical feature. The pandas library in python contains a function get_dummies that takes a categorical variable and creates dummies.

- Removing features that are not useful for a classification problem. The dataset contains constant, empty and duplicate columns that are not useful in training classification algorithms. In order to drop the constant and empty columns;columns where there are more than 1 unique values are identified and kept in the data frame while other columns are removed. Identification of duplicate columns is done by looking into the pairwise correlation of columns in our data frame and dropping one of the two variables where strong pairwise correlation is seen i.e. if the correlation is equal to 1.

4 Dataset Exploration

After the preprocessing of the Data and converting them into a format needed to train classification algorithms, it is useful to explore the data and look into what is actually contained in the data. Exploring the data gives a better understanding of how the different feature variables are related to the target variable 'poor'. Although Exploring data also plays an important role in deciding which features are important to focus on, the large number of features present in the data we use makes it difficult to explore the relation between the target variable and each individual features. But it is still useful to explore a few features of the data.

Exploring to see presence of distinct relationships between features and the target variable can be achieved by plotting the distribution of features against the target variable. The NLSS survey dataset has several features that are not categorical. The distribution of these features between poor and non-poor responses can be compared to see if the distributions are separable, a clearly separable distribution indicates that the particular feature in useful as a predictor. Figure below shows the comparison of eight such features between poor and non-poor responses.

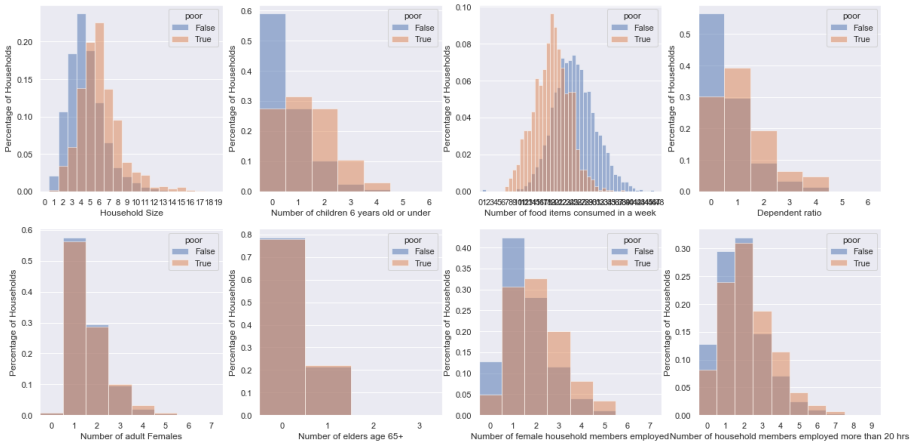


Figure 4: distribution of numerical features between poor and non-poor responses

The histograms show that the distribution of most numerical responses between poor and non-poor are not distinct, except for the distribution of household size and the number of food items consumed in a week. In general it is expected that these two numerical features will function better as predictors. The distribution of the food items consumed in a week shows the best distinction in its distribution between poor and non-poor households. This hints that features related to household consumption will play an important role as better predictors. The survey data does not contain consumables as features, but have information on share of monthly household expenditure in consumption of goods and facilities such as education,electricity,garbage disposal,

tobacco, food consumption etc. Plotting the % difference between poor and non poor households in monthly expenditure share of these goods and services gives an idea of expenditure pattern of poor and non-poor households.

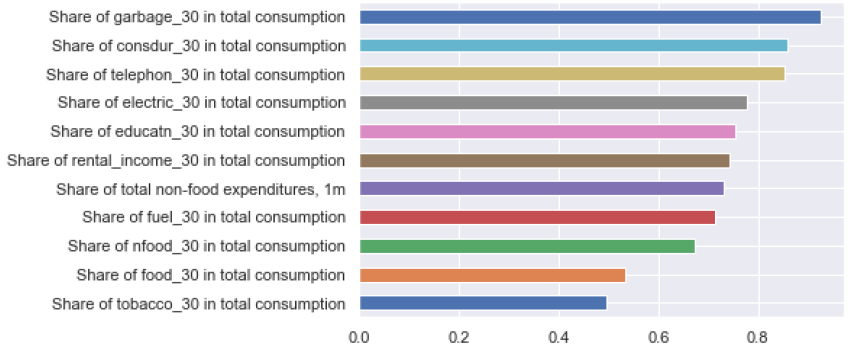


Figure 5: % difference between poor and non poor households in monthly expenditure of goods and services in descending order

The plot above shows that poor and non-poor households have the largest difference in expenditure on goods and services that are beyond the basic needs for survival. For example, the largest difference is seen in the share of expenditure in services of garbage disposal, i.e. non-poor families have expenditures in proper disposal of garbage and waste while poor families tend not to use such services. Therefore, we can expect monthly expenditure share features on such goods and services to be good predictors to differentiate between poor and non-poor households.

One final exploration of the data that we can do is compute the actual number of poor and non poor households that are present in the dataset. The NLSS dataset used in this project identifies 19.2% of the total households present in the dataset as poor households and 80.8% of the households as non-poor. This information clearly shows that the dataset is unbalanced and can possibly be a source of difficulty for our classification algorithm. The method this project uses to deal with the unbalanced dataset is discussed in the next subsection.

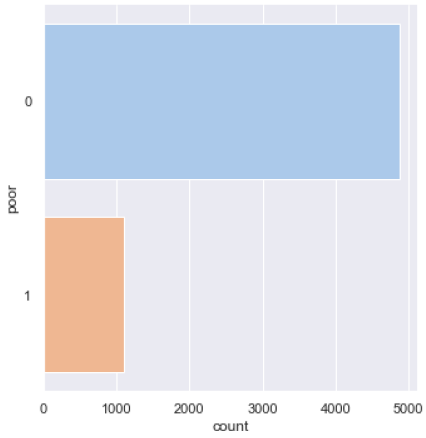


Figure 6: Count of poor and non-poor households labeled in the Dataset.

4.1 Dealing With Class Imbalance

The method implemented in this project to deal with the problem of unbalanced class is Oversampling the under-represented class. One of the most popular oversampling methods is SMOTE(Synthetic Minority Oversampling Technique), present as a function in the python package Python package called ”imblearn”. This method creates synthetic samples of the under-represented class by finding nearest neighbors and making minor random perturbations.[4] while training each classification algorithm in this project, the training has been done once on the imbalanced training dataset and once on the oversampled training dataset, through which we also compare the performance of each algorithm when dealing with imbalanced dataset.

Oversampling method has been choosen over other methods like using class weights and undersampling because

- Increasing the sample size by oversampling is expected to preform better than undersampling method which would decrease the sample size. A larger sample size is expected to give the algorithm a better chance of learning.
- All of the classification algorithm models present in the Scikit-learn python package do not have parameters that can be passed for class weights. Therefore to make all the training processes of the different algorithms uniform, this method is dropped.

5 Classification algorithms

5.1 Mathematical concept behind classification

Classification is a supervised learning task, where the dataset used must be represented as a labelled pair:

$$(x_i, y_j) \tag{11}$$

where x_i are the different features that represent a case in the dataset that has been labelled using a class y_j. In the NLSS dataset we have each household represented by features (the household characteristics, collected from the Questionnaire) and a class label represented by the value of the "poor" variable which we will refer to as the class variable. A classification algorithm is also an estimator, which takes as input a dataset and produces an class estimate for objects in the dataset. Using classification algorithm to classify objects in a dataset requires two steps, the first step which we classically call the "training" step and the second step the "testing" step. The training step is where we input a subset of our dataset as training-data into the classifier and allow the algorithm to "learn" from the training data. The objective of the learning process is to obtain a "well learnt" algorithm that when fed with the test dataset generalizes in a meaningful way what it has "learnt" from the training sample. The "learning" task looked upon mathematically, is trying to estimate the conditional distribution:[5]

$$P(Y = y_j \mid X = x_i) \tag{12}$$

In reality, estimating this conditional distribution becomes a difficult task, therefore most algorithms that are used for classification look at the problem of classification as a decision function approximation problem where the classification algorithm, i.e an estimator (that we generally represent as $\mathcal{D}$ is mathematically defined as:[6]

$$\mathcal{D} : x_i \rightarrow y_i \tag{13}$$

This follows that the main goal of classification algorithm is to find an optimal decision function which minimizes a loss function defined according to the task at hand. For most classification problems the loss function is built around counting the misclassifications.

5.2 Classification Algorithms used in the Project

A total of six different models are built in this project applying open source binary classification algorithms. The six different algorithms used to build these models are:

- Decision Trees
- K-Nearest Neighbours
- Linear Discriminant analysis
- Logistic regression

- Random forest
- Support Vector Machines

while building the models, the algorithmic functions used are from the Scikit-learn package in python. While building each model In the first step, the algorithm is applied to the full set of features present in the unbalanced(original)dataset. In the second step, the algorithm is applied to the over-sampled balanced dataset followed by subsequent steps of applying different methods of parameter tuning with cross validation to increase the model performance. The performance of each model used in the project has been evaluated using different evaluation metrics. The method of computing these evaluation metrics is based on first computing the confusion matrix of binary classification.

6 Evaluation Metrics

Generally in binary classification problems the target class is also referred to as the "positive" class and the other class as "negative" class. Building a confusion matrix is then looked upon as building a 2 2 matrix where the elements in the diagonal are correctly classified instances from the data. These diagonal elements are also referred to as True Positives: TP (correctly classified as positive class) and True Negatives:TN (correctly classified as negative class). The off-diagonal represent instances of false classification. The diagonal elements are referred to as False Positives: FP (incorrectly classified as positive class) and False Negatives: FN (incorrectly classified as negative class).[7] The figure below shows the representation of the confusion matrix as a Table.

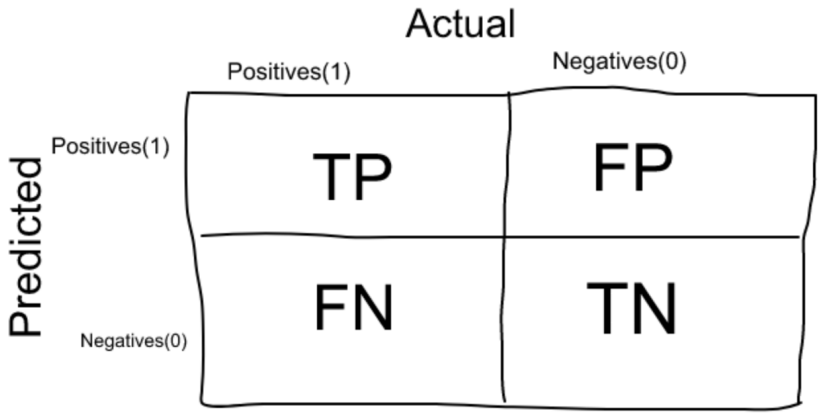


Figure 7: Confusion matrix

In this project our target class is the class "Poor" and is is taken to be the positive class and "Non-Poor" class the negative class. Using the elements of the confusion

matrix we can then derive several performance Metrics that help us evaluate the classification Performance of our models. The metrics used to evaluate the performance of the models used in this project are as follows:

- **Accuracy:** Accuracy in classification problems simply is ratio of correct predictions made over all predictions made. Mathematically, accuracy is defined by:

$$Accuracy = \frac{TP + TN}{TP + FP + FN + TN} \tag{14}$$

  However, using only accuracy to compare model performance will be misleading when dealing with imbalanced class distribution. in the dataset. In such cases even when the model's accuracy is a high value, the model will not be a good predictor of the minority class of the imbalanced dataset. [1]

- **Precision:** Precision gives us the ratio of the true positive value against all the positive predictive values. In simpler terms precision answers the question, "Of all the identified positive instances, how many identifications were correct?"[]. Mathematically precision is defined by:

$$Precision = \frac{TP}{TP + FP} \tag{15}$$

  The formula shows that As false positive instances increases the precision decreases. A classifier may achieve high recall by predicting many false positives. As such, precision a useful counterbalance to recall. [1]

- **Recall:** Recall provides a measure of the ratio of correctly classified samples to the total number of samples in that particular class. Recall is particularly useful in cases of imbalanced classes. [1]

- **F1 score:** The F1 score is the harmonic mean of the precision and recall. Mathematically represented by:

$$F1 = \frac{2TP}{2TP + FP + FN} \tag{16}$$

$$F1 \in [0, 1] \tag{17}$$

  If one of the number between precision and recall is really small, the F1 score is more closer to the smaller number. [1]

- **ROC curve:** The receiver operating characteristic (ROC) curve curve is constructed by plotting the recall or the true positive rate against the false positive rate. For perfect classification, the ROC curve will consists of a false positive rate of 0 and a true positive rate of 1 leading to an area of 1 under the ROC curve. [1]

7 Results

7.1 Classification Results using all Features and Unbalanced Dataset

As shown in figure 8 below, the logistic regression model performs the best with 97 % accuracy. Its performance against other metrics indicates that while the algorithm correctly classifies the same percentage of overall households, it is substantially better at identifying "Poor" households, with a recall of 0.94.

Algorithm	Accuracy	Precision	Recall	F1	Area under ROC
Logistic Regression	0.977	0.941	0.941	0.941	0.996
LDA	0.938	0.835	0.941	0.885	0.987
K-NN	0.852	0.742	0.636	0.685	0.915
Decision Trees	0.960	0.864	1.0	0.927	0.973
Random Forest	0.960	0.865	1.0	0.927	0.990
SVM	0.962	0.957	0.889	0.922	0.992

Figure 8: performance metrics result table of the different algorithms applied to the full set of features in the Unbalanced Dataset.

7.2 Classification using all Features and Balanced Dataset

As the case with the unbalanced dataset we can see from figure 9 below that the logistic regression model performs the best with 96 % accuracy. Its performance against other metrics indicates that while the algorithm correctly classifies the same percentage of overall households, it is substantially better at identifying "Poor" households, with a recall of 0.98.

Algorithm	Accuracy	Precision	Recall	F1	Area under ROC
Logistic Regression	0.966	0.893	0.98	0.935	0.997
LDA	0.882	0.685	0.992	0.811	0.99
K-NN	0.852	0.742	0.636	0.685	0.915
Decision Trees	0.960	0.864	1.0	0.927	0.973
Random Forest	0.959	0.863	1.0	0.926	0.997
SVM	0.966	0.896	0.979	0.936	0.996

Figure 9: performance metrics result table of the Cross validated models applied to the full set of features in the Balanced Dataset.

7.3 Predicted Poverty

As seen in figure 10 below when we compare the actual national poverty rate calculated using the household weight against the poverty rate predicted using the model predictions, we see that the difference is least for the Support Vector Machine(SVM) classifier i.e the predictions made by the SVM algorithm were better at predicting poverty at an national level.

Algorithm	Actual Poverty rate	Predicted Poverty rate	Difference
Logistic Regression	25.48%	27.08%	-1.60
LDA	25.48%	29.29%	-3.81
K-NN	25.48%	20.38%	5.10
Decision Trees	25.48%	26.47%	-0.99
Random Forest	25.48%	26.61%	-1.13
SVM	25.48%	26.06%	-0.58

Figure 10: Difference between actual poverty rate and predicted poverty rate by different classification algorithms.

8 References

References

[1] P. B. "Casey A. Fitzpatrick" and O. Dupriez", *Machine Learning for Poverty Prediction*. 2018-06-13.

[2] N. PLULIKOVA, *POVERTY ANALYSIS USING MACHINE LEARNING METHODS*. 2016.

[3] H. Jaeger, *Principles of Statistical Modelling lecture notes spring 2019*. 2019.

[4] S. S. 'Nor S Sani' and M. A. Rahman', *Machine Learning Approach for Bottom 40 Percent Households (B40) Poverty Classification*. 2018-09-1.

[5] K. P. Murphy, *Machine Learning A Probabilistic Perspective*. 2012.

[6] H. Jaeger, *Machine Learning lecture notes spring 2019*. 2019.

[7] J. F. "Trevor hastie", "Robert Tibshirani", *The Elements of Statistical Learning: Data Mining, Inference and Prediction*. 2001.

A Code

A.1 data parsing and visualization

```
"""
Created on Fri Apr  5 13:57:38 2019

@author: atit
"""

import os
import sys
import json

import numpy as np
import pandas as pd
from pandas.io.stata import StataReader

from matplotlib import pyplot as plt
from IPython.display import display
import seaborn as sns
sns.set()

from sklearn.model_selection import train_test_split



filepath_household = "nlss_3.dta"
filepath_individual = "poverty_nlss.dta"

'''function for loading stata file'''

def load_stata(filepath,indexcol, drop_minornans=False):

    data = pd.read_stata(filepath, convert_categoricals=False).set_index(indexcol)


# convert the categorical variables into
# the category type
    with StataReader(filepath) as reader:
        reader.value_labels()


        mapping = {col: reader.value_label_dict[t] for col, t in
                   zip(reader.varlist, reader.lbllist)
                   if t in reader.value_label_dict}
```

```
# drop records with name labels.

        data.replace(mapping, inplace=True)
       # convert the categorical variables into
        # the category type
        cat_list = []
        for c in data.columns:
            if c in mapping:
                cat_list.append(c)
                data[c] = data[c].astype('category')
        data['poor'] = data['poor'].astype('category')
        data.drop('gap',axis=1,inplace=True)
        data.drop('gapsq',axis=1,inplace=True)
        data.drop('food_poor',axis=1,inplace=True)
        data.drop('inc_poor',axis=1,inplace=True)

        data.drop('Date',axis=1,inplace=True)


        for i in data.columns:
            if data[i].dtype == "object":
                data.drop(i, axis=1, inplace=True)
        # drop records with only a few nans

        if drop_minornans:
            nan_counts = (data.applymap(pd.isnull)
                          .sum(axis=0)
                          .sort_values(ascending=False))
            nan_cols = nan_counts[(nan_counts > 0) & (nan_counts < 10)].index.values
            data = data.dropna(subset=nan_cols)


        questions = reader.variable_labels()




    return data, questions, cat_list


'''load data'''
```

```
nep_hhold , nep_hhold_q, c_list = load_stata(filepath_household,['xhnum'],drop_minornans=Tru

nep_hhold.columns[200:242]
pd.DataFrame.from_dict(nep_hhold_q, orient='index')[240:300]
print (c_list)
s = 'Nepal household data has {:,} rows and {:,} columns'
print(s.format(nep_hhold.shape[0], nep_hhold.shape[1]))
s = 'Percent poor: {:0.1%} \tPercent non-poor: {:0.1%}'
per = nep_hhold.poor.value_counts(normalize=True)
print(s.format(per[1],per[0]))
print(nep_hhold.head())

print(nep_hhold_q)
nep_hhold.dtypes[0:50]
nep_hhold.Eastern
pd.DataFrame.from_dict(nep_hhold_q, orient='index')[40:100]

g = sns.catplot(y='poor',kind="count", orient="h",palette="pastel", data=nep_hhold)

#Demographics




#Expoloring Education
#highest level of household head education
#only look at interesting feature in 1
#answered yes to below the literaciy rate:
sns.catplot(x="poor", hue="head_educ_1", kind="count",palette="pastel", edgecolor=".6",data=
# Answered true to question literate or below five years of primary education:
sns.catplot(x="poor", hue="head_educ_2", kind="count",palette="pastel", edgecolor=".6",data=
# answered yes to completed 7 years of education
sns.catplot(x="head_educ_5", hue="poor", kind="count",palette="pastel", edgecolor=".6",data=
#answered yes to completed highschool
sns.catplot(x="head_educ_5", hue="poor", kind="count",palette="pastel", edgecolor=".6",data=
print(nep_hhold.head_edu)

# minority groups
#muslim
sns.catplot(x="poor", hue="head_ethgroup_10", kind="count",palette="pastel", edgecolor=".6",
#Janajati Hill
sns.catplot(x="poor", hue="head_ethgroup_8", kind="count",palette="pastel", edgecolor=".6",d
#Janajati Terai
sns.catplot(x="poor", hue="poor", kind="head_ethgroup",palette="pastel", edgecolor=".6",data
```

```
sns.catplot(y="head_ethgroup", kind="count",palette="pastel", edgecolor=".6",data=nep_hhold)
sns.catplot(x="wi30_hosp", kind="count",palette="pastel", edgecolor=".6",data=nep_hhold,)
sns.catplot(y="max_educ_fem", kind="count",palette="pastel", edgecolor=".6",data=nep_hhold)



sns.distplot(nep_hhold.head_age, hist= False)

sns.distplot(nep_hhold.hhsize,)




#load the individual level data
nep_indv, nep_indv_q , c_idv_list= load_stata(filepath_individual,['xhnum','v07_idc'])

'''s = 'Nepal household data has {:,} rows and {:,} columns'
print(s.format(nep_indv.shape[0], nep_indv.shape[1]))
s = 'Percent poor: {:0.1%} \tPercent non-poor: {:0.1%}'
per = nep_indv.poor.value_counts(normalize=True)
print(s.format(per[1],per[0]))
print(nep_indv.head())'''


###changing target variable poor into boolean
nep_hhold.poor = (nep_hhold.poor == 1)

nep_hhold.wt_hh


'''Derive some features from individual data if needed'''


#looking into features in individual data:

'''pd.DataFrame.from_dict(nep_indv_q, orient='index')[150:200]'''

'''Feature Inspection and Reduction'''

nep_hhold = pd.get_dummies(nep_hhold, drop_first=True, dummy_na=True, prefix_sep='__' )
print("Nepal household shape with dummy variables added", nep_hhold.shape)

# remove columns with only one unique value removing survey answers that remain same
nep_hhold = nep_hhold.loc[:, nep_hhold.nunique(axis=0) > 1]
```

```
print("Nepal household shape with constant columns dropped", nep_hhold.shape)

# remove duplicate columns - questions with identical ansewrs for each household
def drop_duplicate_columns(df, ignore=[], inplace=False):
    if not inplace:
        df = df.copy()

    # pairwise correlations
    corr = df.corr()
    corr[corr.columns] = np.triu(corr, k=1)
    corr = corr.stack()

    # for any perfectly correlated variables, drop one of them
    for ix, r in corr[(corr == 1)].to_frame().iterrows():
        first, second = ix

        if second in df.columns and second not in ignore:
            df.drop(second, inplace=True, axis=1)

    if not inplace:
        return df

drop_duplicate_columns(nep_hhold, ignore=['wt_ind', 'wt_hh'], inplace=True)
print("Nepal household shape with duplicate columns dropped", nep_hhold.shape)

### Drop rows wint missing values and NA
nep_hhold = nep_hhold.dropna()
print("Nepal household shape with  NA rows dropped", nep_hhold.shape)

#Explore Data


def plot_numeric_hist(df,
                      col,
                      x_label,
                      y_label='Percentage of Households',
                      target='poor',
                      integer_ticks=True,
                      ax=None):
    if ax is None:
        ax = plt.gca()

    df.groupby(df[target])[col].plot.hist(bins=np.arange(0, df[col].max()) - 1,
                                          alpha=0.5,
                                          normed=True,
                                          ax=ax)
```

```
    ax.set_xlim([0,df[col].max()])
    if integer_ticks:
        ax.set_xticks(np.arange(0,df[col].max()) + 0.5)
        ax.set_xticklabels(np.arange(0,df[col].max()+1, dtype=int))
        ax.xaxis.grid(False)
    ax.set_xlabel(x_label)
    ax.set_ylabel(y_label)
    ax.legend(title='poor')
pd.DataFrame.from_dict(nep_hhold_q, orient='index')[200:260]

#Distribution of childeren
nep_hhold['nchild'] = nep_hhold.nkid06 + nep_hhold.nkid715
plot_numeric_hist(nep_hhold,'nchild','Number of children')



#Distribution of adults
nep_hhold['nadult'] = nep_hhold.nadultm + nep_hhold.nadultf
plot_numeric_hist(nep_hhold,'nadult','Number of Adult')

#employment
plot_numeric_hist(nep_hhold,'nemp','Number of employed adults')
plot_numeric_hist(nep_hhold,'nemp_male','Number of employed adult male')
plot_numeric_hist(nep_hhold,'nemp_fem','Number of employed adult female')
sns.catplot(x="head_occup_2", kind="bar",hue='poor', edgecolor=".6",data=nep_hhold)


#Migration
sns.catplot(x="has_formig", kind="box",hue='poor',data=nep_hhold)

#Education









#2d distributions
sns.jointplot(x='', y='hhsize',kind='kde',color='k', data=nep_hhold)
```

```
sns.kdeplot(nep_hhold.nemp_male, nep_hhold.nemp_fem)
```

```
#share of childeren vs household size
nep_hhold['child'] = nep_hhold.s_kid06 + nep_hhold.s_kid715
sns.catplot(x="hhsizeT",y='child', hue='poor',palette="pastel", kind='point',markers=["^", '
sns.catplot(x="hhsize",y='child', hue='poor',palette="pastel", kind='point',markers=["^", "c

#share of adults VS household size
nep_hhold['adult'] = nep_hhold.s_adultm + nep_hhold.s_adultf
sns.catplot(x="hhsizeT",y='adult', hue='poor',palette="pastel", kind='point',markers=["^", '
sns.catplot(x="hhsize",y='adult', hue='poor',palette="pastel", kind='point',markers=["^", "c
#share of elders VS household size
sns.catplot(x="hhsizeT",y='s_elderly', hue='poor',palette="pastel", kind='point',markers=["
sns.catplot(x="hhsize",y='s_elderly', hue='poor',palette="pastel", kind='point',markers=["^'
```

```
plot_numeric_hist(nep_hhold,'hhsizeT','Capped Household size')
```

```
# Compare poor vs non-poor numeric features
#   We have 8 numeric features, so we make a 2x3 grid to plot them
fig, axes = plt.subplots(2, 4, figsize=(20,10))
plot_numeric_hist(nep_hhold,
                  'hhsize',
                  'Household Size',
                  ax=axes[0][0])
plot_numeric_hist(nep_hhold,
                  'nkid06',
                  'Number of children 6 years old or under',
                  ax=axes[0][1])
plot_numeric_hist(nep_hhold,
                  'nfooditm_7',
                  'Number of food items consumed in a week',
                  ax=axes[0][2])
plot_numeric_hist(nep_hhold,
                  'depratio3',
                  'Dependent ratio',
```

```
                  ax=axes[0][3])
plot_numeric_hist(nep_hhold,
                  'nadultf',
                  'Number of adult Females',
                  ax=axes[1][0])
plot_numeric_hist(nep_hhold,
                  'nelderly',
                  'Number of elders age 65+',
                  ax=axes[1][1])
plot_numeric_hist(nep_hhold,
                  'nemp_fem',
                  'Number of female household members employed',
                  ax=axes[1][2])
plot_numeric_hist(nep_hhold,
                  'nemp20',
                  'Number of household members employed more than 20 hrs',
                  ax=axes[1][3])

plt.show()

pd.DataFrame.from_dict(nep_hhold_q, orient='index')[100:150]

# Filter weekly consumption and group by poor/non-poor
consumption_columns = [x for x in nep_hhold.columns if x.startswith('sh_') and x.endswith('_
consumption = (nep_hhold.groupby('poor')[consumption_columns].sum().T)

consumption.columns = ['Non_poor', 'Poor']
consumption['total'] = consumption.sum(axis=1)
consumption['percent'] = consumption.total / nep_hhold.shape[0]

# Match up the consumable names for readability
get_consumable_name = lambda x: nep_hhold_q[x.split('__')[0]]
consumption.index = consumption.index.map(get_consumable_name)


consumption['difference'] = (consumption.Non_poor - consumption.Poor) / consumption.total
display(consumption.sort_values('difference', ascending=False))

# Plot  share of weekly consumption as % difference in poor and non poor household
(consumption.difference.sort_values(ascending=False).sort_values(ascending=True).plot.barh()



# Filter monthly consumption and group by poor/non-poor
consumption_columns = [x for x in nep_hhold.columns if x.startswith('sh_') and x.endswith('_
consumption = (nep_hhold.groupby('poor')[consumption_columns].mean().T)
```

```
consumption.columns = ['Non_poor', 'Poor']
consumption['total'] = consumption.sum(axis=1)
consumption['percent'] = consumption.total / nep_hhold.shape[0]
consumption['difference'] = (consumption.Non_poor - consumption.Poor)
display(consumption.sort_values('difference', ascending=False))

# Match up the consumable names for readability
get_consumable_name = lambda x: nep_hhold_q[x.split('__')[0]]
consumption.index = consumption.index.map(get_consumable_name)

consumption['difference'] = (consumption.Non_poor - consumption.Poor) / consumption.total
display(consumption.sort_values('total', ascending=False))

# Plot  share of weekly consumption as % difference in poor and non poor household
dd= pd.DataFrame(consumption.sort_values('difference', ascending=False))


(consumption.difference.sort_values(ascending=True).sort_values(ascending=True).plot.barh())
(consumption.Non_poor.sort_values(ascending=True).sort_values(ascending=True).plot.bar())




# Train and Test Split, with 25% data in test
print(nep_hhold.shape)
nep_train, nep_test = train_test_split(nep_hhold, test_size=0.25,random_state=1443,stratify=


#save the test and train data to files

nep_train.to_pickle("nepal_poverty_train.pkl")
nep_test.to_pickle("nepal_poverty_test.pkl")
with open("nepal_indv_questions.json", 'w') as fp:
    json.dump(nep_indv_q, fp)
with open("nepal_pov_questions.json", 'w') as fp:
    json.dump(nep_hhold_q, fp)
```

A.2 Functions built for project

```
    def clip_yprob(y_prob):
    """Clip yprob to avoid 0 or 1 values. Fixes bug in log_loss calculation
    that results in returning nan."""
    eps = 1e-15
    y_prob = np.array([x if x <= 1-eps else 1-eps for x in y_prob])
    y_prob = np.array([x if x >= eps else eps for x in y_prob])
```

```
    return y_prob

def split_features_labels_weights(path,weights=['wt_ind', 'wt_hh'],weights_col=['wt_ind'],la
    data = pd.read_pickle(path)
    return (data.drop(weights + label_col, axis=1),
            data[label_col],
            data[weights_col])

''' standardize the features (substracting mean from the columns)'''

def standardize(df, numeric_only=True):
    if numeric_only is True:
    # find non-boolean columns
        cols = df.loc[:, df.dtypes != 'uint8'].columns
    else:
        cols = df.columns
    for field in cols:
        m, s = df[field].mean(), df[field].std()
        # account for constant columns
        if np.all(df[field] - m != 0):
            df.loc[:, field] = (df[field] - m) / s

    return df



def load_data(path, selected_columns=None, ravel=True, standardize_columns='numeric'):
    X, y, w = split_features_labels_weights(path)
    if selected_columns is not None:
        X = X[[col for col in X.columns.values if col in selected_columns]]
    if standardize_columns == 'numeric':
        standardize(X)
    elif standardize_columns == 'all':
        standardize(X, numeric_only=False)
    if ravel is True:
        y = np.ravel(y)
        w = np.ravel(w)

    return (X, y, w)



def get_vif(X):
    vi_factors = [variance_inflation_factor(X.values, i)
                              for i in range(X.shape[1])]
```

```
    return pd.Series(vi_factors,
                     index=X.columns,
                     name='variance_inflaction_factor')


def get_coefs_df(X, coefs, index=None, sort=True):
    coefs_df = pd.DataFrame(np.std(X, 0) * coefs)
    coefs_df.columns = ["coef_std"]
    coefs_df['coef'] = coefs
    coefs_df['abs'] = coefs_df.coef_std.apply(abs)
    if index is not None:
        coefs_df.index = index
    if sort:
        coefs_df = coefs_df.sort_values('abs', ascending=False)

    return coefs_df

def predict_poverty_rate(train_path, test_path, model,
                         standardize_columns='numeric',
                         ravel=True,
                         selected_columns=None,
                         show=True,
                         return_values= True):
    # Recombine the entire dataset to get the actual poverty rate
    X_train, y_train, w_train = load_data(TRAIN_PATH,
                                          standardize_columns=standardize_columns,
                                          ravel=ravel,
                                          selected_columns=selected_columns)
    X_test, y_test, w_test = load_data(TEST_PATH,
                                       standardize_columns=standardize_columns,
                                       ravel=ravel,
                                       selected_columns=selected_columns)
    pov_rate = pd.DataFrame(np.vstack((np.vstack((y_train, w_train)).T,
                                       np.vstack((y_test, w_test)).T)),
                            columns=['poor', 'wta_pop'])
    pov_rate_actual = (pov_rate.wta_pop * pov_rate.poor).sum() / pov_rate.wta_pop.sum()

    # Make predictions on entire dataset to get the predicted poverty rate
    pov_rate['pred'] = model.predict(np.concatenate((X_train.as_matrix(), X_test.as_matrix()
    pov_rate_pred = (pov_rate.wta_pop * pov_rate.pred).sum() / pov_rate.wta_pop.sum()

    if show == True:
        print("Actual poverty rate: {:0.2%} ".format(pov_rate_actual))
        print("Predicted poverty rate: {:0.2%} ".format(pov_rate_pred))
    if return_values:
        return pov_rate_actual, pov_rate_pred
```

```
    else:
        return
def calculate_metrics(y_test, y_pred, y_prob=None, sample_weights=None):
    """Cacluate model performance metrics"""

    # Dictionary of metrics to calculate
    metrics = {}
    metrics['confusion_matrix']  = confusion_matrix(y_test, y_pred, sample_weight=sample_wei
    metrics['roc_auc']           = None
    metrics['accuracy']          = accuracy_score(y_test, y_pred, sample_weight=sample_weigh
    metrics['precision']         = precision_score(y_test, y_pred, sample_weight=sample_weig
    metrics['recall']            = recall_score(y_test, y_pred, sample_weight=sample_weights
    metrics['f1']                = f1_score(y_test, y_pred, sample_weight=sample_weights)
    metrics['cohen_kappa']       = cohen_kappa_score(y_test, y_pred)
    metrics['cross_entropy']     = None
    metrics['fpr']               = None
    metrics['tpr']               = None
    metrics['auc']               = None

    # Populate metrics that require y_prob
    if y_prob is not None:
        clip_yprob(y_prob)
        metrics['cross_entropy']     = log_loss(y_test,
                                                clip_yprob(y_prob),
                                                sample_weight=sample_weights)
        metrics['roc_auc']           = roc_auc_score(y_test,
                                                     y_prob,
                                                     sample_weight=sample_weights)

        fpr, tpr, _ = roc_curve(y_test,
                                y_prob,
                                sample_weight=sample_weights)
        metrics['fpr']               = fpr
        metrics['tpr']               = tpr
        metrics['auc']               = auc(fpr, tpr, reorder=True)

    return metrics

def conf_mat(metrics):
    array = metrics['confusion_matrix']
    classes=['poor','non-poor']
    df_cm = pd.DataFrame(array, index = [i for i in classes],
                  columns = [i for i in classes])
    plt.figure(figsize = (10,7))
    sn.heatmap(df_cm, annot=True,cmap="Blues")
```

```
def metrics_table(metrics,model_name):
    workbook = xlsxwriter.Workbook(model_name+'.xlsx')
    worksheet = workbook.add_worksheet()
    metrics.pop('confusion_matrix')
    metrics.pop('fpr')
    metrics.pop('tpr')
    metrics.pop('cohen_kappa')
    metrics.pop('cross_entropy')
    metrics.pop('roc_auc')

    head = []
    for key in metrics:
        head.append(key)
    row=0
    column = 0
    for item in head:
        worksheet.write(row,column,item)
        column +=1
    row = 1
    column = 0
    for key in metrics:
        worksheet.write(row, column, metrics[key])
        column +=1

    workbook.close()

def pov_table(pred,model_name):
    workbook = xlsxwriter.Workbook(model_name+'pov.xlsx')
    worksheet = workbook.add_worksheet()
    head = ['Actual','Predicted']
    row=0
    column = 0
    for item in head:
        worksheet.write(row,column,item)
        column +=1
    row = 1
    column = 0
    for i in range(len(head)):
        worksheet.write(row, column, pred[i])
        column +=1

    workbook.close()

def subsample(X, y, w, stratify=True, seed=566):
    n_samples = int(SUBSAMPLE * X.shape[0])
```

```
    rng = np.random.RandomState(seed)

    if stratify:
        y_rate = y.mean()
        n_true = int(n_samples * y_rate)
        n_false = n_samples - n_true

        true_idx = rng.choice(np.where(y)[0], n_true, replace=False)
        false_idx = rng.choice(np.where(~y)[0], n_false, replace=False)

        sample_idx = np.union1d(true_idx, false_idx)
    else:
        sample_idx = rng.choice(np.arange(X.shape[0]), n_samples, replace=False)

    return X.iloc[sample_idx, :], y[sample_idx], w[sample_idx]
```

A.3 logistic Regression

```
# -*- coding: utf-8 -*-
"""
Created on Tue Apr  9 21:25:19 2019

@author: atit
"""
# Load and transform the training data
X_train, y_train, w_train = load_data(TRAIN_PATH)
# Load and transform the test set
X_test, y_test, w_test = load_data(TEST_PATH)


#### model with weights
model.fit(X_train, y_train, sample_weight=w_train)
score_w = model.score(X_train, y_train, sample_weight=w_train)
coefs_w = get_coefs_df(X_train, model.coef_[0])['abs']

# Run the model
y_pred_w = model.predict(X_test)
y_prob_w = model.predict_proba(X_test)[:,1]

# prediction and Metrics
pred_w = predict_poverty_rate(TRAIN_PATH,TEST_PATH,model)
metrics_lr_w = calculate_metrics (y_test, y_pred_w, y_prob_w)
```

```
##results
conf_mat(metrics_lr_w)
metrics_table(metrics_lr_w,'logisticreg_weights')
pov_table(pred_w,'logisticreg_weights')


### model with cross validation

# Load and transform the training data
X_train, y_train, w_train = load_data(TRAIN_PATH)

# Fit the model
model = LogisticRegressionCV(Cs=10, cv=5, verbose=1)
model.fit(X_train, y_train, sample_weight=w_train)

# Get an initial score
score = model.score(X_train, y_train, sample_weight=w_train)

coefs = get_coefs_df(X_train, model.coef_[0])

# Display best parameters
best_params = model.C_[0]

# Run the model
y_pred_cv = model.predict(X_test)
y_prob_cv = model.predict_proba(X_test)[:,1]


#prediction and Metrics
pred_cv = predict_poverty_rate(TRAIN_PATH,TEST_PATH,model)
best_model_metrics= calculate_metrics (y_test, y_pred_cv, y_prob_cv, w_test)


##results
conf_mat(best_model_metrics)
metrics_table(best_model_metrics,'logisticreg_weights')
pov_table(pred_cv,'logisticreg_weights')

###class balance
# Load and transform the training data
X_train, y_train, w_train = load_data(TRAIN_PATH)
cols = X_train.columns
X_train.shape

# Apply oversampling with SMOTE
X_train, y_train = SMOTE().fit_sample(X_train, y_train)
```

```
print("X shape after oversampling: ", X_train.shape)

# Fit the model
model = LogisticRegression()
model.fit(X_train, y_train)

# Get an initial score
score = model.score(X_train, y_train)
print("In-sample score: {:0.2%}".format(score))

# Store coefficients
coefs = get_coefs_df(X_train, model.coef_[0], index=cols)

# Load the test set
X_test, y_test, w_test = load_data(TEST_PATH)

# Run the model
y_pred = model.predict(X_test)
y_prob = model.predict_proba(X_test)[:,1]

metrics_ov= calculate_metrics (y_test, y_pred, y_prob, w_test)
pred_ov =  predict_poverty_rate(TRAIN_PATH,TEST_PATH,model)

##results
conf_mat(metrics_ov)
metrics_table(metrics_ov,'logisticreg_weights')
pov_table(pred_ov,'logisticreg_weights')
```

A.4 Linear Discriminant Analysis

```
    #!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Fri Apr 12 11:46:37 2019

@author: atit
"""

import os
import sys
import json
import xlsxwriter

import numpy as np
import pandas as pd
```

```
from matplotlib import pyplot as plt
from IPython.display import display
import seaborn as sns
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

from sklearn.model_selection import GridSearchCV
from imblearn.over_sampling import SMOTE


from sklearn.metrics import (
    confusion_matrix,
    log_loss,
    roc_auc_score,
    accuracy_score,
    precision_score
)

from sklearn.metrics import (
    recall_score,
    f1_score,
    cohen_kappa_score,
    roc_curve,
    auc
)

TRAIN_PATH = "nepal_poverty_train.pkl"
QUESTION_PATH = "nepal_pov_questions.json"
TEST_PATH = "nepal_poverty_test.pkl"


ALGORITHM_NAME = 'lda'
COUNTRY = 'Nepal'

with open(QUESTION_PATH, 'r') as fp:
    questions = json.load(fp)

def split_features_labels_weights(path,weights=['wt_ind', 'wt_hh'],weights_col=['wt_ind'],la
    data = pd.read_pickle(path)
    return (data.drop(weights + label_col, axis=1),
            data[label_col],
            data[weights_col])

''' standardize the features (substracting mean from the columns)'''

def standardize(df, numeric_only=True):
    if numeric_only is True:
```

```
    # find non-boolean columns
        cols = df.loc[:, df.dtypes != 'uint8'].columns
    else:
        cols = df.columns
    for field in cols:
        m, s = df[field].mean(), df[field].std()
        # account for constant columns
        if np.all(df[field] - m != 0):
            df.loc[:, field] = (df[field] - m) / s

    return df

def get_coefs_df(X, coefs, index=None, sort=True):
    coefs_df = pd.DataFrame(np.std(X, 0) * coefs)
    coefs_df.columns = ["coef_std"]
    coefs_df['coef'] = coefs
    coefs_df['abs'] = coefs_df.coef_std.apply(abs)
    if index is not None:
        coefs_df.index = index
    if sort:
        coefs_df = coefs_df.sort_values('abs', ascending=False)

    return coefs_df

def clip_yprob(y_prob):
    """Clip yprob to avoid 0 or 1 values. Fixes bug in log_loss calculation
    that results in returning nan."""
    eps = 1e-15
    y_prob = np.array([x if x <= 1-eps else 1-eps for x in y_prob])
    y_prob = np.array([x if x >= eps else eps for x in y_prob])
    return y_prob

def predict_poverty_rate(train_path, test_path, model,
                         standardize_columns='numeric',
                         ravel=True,
                         selected_columns=None,
                         show=True,
                         return_values=True):
    # Recombine the entire dataset to get the actual poverty rate
    X_train, y_train, w_train = load_data(TRAIN_PATH,
                                          standardize_columns=standardize_columns,
                                          ravel=ravel,
                                          selected_columns=selected_columns)
    X_test, y_test, w_test = load_data(TEST_PATH,
                                       standardize_columns=standardize_columns,
                                       ravel=ravel,
```

```
                                        selected_columns=selected_columns)
    pov_rate = pd.DataFrame(np.vstack((np.vstack((y_train, w_train)).T,
                                        np.vstack((y_test, w_test)).T)),
                            columns=['poor', 'wta_pop'])
    pov_rate_actual = (pov_rate.wta_pop * pov_rate.poor).sum() / pov_rate.wta_pop.sum()

    # Make predictions on entire dataset to get the predicted poverty rate
    pov_rate['pred'] = model.predict(np.concatenate((X_train.as_matrix(), X_test.as_matrix()
    pov_rate_pred = (pov_rate.wta_pop * pov_rate.pred).sum() / pov_rate.wta_pop.sum()

    if show == True:
        print("Actual poverty rate: {:0.2%} ".format(pov_rate_actual))
        print("Predicted poverty rate: {:0.2%} ".format(pov_rate_pred))
    if return_values:
        return pov_rate_actual, pov_rate_pred
    else:
        return

def calculate_metrics(y_test, y_pred, y_prob=None, sample_weights=None):
    """Cacluate model performance metrics"""

    # Dictionary of metrics to calculate
    metrics = {}
    metrics['confusion_matrix']  = confusion_matrix(y_test, y_pred, sample_weight=sample_wei
    metrics['roc_auc']           = None
    metrics['accuracy']          = accuracy_score(y_test, y_pred, sample_weight=sample_weigh
    metrics['precision']         = precision_score(y_test, y_pred, sample_weight=sample_weig
    metrics['recall']            = recall_score(y_test, y_pred, sample_weight=sample_weights
    metrics['f1']                = f1_score(y_test, y_pred, sample_weight=sample_weights)
    metrics['cohen_kappa']       = cohen_kappa_score(y_test, y_pred)
    metrics['cross_entropy']     = None
    metrics['fpr']               = None
    metrics['tpr']               = None
    metrics['auc']               = None

    # Populate metrics that require y_prob
    if y_prob is not None:
        clip_yprob(y_prob)
        metrics['cross_entropy']     = log_loss(y_test,
                                                clip_yprob(y_prob),
                                                sample_weight=sample_weights)
        metrics['roc_auc']           = roc_auc_score(y_test,
                                                     y_prob,
                                                     sample_weight=sample_weights)

        fpr, tpr, _ = roc_curve(y_test,
```

```
                                y_prob,
                                sample_weight=sample_weights)
        metrics['fpr']              = fpr
        metrics['tpr']              = tpr
        metrics['auc']              = auc(fpr, tpr, reorder=True)

    return metrics

def conf_mat(metrics):
    array = metrics['confusion_matrix']
    classes=['poor','non-poor']
    df_cm = pd.DataFrame(array, index = [i for i in classes],
                  columns = [i for i in classes])
    plt.figure(figsize = (10,7))
    sn.heatmap(df_cm, annot=True,cmap="Blues")

def metrics_table(metrics,model_name):
    workbook = xlsxwriter.Workbook(model_name+'.xlsx')
    worksheet = workbook.add_worksheet()
    metrics.pop('confusion_matrix')
    metrics.pop('fpr')
    metrics.pop('tpr')
    metrics.pop('cohen_kappa')
    metrics.pop('cross_entropy')
    metrics.pop('roc_auc')

    head = []
    for key in metrics:
        head.append(key)
    row=0
    column = 0
    for item in head:
        worksheet.write(row,column,item)
        column +=1
    row = 1
    column = 0
    for key in metrics:
        worksheet.write(row, column, metrics[key])
        column +=1

    workbook.close()

def pov_table(pred,model_name):
    workbook = xlsxwriter.Workbook(model_name+'pov.xlsx')
    worksheet = workbook.add_worksheet()
    head = ['Actual','Predicted']
```

```
    row=0
    column = 0
    for item in head:
        worksheet.write(row,column,item)
        column +=1
    row = 1
    column = 0
    for i in range(len(head)):
        worksheet.write(row, column, pred[i])
        column +=1

    workbook.close()



def load_data(path, selected_columns=None, ravel=True, standardize_columns='numeric'):
    X, y, w = split_features_labels_weights(path)
    if selected_columns is not None:
        X = X[[col for col in X.columns.values if col in selected_columns]]
    if standardize_columns == 'numeric':
        standardize(X)
    elif standardize_columns == 'all':
        standardize(X, numeric_only=False)
    if ravel is True:
        y = np.ravel(y)
        w = np.ravel(w)

    return (X, y, w)



# Load the train and test set

X_train, y_train, w_train = load_data(TRAIN_PATH)
X_test, y_test, w_test = load_data(TEST_PATH)

###LDA with all features:

# Fit the model
model = LinearDiscriminantAnalysis()
model.fit(X_train, y_train)

# Get an initial score
score = model.score(X_train, y_train, w_train)
coefs = get_coefs_df(X_train, model.coef_[0])
```

```
# Run the model
y_pred_full = model.predict(X_test)
y_prob_full = model.predict_proba(X_test)[:,1]


#metrics and Prediction
metrics_lda_full = calculate_metrics(y_test,y_pred_full,y_prob_full,w_test)
pov_lda_full = predict_poverty_rate(TRAIN_PATH,TEST_PATH,model)

#results
conf_mat(metrics_lda_full)
metrics_table(metrics_lda_full,'lda_full')
pov_table(pov_lda_full,'lda_full')


#Transform LDA RESULTS
X_lda = model.transform(X_train)

mask = (y_train == 1)
fig, axes = plt.subplots(1,2, figsize=(12,4))
axes[0].scatter(X_lda[mask], y_train[mask], color='b', marker='+', label='poor')
axes[0].scatter(X_lda[~mask], y_train[~mask], color='r', marker='o', label='non-poor')
axes[0].set_title('LDA Projected Data')
axes[0].set_xlabel('Transformed axis')
axes[0].set_ylabel('\'Poor\' Probability')
axes[0].legend()

sns.kdeplot(np.ravel(X_lda[mask]), color='b', ax=axes[1], label='poor')
sns.kdeplot(np.ravel(X_lda[~mask]), color='r', ax=axes[1], label='non-poor')
axes[1].set_title('LDA Projected Density')
axes[1].set_xlabel('Transformed axis')
axes[1].set_ylabel('Class Density')
axes[1].legend()
plt.show()


### class Balance Oversampling

# Load and transform the training data
X_train, y_train, w_train = load_data(TRAIN_PATH)
cols = X_train.columns

# Apply oversampling with SMOTE
```

```
X_train, y_train = SMOTE().fit_sample(X_train, y_train)
print("X shape after oversampling: ", X_train.shape)

# Fit the model
model = LinearDiscriminantAnalysis()
model.fit(X_train, y_train)


# Get an initial score
score = model.score(X_train, y_train)


# Store coefficients
coefs = get_coefs_df(X_train, model.coef_[0], index=cols)

# Load the test set
X_test, y_test, w_test = load_data(TEST_PATH)

# Run the model
y_pred_b = model.predict(X_test)
y_prob_b = model.predict_proba(X_test)[:,1]


metrics_lda_b = calculate_metrics(y_test,y_pred_b,y_prob_b,w_test)
pov_lda_b = predict_poverty_rate(TRAIN_PATH,TEST_PATH,model)


#results
conf_mat(metrics_lda_b)
metrics_table(metrics_lda_b,'lda_b')
pov_table(pov_lda_b,'lda_b')







###LDA with GRID SEARCH CV

# build the model
estimator = LinearDiscriminantAnalysis()
parameters = {'solver': ['svd']}

model = GridSearchCV(estimator, parameters, verbose=1, cv=5)
model.fit(X_train, y_train)
```

```
# Get an initial score
score = model.score(X_train, y_train)
coefs = get_coefs_df(X_train, model.best_estimator_.coef_[0])


# Run the model
y_pred_cv = model.predict(X_test)
y_prob_cv = model.predict_proba(X_test)[:,1]


metrics_lda_cv = calculate_metrics(y_test,y_pred_cv,y_prob_cv,w_test)
pov_lda_cv = predict_poverty_rate(TRAIN_PATH,TEST_PATH,model)


#results
conf_mat(metrics_lda_cv)
metrics_table(metrics_lda_cv,'lda_cv')
pov_table(pov_lda_cv,'lda_cv')
```

A.5 Decision trees

```
    ### Decision Trees with all features



# Load and transform the training data
X_train, y_train, w_train = load_data(TRAIN_PATH)

# Fit the model
model = DecisionTreeClassifier()
model.fit(X_train, y_train)

# Get an initial score
score = model.score(X_train, y_train, w_train)
print("In-sample score: {:0.2%}".format(score))
feat_imps = get_feat_imp_df(model.feature_importances_, index=X_train.columns)

# Load the test set
X_test, y_test, w_test = load_data(TEST_PATH)

# Run the model
y_pred_dt = model.predict(X_test)
y_prob_dt = model.predict_proba(X_test)[:,1]

metrics_dt = calculate_metrics(y_test,y_pred_dt,y_prob_dt,w_test)
```

```
print(metrics_dt)
print(feat_imps)



# tunig tree with depth, min leaves, minim feature at split, choosen by cv


# Load and transform the training data
# Load the test set
X_train, y_train, w_train = load_data(TRAIN_PATH)
cols = X_train.columns
X_train.shape

# Apply oversampling with SMOTE
X_train, y_train = SMOTE().fit_sample(X_train, y_train)
print("X shape after oversampling: ", X_train.shape)

# build the model
estimator = DecisionTreeClassifier()
parameters = {'max_depth': np.arange(1,16,5),
              'min_samples_split': np.arange(2,21,10),
              'min_samples_leaf': np.arange(1,46,20)
             }

model = GridSearchCV(estimator, parameters,  verbose=1, cv=5, n_jobs=-1)
model.fit(X_train, y_train)

# Get an initial score
score = model.score(X_train, y_train)
print("In-sample score: {:0.2%}".format(score))
print("Best model parameters:", model.best_params_)

# Load the test set
X_test, y_test, w_test = load_data(TEST_PATH)

# Run the model
y_pred_dtcv = model.predict(X_test)
y_prob_dtcv = model.predict_proba(X_test)[:,1]

metrics_dtcv = calculate_metrics(y_test,y_pred_dtcv,y_prob_dtcv,w_test)
pred_ov =  predict_poverty_rate(TRAIN_PATH,TEST_PATH,model)
print(metrics_dtcv)
print(feat_imps_cv)

best_model = model.best_estimator_
```

A.6 Random Forest

```
    ### Ramdom Forest with all features

X_train, y_train, w_train = load_data(TRAIN_PATH)

# Fit the model
model = RandomForestClassifier()
model.fit(X_train, y_train)

# Get an initial score
score = model.score(X_train, y_train, w_train)
print("In-sample score: {:0.2%}".format(score))
feat_imps_rf = get_feat_imp_df(model.feature_importances_, index=X_train.columns)

# Load the test set
X_test, y_test, w_test = load_data(TEST_PATH)

# Run the model
y_pred_rf = model.predict(X_test)
y_prob_rf = model.predict_proba(X_test)[:,1]

metrics_rf = calculate_metrics(y_test,y_pred_rf,y_prob_rf,w_test)
print(metrics_rf)
print(feat_imps_rf)

###  include sample weights.

# Load and transform the training data
X_train, y_train, w_train = load_data(TRAIN_PATH)

# Fit the model
model = RandomForestClassifier(n_estimators=100,
                               max_depth=20,
                               min_samples_leaf=5,
                               min_samples_split=5)
model.fit(X_train, y_train, sample_weight=w_train)

# Get an initial score
score = model.score(X_train, y_train, w_train)
print("In-sample score: {:0.2%}".format(score))
feat_imps_sw = get_feat_imp_df(model.feature_importances_, index=X_train.columns)

# Load the test set
X_test, y_test, w_test = load_data(TEST_PATH)
```

```
# Run the model
y_pred_sw = model.predict(X_test)
y_prob_sw = model.predict_proba(X_test)[:,1]

metrics_sw = calculate_metrics(y_test,y_pred_sw,y_prob_sw,w_test)
print(metrics_sw)
print(feat_imps_sw)


### tune parameters with cv

X_train, y_train, w_train = load_data(TRAIN_PATH)
cols = X_train.columns
X_train.shape

# Apply oversampling with SMOTE
X_train, y_train = SMOTE().fit_sample(X_train, y_train)
print("X shape after oversampling: ", X_train.shape)

# build the model


estimator = RandomForestClassifier()
parameters = {'n_estimators': [10, 50, 100],
              'max_depth': np.arange(1,16,5),
              'min_samples_split': np.arange(2,21,10),
              'min_samples_leaf': np.arange(1,46,20)
             }

model = GridSearchCV(estimator, parameters, verbose=1, cv=5, n_jobs=-1)
model.fit(X_train, y_train)

# Get an initial score
score = model.score(X_train, y_train)
print("In-sample score: {:0.2%}".format(score))
print("Best model parameters:", model.best_params_)

# Load the test set
X_test, y_test, w_test = load_data(TEST_PATH)

# Run the model
y_pred_cv = model.predict(X_test)
y_prob_cv = model.predict_proba(X_test)[:,1]

metrics_cv = calculate_metrics(y_test,y_pred_cv,y_prob_cv,w_test)
pred_ov =  predict_poverty_rate(TRAIN_PATH,TEST_PATH,model)
```

```
print(metrics_cv)
print(feat_imps_cv)

best_model = model.best_estimator_
```

A.7 Support Vector Machines

```
    X_train, y_train, w_train = load_data(TRAIN_PATH)
X_train, y_train, w_train = subsample(X_train, y_train, w_train)


# Fit the model
model = SVC(probability=True)
model.fit(X_train, y_train)

# Get an initial score
score = model.score(X_train, y_train, w_train)
print("In-sample score: {:0.2%}".format(score))

# Load the test set
X_test, y_test, w_test = load_data(TEST_PATH)

# Run the model
y_pred_svm = model.predict(X_test)
y_prob_svm = model.predict_proba(X_test)[:,1]

metrics_svm = calculate_metrics(y_test,y_pred_svm,y_prob_svm,w_test)
print(metrics_svm)

### tuning with cv




# Load and transform the training data
X_train, y_train, w_train = load_data(TRAIN_PATH)
X_train, y_train, w_train = subsample(X_train, y_train, w_train)

cols = X_train.columns

# Apply oversampling with SMOTE
X_train, y_train = SMOTE().fit_sample(X_train, y_train)
X_train = pd.DataFrame(X_train, columns=cols)
print("X shape after oversampling: ", X_train.shape)
```

```
# build the model
estimator = SVC(probability=True)
parameters = {'C': 10**np.linspace(-4,1,3),
              'kernel': ['rbf', 'linear']}
model = GridSearchCV(estimator, parameters, cv=3, verbose=3, n_jobs=4)
model.fit(X_train, y_train)

# Get an initial score
score = model.score(X_train, y_train)
print("In-sample score: {:0.2%}".format(score))
print("Best model parameters:", model.best_params_)

# Load the test set
X_test, y_test, w_test = load_data(TEST_PATH)

# Run the model
y_pred_svmcv = model.predict(X_test)
y_prob_svmcv = model.predict_proba(X_test)[:,1]

metrics_svmcv = calculate_metrics(y_test,y_pred_svmcv,y_prob_svmcv,w_test)
print(metrics_svmcv)
pred_ov =  predict_poverty_rate(TRAIN_PATH,TEST_PATH,model)
```

A.8 K-Nearest Neighbours

```
    ### KNN with all Features and n=15

X_train, y_train, w_train = load_data(TRAIN_PATH)
X_train, y_train, w_train = subsample(X_train, y_train, w_train)


# Fit the model
model = KNeighborsClassifier(n_neighbors=15)
model.fit(X_train, y_train)

# Get an initial score
score = model.score(X_train, y_train, w_train)
print("In-sample score: {:0.2%}".format(score))

# Load the test set
X_test, y_test, w_test = load_data(TEST_PATH)

# Run the model
y_pred_knn15 = model.predict(X_test)
```

```
y_prob_knn15 = model.predict_proba(X_test)[:,1]

metrics_knn15 = calculate_metrics(y_test,y_pred_knn15,y_prob_knn15,w_test)
print(metrics_knn15)



### 5 fold CV with full features


# build the model
estimator = KNeighborsClassifier()
parameters = {'n_neighbors': [3,5,7,9,11,13,15]}


model = GridSearchCV(estimator, parameters, verbose=4, cv=5)
model.fit(X_train, y_train)

# Get an initial score
score = model.score(X_train, y_train)
print("In-sample score: {:0.2%}".format(score))
print("Best model parameters:", model.best_params_)

# Load the test set
X_train, y_train, w_train = load_data(TRAIN_PATH)
cols = X_train.columns
X_train.shape

# Apply oversampling with SMOTE
X_train, y_train = SMOTE().fit_sample(X_train, y_train)
print("X shape after oversampling: ", X_train.shape)



# Run the model
y_pred_knncv = model.predict(X_test)
y_prob_knncv = model.predict_proba(X_test)[:,1]

pred_ov =  predict_poverty_rate(TRAIN_PATH,TEST_PATH,model)
metrics_knncv = calculate_metrics(y_test,y_pred_knncv,y_prob_knncv,w_test)
print(metrics_knncv)
```