Using Machine Learning to classify eligibility of anti-poverty programs
An analysis of the Peruvian Juntos program

Nikolas Anic (14-606-800), Sophia Bieri (15-727-373), Marco Funk (15-708-704)
Mauro Gübeli (16-704-801), Yannik Haller (12-918-645)

Semester Project in Machine Learning for Economic Analysis
University of Zurich

January 8, 2021

Introduction

This is the report of our analysis of the first semester project in the subject of Machine Learning for Economic Analysis. In this project, we are using several Machine Learning techniques to assess the accuracy of eligibility status prediction of Peruvian citizens to the Juntos program, a state-wide anti-poverty measure to financially support families of poor origin.

In order to be eligible for the program, families are required to fall below a certain income threshold. As income in developing countries is difficult to measure, administrators use a proxy-means test to assess financial capability. This test defines financial status according to a consumption-based measure, in which the variable of interest is determined through manual observations of household property.

In their analysis of social-welfare programs, the authors attempt to predict household consumption levels based on a simple OLS regression. They showed that, under the given specification, they were able to attain a Mean-Squared Error (MSE) of 0.1914699 on the training set.

As part of the semester project, we will apply numerous Machine Learning (ML) techniques to underscore the threshold set by the authors. Especially, we will apply five ML techniques that are widely applied in current research. These are K-Nearest Neighbors, Lasso & Ridge Regression, Decision Tree Models, Random Forests as well as Artificial Neural Networks (ANNs). For each of the techniques, we will (I) present a short introduction to the application, (II) define the respective training and validation steps, (III) explain approaches to parameter fine-tuning and (IV) present the error-based results in comparison to the training error obtained by the authors. The code as well as code documentation is presented in a Jupyter Notebook which will be handed in together with the report. Further, we will submit the predicted values for the three best-performing techniques.

In the last section, we will then document the replication of the social-welfare graph constructed by the authors and based on different eligibility thresholds for the program.

Data Preprocessing/Cleaning

While most of the data preprocessing and cleaning has already been taken care of in the provided dataset, we undertake some additional steps in order to get one coherent dataset for all subsequent modelling techniques applied. Mainly, we get rid of the following columns, which are not needed for prediction: 'percapitaconsumption', 'd_fuel_other', 'd_water_other', 'd_wall_other', 'd_roof_other', 'd_floor_other', 'd_insurance_0', 'd_crowd_lessthan1', 'd_lux_0', 'training', 'poor', 'h_hhsize', 'id_for_matlab', 'hhid', 'lncaphat_OLS' and 'percapitahat_OLS'. Next, since we only have binary exploratory variables, and it does not make much sense to impute missing values using a model, in the addition to the fact that there are not many missing values, we drop any observations with missing values. Lastly, we save this final dataset as our Data Preprocessed to be worked with in modelling.

K-Nearest Neighbors

In this section we present the methodology of the k-nearest neighbors algorithm (KNN) in making our predictions. First, we present a short summary of KNN algorithm and give an overview of our dataset. Afterwards, we describe the most important parameters for this algorithm and explain our approach for the parameter fine-tuning. Finally, we compare the outcome of this method with the results from the authors.

KNN is a non-parametric method used for classification and regression. Neighbors-based regression can be used in cases where the data labels are continuous rather than discrete variables. Therefore, for our application we use the regression algorithm. The label assigned to a query point is computed based on the mean of the labels of its k-nearest neighbors.

We take the available training data consisting of 64 features and 22'674 observations to train the model and for the implementation in Python we use the KNeighborsRegressor function provided by the sklearn package. Important parameters for the algorithm are the **number of neighbors** to use for queries, the **weighting function** of the neighbors used for prediction, the **algorithm** used to compute the nearest neighbors and the **distance metric** used for the regression model.

We start by trying to find the optimal number of neighbors k. To do so, we train different models with distinct k's. We evaluate the different model based on the MSE and identify the model with the lowest MSE. In order to achieve more robust models, we build a k-fold cross validation (CV) into our simple trial model. A 10-fold CV leads to the optimal parameter $k \approx 25$ with a mean MSE of 0.21039. We repeat the same using a 7-fold CV and 5-fold CV. While a 7-fold CV leads to an optimal number of neighbors of 26 with an average MSE of 0.2104, the 5-fold CV yields a minimal MSE of 0.21131 with $k \approx 24$. Finally, we take the average of the three attained optimal parameters to use in the end, which gives us the optimal number of neighbors in the model of 25. We will use this specification to test other parameters. At this point, all other parameters are set to standard.

Next, we will optimize the weight function used in predictions. The standard is 'uniform', where all points in each neighborhood are weighted equally. The other option would be 'distance', where neighbors are weighted by the inverse of their distance. In this case, closer neighbors of a query point will have a greater influence than neighbors which are further away. Using a 10-fold CV, we get an average MSE for the uniform distance of 0.21039 and 0.23747 for the distance inverse weighted option. Therefore, we will move on with the uniform distance weighting.

For the optimal choice of algorithm to compute the nearest neighbor, we will use the 'auto' option. This option will attempt to choose the most appropriate algorithm between BallTree, KDTree and a brute-force search based on the values passed to fit method.

Lastly, we can fine-tune the distance metric, where we can choose between Euclidean, Manhattan or Minkowski distance function. The Minkowski distance function with power 2 is equal to the Euclidean distance, while with power 1 it is equal to the Manhattan distance.

We will use again a 10-fold CV to decide which model is optimal. However, the distance metric does not improve the MSE and we therefore will go on with the standard Euclidean distance function.

If we now combine the optimal parameters obtained from the selection process and apply the KNN to the entire training data set of 22'674 observations, we obtain a MSE_{KNN}^{Train} of 0.19367. Applying the same model to the testing data set of 22'704 observations, we get a MSE_{KNN}^{Test} of 0.19251.

Ridge & Lasso Regression Models

In this section we present the methodology we follow to make predictions based on Ridge and Lasso Regression models. In a first basic approach, we simply take the available training data consisting of 64 features and 22'674 observations to train appropriately specified Ridge and Lasso Regression models. All the tools we make use of to estimate these models and getting predictions are provided by the classes Ridge and Lasso from the sklearn package in python. To find the best hyperparameters λ for either model, we program a function which tests a determined grid of values as hyperparameters in the model under consideration, identifies the value for λ that leads to the smallest average MSE by means of K-fold cross validation (CV), and plots each λ in the grid against the corresponding average MSE. This function requires us to specify the following inputs: observations for explanatory variables of the training data (as numpy ndarray or pandas DataFrame), observed labels of the training data (as numpy ndarray or pandas DataFrame), number of folds (K) to use in the K-fold CV, the lower bound and upper bound of the grid of λs to apply (in the sense that we specify the variables *pow_min* and *pow_max*, such that the grid then ranges from 10^{pow_min} to 10^{pow_max}), and the number of values we want to be contained in the grid (in the sense that we specify the variable *nsteps* which determines the number of equally large steps to increase the exponent of the hyperparameter from *pow_min* to *pow_max*). To save time and computational power we then use this function to iteratively determine the best lambdas to use in the Ridge and Lasso Regression models with a sufficiently high accuracy. Hence, we start by running our function using 7-fold CV with a comparatively wide and coarse grid of λs (i.e. a grid from 10^{-15} to 10^{10} by 10 steps of the exponent) to get a first impression of where the optimal lambda might be located. Thereafter, we continue by narrowing and refining the grids step by step until we arrive at a sufficiently precise value for the optimal lambdas. Applying this procedure leads to the optimal hyperparameter $\lambda_{Ridge}^{optCV7} \approx 4.27$ for the Ridge Regression and $\lambda_{Lasso}^{optCV7} \approx 7.21*10^{-5}$ for the Lasso Regression. In order to achieve somewhat more robust models, we repeat the same using 5-fold CV and 10-fold CV, and take the average of the three attained optimal hyperparameters as the λ to use in the end. For 5-fold CV we observe $\lambda_{Ridge}^{optCV5} \approx 4.06$ for the Ridge Regression and $\lambda_{Lasso}^{optCV5} \approx 7.67 * 10^{-5}$ for the Lasso Regression, whereas for 10-fold CV we observe $\lambda_{Ridge}^{optCV10} \approx 4.07$ for the Ridge Regression and $\lambda_{Lasso}^{optCV10} \approx 3.06 * 10^{-5}$ for the Lasso Regression. Finally, this leads to the final hyperparameters $\lambda_{Ridge} \approx 4.1335$ for the Ridge Regression and $\lambda_{Lasso} \approx 5.9816 * 10^{-5}$ for the Lasso Regression. We then use these hyperparameters accordingly to train a Ridge and a Lasso Regression on the whole training data. We observe that the resulting Lasso model sets four parameters exactly to

0, and therefore only relies on 60 out of the 64 available features. Based on the resulting models, we then make predictions for all available observations and calculate the training MSE and test MSE. We observe that the training MSE of both models is slightly higher than the one observed for the OLS model ($MSE_{OLS}^{train} \approx 0.19147$ vs $MSE_{Ridge}^{train} \approx 0.19148$ vs $MSE_{Lasso}^{train} \approx 0.19152$). This intuitively makes sense, as Ridge and Lasso Regressions essentially rely on the same linear model as OLS, but add a regularisation term to the objective function to minimize. By adding this term, the Ridge and Lasso Regressions slightly increase the bias relative to the OLS model (which corresponds to increasing the training MSE) with the aim to substantially decrease the variance of the model. However, since we are relying on the exact same data as the OLS model (i.e. without any expansion of the basis or any additional features), it is not possible to get a lower training MSE than the OLS model using another linear model, because the objective of OLS essentially is to determine those model coefficients that lead to the lowest training MSE achievable with a linear model. For the test MSE, on the other hand, it is indeed possible to get a lower value than the OLS with a Ridge or Lasso Regression model. However, in the present case we observe that both applied models lead to a slightly higher test MSE than the one resulting from the OLS model ($MSE_{OLS}^{test} \approx 0.19087$ vs $MSE_{Ridge}^{test} \approx 0.19088$ vs $MSE_{Lasso}^{test} \approx 0.19095$).

In the second attempt we apply a basis expansion approach. However, since the features in the data are exclusively binary variables, it is meaningless to include squares of features, as the resulting variables would coincide with the original features themselves. Therefore, we apply a basis expansion of degree two (using the sklearn class PolynomialFeatures) to the feature space, but only adding the resulting $\sum_{i=1}^{63} i$ interaction terms to the set of features. This strategy leads to a set of 2'080 features in total. Since a large share of these new features are probably poor predictors, we decide to only apply the method which performs variable selection - namely the Lasso Regression. Hence, we use again the function we defined earlier to iteratively determine the optimal hyperparameter by means of 5-fold CV. In contrast to the basic approach explained above, we directly take the optimal λ obtained by 5-fold CV as our final $\lambda_{LassoExt}^{opt}$, instead of averaging over the three optimal hyperparameters obtained by means of 5, 7 and 10-fold CV. The rational therefore is to save time and computational power, as the required time to perform K-fold CV or estimate the model is much higher than before due to the significantly larger set of features. However, the rest of the methodology coincides with the one explained for the basic approach from above. Using $\lambda_{Lasso}^{opt} \approx 2.2574 * 10^{-4}$ we observe that the Lasso model sets the coefficients for 1'582 out of the 2'080 available features exactly to 0, and therefore only relies on the 498 remaining ones. Further, we observe that this extended Lasso model achieves a lower training and test MSE than the OLS model ($MSE_{OLS}^{train} = 0.19147$ vs $MSE_{LassoExt}^{train} = 0.17559$ & $MSE_{OLS}^{test} = 0.19087$ vs $MSE_{LassoExt}^{test} = 0.18347$).

Thereafter, we use the predictions of this model to produce the following graphs, which are shown in the appendix: Figure 2, which displays a scatter plot of the predictions vs the actual values, and Figure 3, which illustrates the Receiver Operating Characteristic (ROC) curve of the model[1].

[1]Both of these graphics refer exclusively to the training set of the data.

Decision Tree Classifiers

In this section we present the methodological approach used when applying Decision Trees (DT). We use the same amount of training observations as starting point and use the the DT Regressor function provided by the sklearn package. This function takes as an input value the predictors of the training observation and assigns group status according to a random splitting allocation. Especially, it consists of four distinct hyperparameters. These are "max depth", which gives the maximum depth of the tree, "min samples split", the minimum number of sample required to perform a split, "max features", which represents the number of features to consider when looking for the best split and "min samples leaf", an indicator as to how many samples each leaf must at least obtain in order to be classified as such.

If we randomly assign some variables to these four hyperparameters and run the tree model, we obtain a MSE of 0.241893[2]. As this value is far above the MSE obtained with a simple OLS result, we fine-tune the parameters. Doing so requires us to follow a stepwise selection approach in which we assess the effect of varying each hyperparameter separately and select the one which, ceteris paribus, delivers the lowest possible MSE. Throughout the stepwise-selection process, we use a K-Fold Cross Validation approach to account for robustness of the training results, in which we define K to be 10. That approach provides us with an average MSE for different sample splits for each hyperparameter distribution.

We start by varying the inputs for the maximum depth, which indicates the maximum nodes, or splits, that the tree can take. The advantage of increasing the number of splits is that the tree will become more precise and fit the training data better as it can more thoroughly learn the specification of each observations since it has a wider range of attributes to choose from. As a consequence, the bias will automatically decline with an increasing number of splits and depth. The most prominent disadvantage in said case is that the model becomes prone to overfitting, as it depicts the underlying data structure with such a precision that it becomes unable to generalize the results on an independent sample. We define a range of depths from 1 to 32. After running the specification, we see that the resulting MSE distributions show a U-shaped curve, with its minimum value at a depth of 7 and an associated MSE of 0.233833. This shows that we were already able to improve the model to some extent. Also, we can interpret that the model quickly becomes too specified once we increase the depth of the tree size.

Taking max depth to be 7, we proceed with the stepwise selection process and assess the minimum share of the entire sample required to perform a split. The idea is that, when we increase this parameter, the tree becomes more constrained as it has to consider more samples at each node. As a consequence, an increasing factor will lead to a less complex tree structure. In the case of this hyperparameter, we define a range of 10% to 100%. This approach shows that the factor of 10% appears to provide the best prediction on the validation set. Therefore, we further fine tune the parameter and assess the range of 1% to 10%, but with smaller step sizes. Also in this specification, we obtain that the best prediction is obtained by selecting the

[2]In this case, we started with a max depth of 7, min samples split of 7, max features of 21 and min samples leaf of 2

minimum value. However, since the slope of prediction improvements is gradually declining, we assume that further minimisation would not effectively lead to a sufficient decline in model accuracy, which is the reason why we leave the obtained factor of 1% and obtain a MSE of 0.2315338. As we can see, when having an optimal tree depth selected, the choice of minimum sample number does not appear to have an impact on model performance.

Next, we assess the variation of the minimum number of samples required to be at a leaf node. This parameter is similar to the one above, however, it describes the minimum number of samples at the leafs, the base of the tree. Increasing the number of minimal samples will automatically reduce the "spreading of the tree". Interestingly, we can see that the distribution of errors following variation in this hyperparameter follows a somewhat stepwise process. One indication for this might be that, for a given "sample-per-leaf" range, a tree only reduces its spreading after passing a threshold. As a consequence, if the range has not yet passed said threshold, a decline in samples per leaf will not lead to a better prediction since the tree is unable to spread more widely within that parameter range. Overall, by assessing the distribution of error terms, we once again obtain a decline in prediction error with a decline in samples per leaf, which plateaus roughly around 0.001. As a consequence, we obtain an MSE of 0.231433. Also here, choosing for sample sizes per leaf does not appear to substantially impact the accuracy of our model, given the other hyperparameters.

Lastly, we look at the number of features to consider when looking for the best split. The main idea is that, by being able to select from a greater number of features for each split, we can inherently choose from a wider range of combinations that can potentially increase model accuracy. But also here, if we choose too many parameters, we may be unable to generalize the results. We decide to include all parameter values as maximum number of features that the model is able to choose from. Doing so shows a declining but stabilising error distribution. In the case of our model, we find a maximum number of 58 features to define the lowest possible MSE.

If we now combine each optimal hyperparameter obtained from the selection process and apply the DT to the entire training dataset of 22'674 observations, we obtain a MSE_{DT} of 0.2275. As we can see, this MSE is still larger than the one obtained by simple linear regression. One factor for this might be that the underlying data distribution is just better depicted through a linear process instead of a stepwise, range-based distribution. Further notice that one shortcoming of a DT is that it is prone to overfitting based on its lack of bootstrap aggregation properties. As a consequence, we can apply Random Forests to further specify the tree-based model.

Random Forests

In this section we present the methodological approach used when applying Random Forest (RF). We use the same amount of training observations as starting point and use the RandomForestRegressor function provided by the sklearn package. As a basis of the RF approach, one can take the DT method discussed above. The main disadvantage of DT is, that if a tree is grown deep, it has low bias but high variance, so one has to constrain the trees in order to have a balanced trade-off between bias and variance. This is exactly where RF steps in, in order to improve the DT method using statistical learning. RF regression grows trees deep with low bias and high variance. However, it does not stop after having calculated only one tree. It rather grows many trees and averages them to reduce the variance. This is achieved with bootstrap, by taking repeated samples from our training data set. Hence, the trees in RF can be grown deep and do not need to be pruned. If we allow our model to use the full set of predictors to grow our trees, it is a special form of RF which is called bagging. We also restrict our model to use only a random subset of predictors with a fixed number of predictors which is necessary in order to decorrelate our trees. If we would not restrict our model, then there is a very high chance that we have the same predictor in our first split of the tree in most of the cases, which leads to correlated trees. Once we decorrelate our trees, we can further reduce the variance of the trees. In RF regression, it is not necessary to perform a CV in order to obtain a test error. Remember that the key of RF regression is bootstrapping. It can be shown that each tree uses around 2/3 of the observations. The remaining third, called out-of-bag (OOB) observations, can then be used for prediction, since it was not used to grow the tree. Hence, one can use the OOB sample instead of doing a separate cross-validation. However, since the sklearn package does not allow to directly retrieve the MSE of the OOB sample, we also perform a CV (but only 2-fold) in order to calculate a test MSE on unseen data. This is done, since RF are prone to overfitting.

To estimate our preferred model we have to fine-tune only two parameters, the number of trees to grow and the number of predictors we allow to calculate our RF model. We start with the fine-tuning of the number of predictors (m) which are allowed to be taken into the calculation. There are two commonly used number of predictors. First, the number of predictors is not restricted and allowed to contain the full set of predictors (p), $m = p$. This is the equivalent to the above mentioned method Bagging. Since this is a special case of RF, we subsumed it into the RF section. The second subset of predictors, which is mostly used in practice, is equal to the square root of the total set of predictors, $m = \sqrt{p}$. In order to have a third subset of predictors, we further look at another, less often used subset, containing of half the predictors, $m = \frac{p}{2}$. We then run a RF regression loop over the three different set of predictors, keeping everything else constant, apart from the number of trees in the forest (k). For each forest with a given number k of trees and a subset of predictors m, we then calculate the R^2 of the forest. R^2 is defined as $R^2 = 1 - \frac{u}{v}$ where u is the residual sum of squares and v is defined as the total sum of squares. The R^2 can be positive with a maximum of 1 or negative. 0 is achieved if one always predicts the expected value of the dependent variable. A negative value indicates, that the model is arbitrarily worse. These values are then plotted, and analyzed. Bagging (m=p) yields the worst results. The other two subsets of predictors are different, depending on the number of k trees in the forest. However, mostly $m = \sqrt{p}$ performs best, especially as k grows, so we take $m = \sqrt{p}$ as our preferred subset of predictors.

Next we have to determine which number of trees in the forest, k, gives us the best predictions. We therefore again loop a RF regression, keeping everything constant, with the subset of regressors at $m = \sqrt{p}$. The only difference in the model is the number of trees in the forest. Here, we use 2-Fold CV to see how the RF overfitts the data. For every k, we calculate the MSE of the training set, as well as the OOB score. The results are again plotted and analyzed. It turns out that $k = 343$ has the best predictions. If we now combine the optimal parameters obtained from the selection process and apply the RF to the entire training dataset of 22'674 observations where we obtain a MSE_{RF} of 0.05673. Since RF is prone to overfitting, we also have to check how well the predictions perform on unseen data. Therefore we split the 22'674 observations in half and estimate the model and the corresponding training MSE_{RF} on one half of the dataset and then create predictions on the test set (the other half of the dataset) which was not used to estimate the tree. Here we find a much higher MSE of 0.21578. This result provides strong evidence that the RF overfitts the trees and hence only performs very well on the set it is trained on and performs a lot worse if one predicts on data not used to train the RF. It is, however, still an improvement over the DT approach that undermines, such that there is a benefit of the RF approach. Due to the fact that RF is overfitting the data a lot, we do not include this approach for our final predictions. Therefore, we continue with the next modelling approach.

Artificial Neural Networks

In this section we present the methodology of artificial neural networks (Deep Learning) in making our predictions. The novelty of the use of Artificial Neural Networks (hereinafter referred to as ANN) in practice, while not the mathematical foundations behind them, leaves a fair amount of ambiguity in the way to go about its implementation. Nevertheless, this section tries to bring some structure to the approach by taking a look at and adjusting, step by step, the parameters as well as the architecture of the network, continuing with the best performing model (according to mean squared error) at each step. Note, that for the implementation in Python, the Keras library [3] is employed, as an interface for TensorFlow.

To start off, a simple trial model is built as a reference point. This trial ANN is created using the Sequential class and is made up of an input layer, three hidden layers and an output layer. In each layer we initially set the number of neurons equal to the number of input parameters, namely to 64. While this does not leverage the advantageous properties of ANNs, we begin by imposing a linear activation function. As an optimizer, we go for the Adam optimizer, commonly known as being one of the most efficient optimizers. As a loss, we use the mean squared error (hereinafter referred to as MSE) as by convention. We train the model using a number of 30 epochs and a batch size of 32, which is the default mode in Keras. As a validation split, a value of 0.2 is set. This simple trial ANN, leads to an MSE of 0.194753.
Next, we build cross validation (hereinafter referred to as CV) into our simple trial model. We begin with a three fold CV, printing the MSE for each fold (Fold MSE) as well as the CV MSE, which is the MSE of all concatenated predictions, and the Training MSE, which is the MSE on the entire 22'674 observations of the training set. The Training MSE is slightly higher, at 0.201477.
Subsequently, we add in loops over the CV model. The idea of adding loops, is that ANNs

[3]This library and its use can be explored under https://keras.io.

provide a different MSE every time they are run. In order to eliminate some of this variation and create more robust predictions, the model is looped over three times, collecting predictions along the way and finally averaging over them. These averages represent the final predictions and allow us to compute the Final Training MSE, which is equal to 0.199779 in this case.

Now that we have a cross validated model with loops built over it, we can, step by step, tune the parameters and work on the architecture of the ANN. This brings us into a "Wild West"-like environment, where trial and error is key. We look at the following parameters: Number of folds, Number of loops, number of layers, number of neurons, activation function, optimizer, number of epochs and batch size.
The first parameter we take a look at is the **number of folds**. Up to now we have seen an ANN with three folds. Now, we test two more options, namely an ANN with k = 5 as well as k = 10 folds. We get a Final Training MSE of 0.196834 and 0.192687 respectively. Since we get the lowest MSE using 10 folds, we continue with this model.
Next, we increase the **number of loops** from three to ten, giving us a slightly lower Final Training MSE of 0.192151. We stick with ten loops for now, and do not increase or test this further, due to the already significantly high computational power use.
Thirdly, we take a look into one part of the architecture and change the **number of layers** of the ANN, to both five and ten, as opposed to the three it had before. This results in a Final Training MSE equal to 0.192828 and 0.192343. It is noteworthy, that increasing the number of layers to five and ten is by far more expensive computationally and does not decrease the MSE.
Continuing with the architecture, in a fourth step, the **number of neurons** within the layers are altered, namely decreased down to twenty and six neurons per layer. The MSE calculated are 0.192140 and 0.192111 respectively. Therefore, it seems that fewer neurons, perhaps contrary to intuition, lead to a lower MSE.
In a fifth step, we take look at the **activation function**, namely, we test four different activation functions: Rectified Linear Unit (relu), Sigmoid, Softmax and Hyperbolic Tangent Activation Function (tanh). The Final Training MSEs using these four activation functions are 0.182719, 0.181693, 0.181572 and 0.178590. It is noteworthy how the MSE decreases once we leverage the power of ANNs using non-linear activation functions. All MSEs are below 0.19, and moreover, the tanh MSE is below 0.18, which is the model we will use going forward. Note that the activation function in the final layer is always linear.
We then take a look at the **optimizer**. We test out both the Stochastic Gradient Descent (SGD) as well as the RMSprop optimizer, neither of which lead to a lower MSE (0.193739 and 0.198433 respectively). This leads us to stick to the Adam optimizer.
Next, we vary the **number of epochs**. We increase the number first to 50 and then to 100. For 50 epochs, we get an MSE of 0.175558 and for 100 epochs an MSE of 0.174038. We see that increasing the number of epochs, allows the algorithm to search more and decreases the MSE.
Finally, we change the **batch size** and test out both a batch size of 25 as well as one of 20. The Final Training MSEs this provides are 0.174544 and 0.173549 respectively. This MSE = 0.173549 is the lowest MSE we achieve using ANNs. Therefore, we use this as our final ANN model.

We now have the final ANN model we use to make predictions on our test/holdout set. It is a ten fold CV model, with ten loops over it. It has one input layer, three intermediate hidden layers and an output layer, where each hidden layer is made up of six neurons (see architecture in Figure 4 and Figure 5 in the Appendix). The activation function chosen is the tanh activation function, except for the last layer, which contains the linear activation function. In compiling the model, we have the adam optimizer and a MSE loss function. Finally, we train the model with 100 epochs and a batch size of 20.
Lastly, this final ANN model can be applied to the test/holdout set, which the model has never seen before. On this new test/holdout set, the ANN achieves a Test MSE of 0.175511.

Model Selection

In this concluding section we briefly mention which models we choose to create the final predictions we submit together with this write-up. As we observe that neither the KNN nor the DT approach were able to outperform the MSE of the OLS predictions, we decide to not include predictions from these models. Further, we also agree not to rely on the RF model as it appears to be subject to overfitting. When considering both, the Ridge and Lasso model approaches, we observe that the basic models were also not able to beat out the OLS predictions. However, the extended Lasso model as well as the ANN model that we estimate are both able to outperform the OLS predictions in terms of lower training and test MSEs. Hence, we agree on submitting the predictions of these two models as our final result of this project.

Replication of the Social Welfare Figure

In this section we discuss how we replicated the Social Welfare Figure the authors provided in their assessment section. Our version of this graph is shown in Figure 1.

We wanted to thoroughly understand the mechanisms the authors chose behind their welfare assessment strategy. As a consequence, we decided against using some of the variables provided by the full Peru data set (such as the poor indicator) and instead attempted to create our own indications of eligibility. Thus, it may very well be the case that some of the numbers cannot be fully replicated through our code, as we are operating with a somewhat changed data structure.

In order to define the eligibility thresholds, we replicated the Receiver Operating Characteristics (ROC) curve for the predicted values we obtained from the extended Lasso model we estimated[4]. This provided us with a rather similar figure when comparing it with the one of the authors. We started by taking the usual definitions for True Positive Rates, False Positive Rates, True Negative Rates as well as False Negative Rates. In a subsequent step, we calculated the inclusion and 1 - exclusion errors manually. This step was then put into a

[4]The rational for relying on the Lasso model predictions to produce this figure is that among the ML techniques we applied the Lasso is most closely related to the OLS model and therefore enables to best mimic the shape of the original curve.

for loop which defined individual eligibility cutoffs according to different percentile rankings.

Then, we replicated the Figure indicating HH benefits for eligibility cutoffs. We defined HH benefits as the overall monthly program budget divided by overall Peruvian households included according to the pre-defined inclusion error. The idea was that, when using a Universal-Basic Income strategy (UBI), each household will receive some piece of the entire budget, which would be $\frac{880'000'000/12}{6'750'000}$ Peruvian Sols. Then, by continuously increasing the eligibility cutoff, a decreased proportion of the overall sample will be included in the budget constraint, thereby increasing the amount available for those that remain within the program. Accounting for exclusion errors, we then were able to define a similar curve as the one of the authors.

Building on our knowledge we obtained through the creation of the first two plots, we then used the given eligibility cutoffs and inclusion error rates to compute the respective Social Welfare per cutoff rate. Doing so, we followed quite precisely the approach the authors chose. However, based on different prediction measures (i.e. predictions from the extended Lasso model), we clearly do not precisely obtain the same results as the authors. Especially, we observe that the welfare status is lower when the program approaches a UBI character. Quite interestingly, though, is that we obtain a maximum social welfare of **-0.12246** with an inclusion error of **8.44 percent** and a subsequent exclusion error of **32.74 percent**. This indicates that, by allowing a slightly larger inclusion error we were able to attain a substantially lower exclusion error, which means that, besides the overall welfare has increased, we were able to more accurately assign treatment when it was indeed required.

Although the results and approaches do not exactly resemble the path required by the task, we nevertheless deemed it more useful to built our own identification strategy for defining the eligibility, ROC and welfare characteristics. As such, we were not only able to gain important insight into the authors' strategy, but were also able to define additional ideas for both eligibility as well as welfare comparisons while building a sufficiently accurate replication.

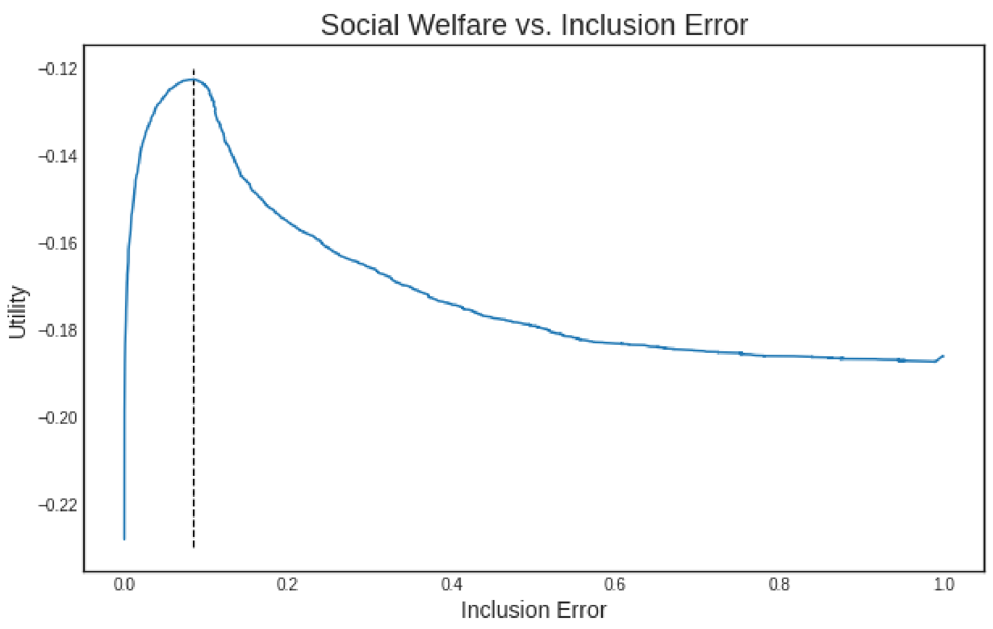

Figure 1: Social Welfare vs. Inclusion Error (using Predictions from the Extended Lasso Model

Appendix

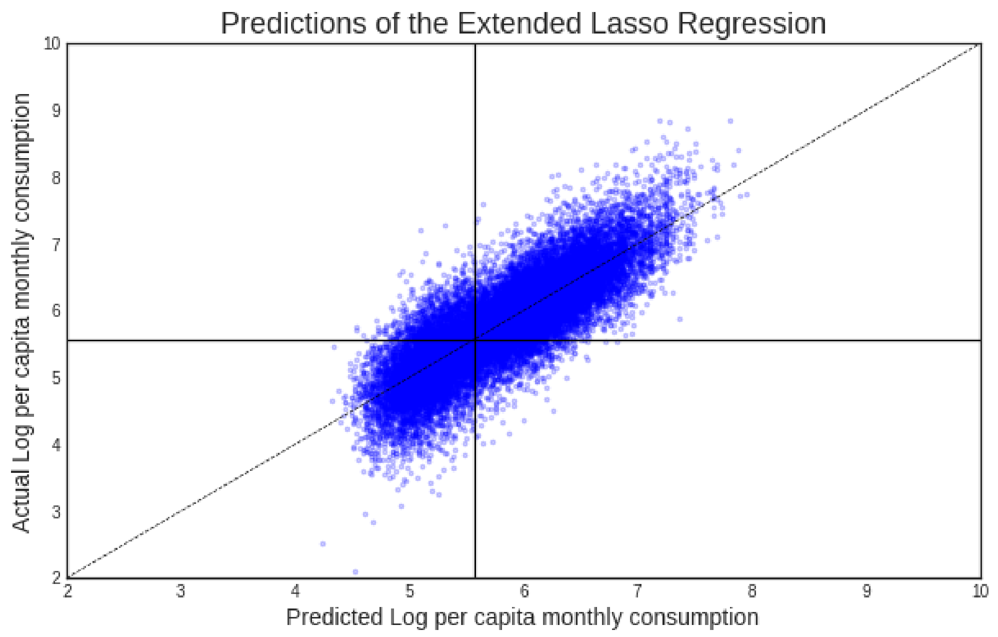

Figure 2: Extended Lasso Regression: Predicted vs. Actual log per Capita Monthly Consumption

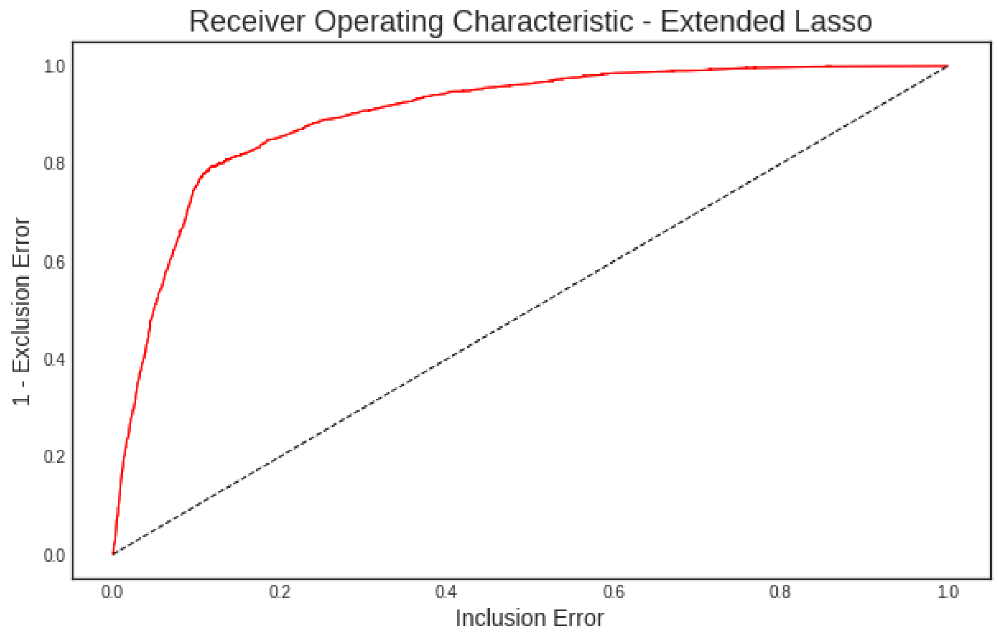

Figure 3: Extended Lasso Regression: Receiver Operating Characteristic

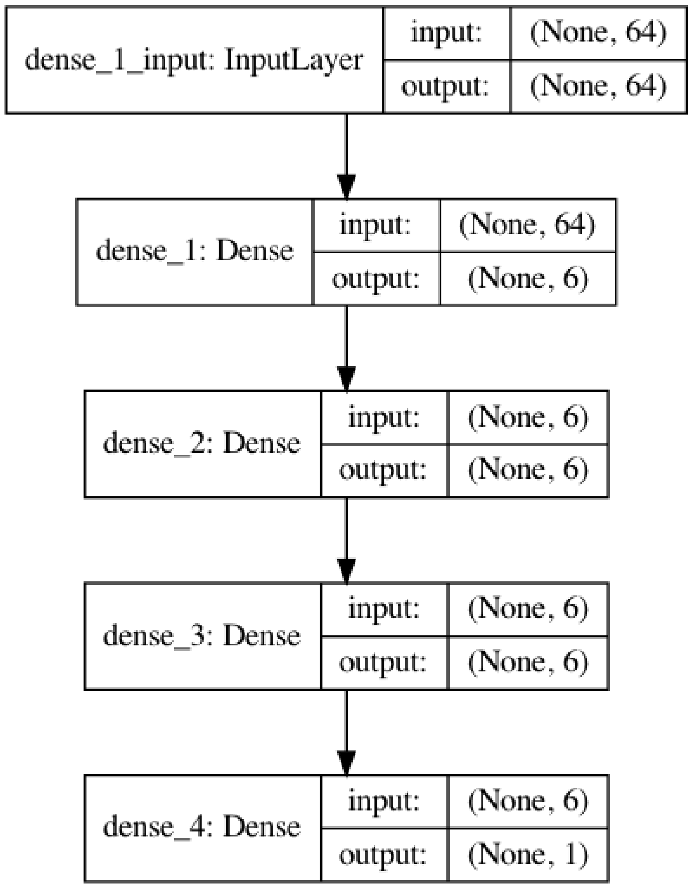

Figure 4: Architecture of the Final ANN

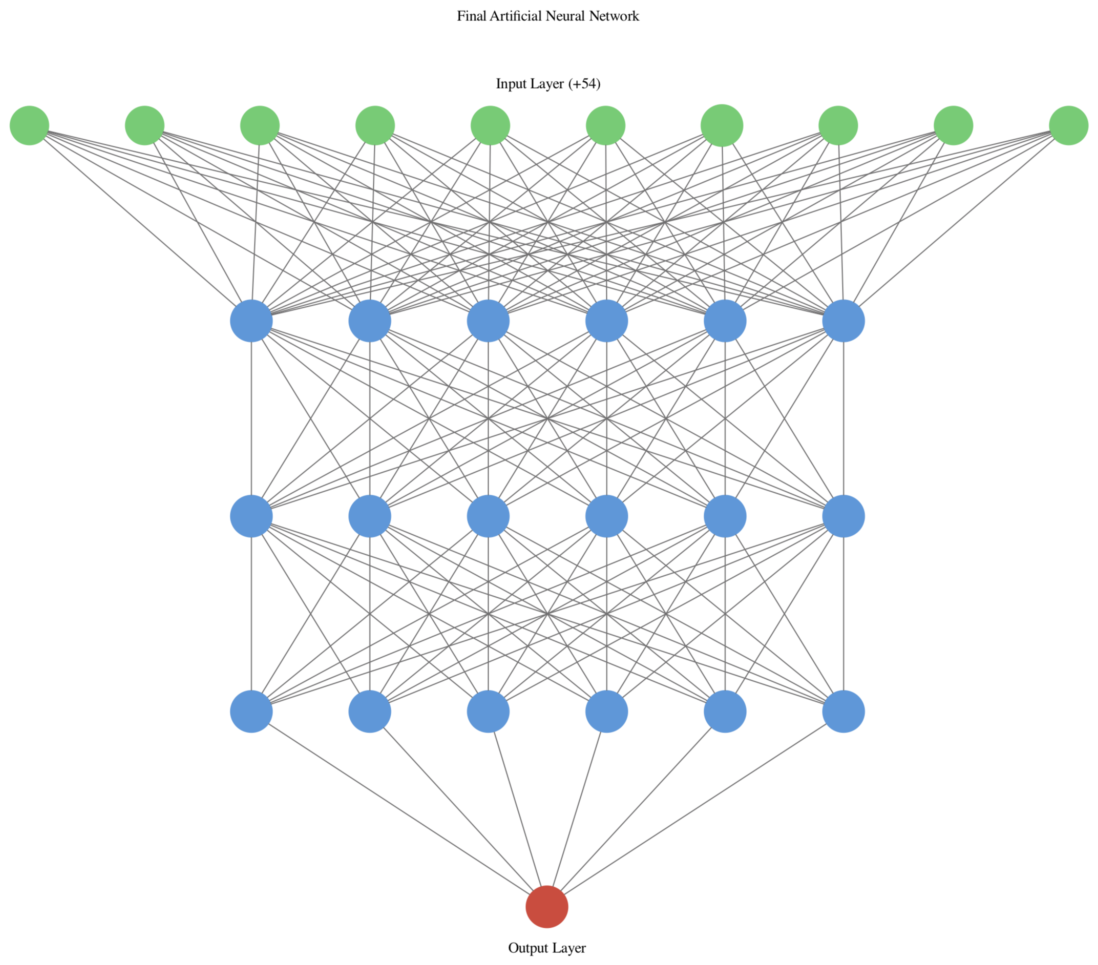

Figure 5: Architecture of the Final ANN