CS 5350/6350: Machine Learning Fall 2023

Final Report: Kurt Namini

The final implementation and instructions for executing the pythong script can be found here

1 Problem Background & Motivation

The problem for this project is leveraging demographic census data to predict income levels. This sort of analysis is important to large institutions like the federal/state/local governments and major corporations that have a large stake in the economic health of the population they provide services to or create policy for. For instance, federal policymakers can leverage these predictions to determine proper resource allocation for social welfare programs. Low-income individuals and families can be properly identified in order to address food insecurity and provide necessary healthcare benefits for those in need. With rising higher-education costs and low primary-school teacher pay, income predictions can be used to create policy that improves access to education and the quality of education. Understanding income inequality can lead to a multitude of helpful applications in public policy. Businesses can leverage these predictions to optimize various practices such as advertising strategies, market expansion, how to price their products, and to predict the behavior of distinct demographics of consumers. Leveraging demographic census data with machine learning can be used in a variety of domains that have a strong impact on large groups of people.

The target prediction is whether or not an individual has an annual income greater than $50,000, which is a binary classification task. Many machine learning techniques are well-researched and have theoretical support for predicting binary classes, which is why I used the techniques discussed in this research paper to leverage the provided demographic data.

2 Data Preparation

Before performing any analysis and making predictions on the testing dataset, I made various changes to the original data sets:

1. Handled missing values in the data, which appeared in three different columns in both the train and test sets.
2. Converted categorical variables to multiple binary variables, and normalized quantitative variables.
3. Added absent binary variables to whichever data set was missing them. It is possible for a particular value of a categorical variable to be absent in either the train or test sets, so when a categorical is converted to multiple binary, there may be missing binary variables.

2.1 Missing Values

The three columns, in both the train and test sets, that contained missing data are "workclass", "occupation", and "native.country". Each of these are categorical variables, so I used multiclass logistic regression to impute the missing values. The process for imputing the missing values is the same for each variable in both the train and test sets.

For each variable with missing values, the other two were removed from the dataset, along with the label present in the train set, and the remaining variable with missing data was treated as the new label. Next, the data was split into a train set without the missing data, and a test set containing the missing data. The labels in the train set were then encoded as distinct integers, one for each possible value for the new label. A logistic regression model was fit on the train set, then used to predict the integer encoded labels for the test set. Finally, the predicted labels were decoded to the original format and inserted back into the original dataset.

2.2 Variable Conversions

The process for converting categorical variables is straightforward. For each categorical variable in the data one boolean variable is created for each possible value. All quantitative variables were transformed into standard normal z-scores by subtracting all elements by that column's mean, then dividing by the standard deviation.

2.3 Missing Binary Variables

In the training set, we do not see the value "Holand-Netherlands" for native.country, but we do see it in the testing set. As a result, there is no binary variable for this value of native country in the train set, but there is in the test set. I know this value will be false, or 0, for every row in the train set, so I inserted a column of zeros into the train set at the same index as the test set.

2.4 Assessing Improvement

In the midterm report, my final model was an Adaboost ensemble with 500 decision trees with a maximum depth of 2 and a learning rate of 0.9. After predicting the test set values and submitting to Kaggle, the prediction accuracy was 76.12%. The data used for that submission had missing values imputed with the mode, or most common category, for each variable with missing values. To determine whether the multi-class logistic regression method of imputing missing values improved performance, I trained the same exact model on the new processed train set described above and predicted the test labels. After submitting these predictions, the prediction accuracy improved roughly 3% to an accuracy of 79.41%. It is safe to say that the logistic regression imputation greatly increased model accuracy, and should be used for the final analysis. The final processed train and test sets have the same number of rows, while the columns for each increased to 105 (excluding the label in the train set).

3 Neural Network Architecture

For the final analysis I chose to train a feed-forward neural network. In this section I discuss my process for determining the network structure through theoretical/practical reasoning, as well as K-fold cross-validation. After selecting the final architecture I tuned the remaining hyper-parameters such as learning rate, epochs, and batch size used during training. This portion of tuning is assessed through Kaggle submissions.

3.1 Hidden Layers

From my work in the midterm report attempting to train a Perceptron linear classifier I learned that the data is not linearly separable due to low validation accuracy. In lecture, we learned that a single neuron with threshold activation also represents a linear classifier. Therefore, to fully leverage a neural network I need to include at least one hidden layer of neurons.

I also used decision trees, adaboosted trees, and random forests in the midterm report, which all represent complex boolean functions. Again, in lecture we learned that threshold networks with one hidden layer also represent boolean functions, which I have already attempted with the models mentioned above. In order to utilize a neural network in a way unique to what I have already done for my midterm analysis the architecture should include at least two hidden layers. Putting it all together, I will compare network architectures with 2 hidden layers and 3 hidden layers.

3.2 Neurons Per Hidden Layer

Given a neural network's free form framework and endless options for the number of hidden neurons, I considered a value related to the size of the input layer. To determine my options for hidden layer size, I searched for practical advice and discussions from online forums, such as Stack Exchange. From this research, I encountered many rule-of-thumb methods, most of which state that hidden layers should have size between the number of inputs and outputs. Using this general idea, I decided to compare hidden layer size equal to the input layer, and roughly 75% of the input layer. For simplicity, I make the size of every hidden layer the same.

3.3 Activation Functions

Similar to the selection of hidden layer size, I found practical advice and discussions focused on the use of activation functions. Among the most commonly used are the sigmoid, tanh, and ReLU functions. However, there are unique qualities for each that greatly impacts the learning procedure. For example, when using either sigmoid or tanh activations, we encounter the vanishing gradient problem. This is when an input is large enough in magnitude that the gradient of the activation is extremely small, or essentially zero. Because of the chain rule's presence in backpropagation, there is a good chance that somewhere in a single update (when using sigmoid or tanh activation) and cause unexpected behavior during training. The ReLU activation does not have this issue, which is why it is most used of the three options stated here. Additionally, ReLU has variants that can further improve training or increase computation speed. I decided to include ReLU and ELU (exponential linear units) in the set of activations used in cross-validation. The final activation is always the sigmoid function, which works well with binary classification.

3.4 Cross-Validation

To perform cross-validation with the architecture options I decided on above I created 8 networks, one for each setting of hidden layers, hidden layer size, and hidden activation functions. The learning rate, epochs, and batch size were held to constant values of 0.01, 50, and 200, respectively. 5-fold cross-validation was performed on the processed train set for each network, the results are in the table below.

Hidden Layers	Neurons/Layer	Activation	CV Accuracy
2	75	ReLU	87.23%
2	75	ELU	87.1%
2	100	ReLU	87.38%
2	100	ELU	87.41%
3	75	ReLU	87.18%
3	75	ELU	87.1%
3	100	ReLU	87.78%
3	100	ELU	86.46%

At first glance, there does not appear to be large differences in the cross-validation accuracies. Still, there are a few things to make note of. Holding the hidden layers and layer size constant, the models with ReLU activation perform slightly better on validation sets than their ELU activation counterparts. The networks with 100 neurons per hidden layer perform also perform slightly better than the 75-width counterparts. Finally, the best performing network here follows that logic, and also has 3 hidden layers, for a total of 4 layers when including the output layer. The network I will continue with will have 3 hidden layers, roughly 100 neurons in each hidden layer, and ReLU activations in the hidden layers.

4 Final Model Results

For the tuning of the remaining hyper-parameters (learning rate, epochs, batch size) I take an iterative approach changing one or two hyper-parameters at a time, then submitting the predicted results to Kaggle to obtain the accuracy. For the first iteration, I tried the same setting that was used in cross-validation for network architecture. Results are in the table below.

Rate	Epochs	Batch Size	Kaggle Accuracy
0.01	50	200	88.348%
0.005	50	100	87.462%
0.005	50	200	87.728%
0.01	50	100	88.379%
0.01	75	100	87.773%

As you can see from the table, my prediction accuracy did not change very much through each iteration, but there are two clear selections of hyperparameters that performed the best on unseen data. For my final submission, and what will be included in the final script, will be a 4-layer feed-forward neural network with 105 neurons per hidden layer, ReLU hidden activations, sigmoid output activation, and trained using learning rate 0.01 over 50 epochs and 100 examples used for each batch update.

5 Further Work

If given more time to work with this dataset, there are two areas in which I would attempt other strategies increase generalization accuracy: data preparation and network architecture/tuning.

For data preparation, what I had not considered in this analysis is testing for outliers. As is well known, outliers can have strong influence on the training of a classifier or regressor. If these were identified, analyzed for leverage on the data, and removed if proven to impact the data negatively, there could be positive impact on training.

Considering my current training procedure, I would include some early-stopping logic in tuning the final model using a hold-out set. This would allow me to avoid tuning the number of epochs and updates done during the training procedure, and would increase generalization accuracy by not over-fitting the data. The architecture I came up with was purely experimental, derived from my own understanding of neural networks. I would like to look into whether a similar sort of prediction task has been performed using neural networks, and the architecture used in that similar scenario. On top of that, there are many predefined and commonly known network architectures that I did not consider for my final classifier. If any of those are candidates for increasing accuracy, I would try those that seem to fit this scenario as well.