Final Project Report on the Income Prediction Kaggle Competition

Sabina Miani
CS 5350 – Machine Learning
19 December 2022

Introduction

I chose to work on the individual project via Kaggle on the income prediction problem. My username on Kaggle is 'Sabina Miani'. Some code is on Kaggle, or it can also be found in this GitHub repository: https://github.com/sabinamiani/CS5350-Income-Prediction.

Problem

The income prediction problem combines various characteristics of individuals gathered from the 1994 census database and tries to correctly identify each individual's income based on whether they make above 50K per year. This problem is noteworthy for a few reasons, namely, that census data is more thorough than I had previously thought and that evaluating data that is not explicitly given may still be attainable with an accuracy rating as there may be outliers and non-conformist datasets. Attaining implicit information like this can be applied to other similar problems.

This income prediction problem can be thought of as an application for binary classification where there is only one output label (>50K) of value either one or zero for true or false using the evaluation method based on Area Under ROC (AUC) curve. AUC calculates the area of the curve using thresholds, true positive rate (TPR) and false positive rate (FPR), to measure the overall performance of the model.

Using machine learning to solve this problem was ideal as there are various features that may be correlated, but very difficult for a human to classify, therefore, using a computer to learn the subtle complexities of this dataset was the preferred method. As mentioned above, this problem is an example of binary classification. Applying machine learning principles to a binary classification problem becomes simple with the use of existing python libraries like numpy, pandas, sklearn, and others.

Dataset

The 1994 census database was parsed with this specific project in mind by creating a clean and usable dataset. This dataset is housed within two separate csv files, one containing the training

data and one containing the testing data. The datasets consist of fourteen features which are both numerical and categorical in type and the training set has one additional feature for the income prediction score which is binary, one or zero.

Methods

Researching python libraries and how to use them for this binary classification problem was of high priority. There are a variety of machine learning libraries for python available for public use. Starting with decision trees, I researched different python libraries to create various structures and algorithms. I found the *DecisionTreeClassifier* and *DecisionTreeRegressor* methods from the sklearn library which uses a version of the CART algorithm and does not support categorical variables. I looked around for other libraries before choosing a specific method or library. I looked at various boosting libraries next and found multiple that seemed useful including Boost, Scikit-Learn, XGBoost, and LightGBM. After considering all of these, I decided to just pick a well-documented library. I chose to implement XGBoost since it had good documentation and worked well with other python libraries like pandas and numpy.

Label-Encoding

The csv data files containing both the training and testing data needed to be transformed into workable data structures. Using the pandas library, the csv files were converted into the unique DataFrame type. Since some of the features are categorical, meaning they are values other than numbers, converting them into numerical data was necessary to use many of the machine learning libraries found. However, a discovered experimental method to automatically convert the categorical data into numerical data during the model training phase showed promising results. Unfortunately, I was unable to get the method to work correctly with the given dataset. This may have been a result of human error in part of not understanding the correct input and output of the experimental method or it could be that the method was too unstable to be reliable. Either way, a new approach was needed.

Further research into converting categorical data, described this situation as label-encoding, which means that a numerical label would be assigned to each new category value as it was discovered. After understanding label-encoding, it was found that the pandas DataFrames have a way to convert categorical data into numerical data using the DataFrame ‘cat’ property, short for CategoricalAccessor. This property can define a code for each new categorical value in the specified DataFrame given a specific feature or column in the DataFrame. After iterating through each categorical feature in the census dataset and reassigning the DataFrame to the produced cat codes, the data was successfully converted. Besides this approach, there are many ways of converting categorical data. I attempted other methods, but ultimately used the method described.

XGBoost

After converting the data into usable structures, applying a learning algorithm was the next step. Earlier in the semester, the number of parameters and unknown applications of XGBoost were overwhelming which led to the use of a logistic regression model producing probability output. Thankfully, the label is between one and zero and the probability outputs worked despite this. After changing the objective function from binary logistic to binary hinge to produce the zero and one label values, the prediction scores did not change. This lack of change is likely due to similar computation algorithms for both objective functions.

XGBoost has a variety of hyperparameters one can adjust based on the application and algorithm. After learning more about the hyperparameters and potential benefits of changing some of them, I found that changing the learning rate and gamma values drastically changed the results. Experimenting with the learning rate, gamma, and maximum depth (using tree logic) gave varied results (see *Text 1*).

Running the same code, but changing the number of rounds during training from 10 to 20, the training accuracy score went above 91%. This shows signs of overfitting, and when submitting the best score prediction model via Kaggle, the test score was about 75% where previous Kaggle submissions with lower training accuracy scores received significantly higher test scores in the 90 percent range. Overfitting behavior was likely as the max depth and number of training rounds increased.

To counteract the overfitting behavior, cross validation was applied using XGBoost cv function to test for the best hyperparameters. It was found that when the learning rate was 0.1, the AUC evaluation mean score for each iteration was consistently low. Not only was it low, but the same values, 0.5 for mean and 0.0 for standard deviation, persisted across each iteration of hyperparameter adjustments. After applying cross validation, it was evident that changing gamma actually did not change the AUC scores at all. This was very surprising especially since the training accuracy scores varied slightly when running the trained model. When running without changing gamma (see *Text 2*), the higher learning rates seemed to produce a higher AUC mean score with similar findings when increasing the max depth. Unfortunately, this did not solve the overfitting issues like planned. This is likely due to human error in understanding the results produced by the unfamiliar methods.

Results

---

max depth= 2 lr= 0.1 gamma= 0.1 score= 0.73004 BEST!
max depth= 2 lr= 0.1 gamma= 0.7 score= 0.73004
max depth= 2 lr= 0.1 gamma= 0.3 score= 0.73004
max depth= 2 lr= 0.1 gamma= 1 score= 0.73004
max depth= 2 lr= 0.1 gamma= 0.9 score= 0.73004
max depth= 2 lr= 0.7 gamma= 0.1 score= 0.84284 BEST!

```
max depth= 2  lr= 0.7  gamma= 0.7  score= 0.84284
max depth= 2  lr= 0.7  gamma= 0.3  score= 0.84284
max depth= 2  lr= 0.7  gamma= 1  score= 0.84284
max depth= 2  lr= 0.7  gamma= 0.9  score= 0.84284
max depth= 2  lr= 0.3  gamma= 0.1  score= 0.83976
max depth= 2  lr= 0.3  gamma= 0.7  score= 0.83976
max depth= 2  lr= 0.3  gamma= 0.3  score= 0.83976
…
max depth= 8  lr= 0.3  gamma= 1  score= 0.87732
max depth= 8  lr= 0.3  gamma= 0.9  score= 0.87732
max depth= 8  lr= 1  gamma= 0.1  score= 0.89848  BEST!
max depth= 8  lr= 1  gamma= 0.7  score= 0.89424
max depth= 8  lr= 1  gamma= 0.3  score= 0.89772
max depth= 8  lr= 1  gamma= 1  score= 0.89872  BEST!
max depth= 8  lr= 1  gamma= 0.9  score= 0.89764
max depth= 8  lr= 0.9  gamma= 0.1  score= 0.89812
max depth= 8  lr= 0.9  gamma= 0.7  score= 0.89244
max depth= 8  lr= 0.9  gamma= 0.3  score= 0.89384
max depth= 8  lr= 0.9  gamma= 1  score= 0.89272
max depth= 8  lr= 0.9  gamma= 0.9  score= 0.89244
```

Text 1. Partial code output while training the XGBoost on 10 rounds with various tree depths, learning rates, and gamma values. The accuracy score on the training dataset is also provided. For easy high score depiction, ‘BEST!’ appears after each new best score found.

```
max depth= 2  lr= 0.1
   train-auc-mean  train-auc-std  test-auc-mean  test-auc-std
0             0.5            0.0            0.5           0.0
1             0.5            0.0            0.5           0.0
2             0.5            0.0            0.5           0.0
3             0.5            0.0            0.5           0.0
4             0.5            0.0            0.5           0.0
max depth= 2  lr= 0.7
   train-auc-mean  train-auc-std  test-auc-mean  test-auc-std
0        0.756951       0.001203       0.756884      0.004796
1        0.756951       0.001203       0.756884      0.004796
2        0.714182       0.002611       0.714023      0.009234
3        0.717711       0.002540       0.717924      0.009388
4        0.717658       0.002747       0.718007      0.009413
max depth= 2  lr= 0.3
   train-auc-mean  train-auc-std  test-auc-mean  test-auc-std
0        0.500000       0.000000       0.500000      0.000000
```

```
1       0.756951      0.001203      0.756884      0.004796
2       0.756951      0.001203      0.756884      0.004796
3       0.756951      0.001203      0.756884      0.004796
4       0.713973      0.002340      0.713942      0.009326
…
max depth= 8  lr= 0.3
   train-auc-mean  train-auc-std  test-auc-mean  test-auc-std
0       0.500000      0.000000      0.500000      0.000000
1       0.784601      0.005627      0.776590      0.004959
2       0.838839      0.001545      0.817831      0.004937
3       0.839449      0.001414      0.817604      0.007650
4       0.838956      0.002299      0.812690      0.005898
max depth= 8  lr= 1
   train-auc-mean  train-auc-std  test-auc-mean  test-auc-std
0       0.832275      0.001165      0.814851      0.005302
1       0.825566      0.003383      0.794977      0.004561
2       0.819351      0.005142      0.788407      0.008479
3       0.820753      0.002565      0.784585      0.006921
4       0.822934      0.004104      0.784153      0.009138
max depth= 8  lr= 0.9
   train-auc-mean  train-auc-std  test-auc-mean  test-auc-std
0       0.831644      0.001250      0.816018      0.005478
1       0.827117      0.002433      0.796244      0.005461
2       0.818328      0.004768      0.790698      0.007081
3       0.820322      0.005406      0.787642      0.005781
4       0.822960      0.005285      0.788823      0.004584
```

Text 2. Sample XGBoost cv method output after running with several max depth and learning rate values after 5 rounds of boosting using the binary hinge objective function with respect to AUC evaluation.

After evaluating the above outputs and then using some trial and error in submitting various output files, I found a high accuracy test score while having a max depth of 5 and learning rate at 1. When submitting it through Kaggle, the reported score was 92.281%. I had very similar results when submitting results from using a max depth of 4. However, when using a greater depth than 5, the accuracy results significantly decreased. When the max depth was set to 6, the accuracy score dropped to 79.121% and decreased slightly from there each time the max depth was lowered. If the output from Text 2 can be trusted, then the greater the max depth, the better the accuracy score should have been; however, this is clearly not the case and instead is the product of severe overfitting.

Continuation

In furthering this project solution, more research into understanding the methods and more about XGBoost would be required. Overcoming the overfitting issues required a trial and error method for the approach described above; however, in the future, a more precise method of finding hyperparameters should be applied.

Using an alternate library to train a binary classification model may provide better results. Exclusively using XGBoost for this solution, may not have been the best library for the problem set and more exploration into libraries may prove beneficial. That said, very positive results were obtained and a high accuracy for the test set was achieved.

References

XGBoost library
Python library specifics: https://xgboost.readthedocs.io/en/stable/python/python_intro.html
Binary Classification example:
https://github.com/dmlc/xgboost/tree/master/demo/CLI/binary_classification
Experimental Categorical data:
https://xgboost.readthedocs.io/en/stable/tutorials/categorical.html
Parameters: https://xgboost.readthedocs.io/en/stable/parameter.html

Python Pandas
Read_csv method: https://pandas.pydata.org/docs/reference/api/pandas.read_csv.html
To_csv method: https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.to_csv.html
Dataframe: https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.html

Label-encoding
Options: https://pbpython.com/categorical-encoding.html
Scikit library encoding method: https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.LabelEncoder.html#sklearn.preprocessing.L abelEncoder

accuracy_score method:
https://scikit-learn.org/stable/modules/classes.html#module-sklearn.metrics and
https://scikit-learn.org/stable/modules/generated/sklearn.metrics.accuracy_score.html