PREDICTING INCOME POTENTIAL BY BUILDING CLASSIFIERS

Hong Chen[*]
chenhongjin2005@hotmail.com

Jing Li [*]
jliellen@my.yorku.ca

Haoran Zhou [*]
zhouk@my.yorku.ca

Amir Rahimi [*]
amirrah@my.yorku.ca

Huan Ho [*]
bruceho1@my.yorku.ca

Shiyi Du[*]
namesdu@my.yorku.ca

Zijiang Yan[*]
sunnyyzj@my.yorku.ca

Xiaoyu Yin [*]
consyinxy@gmail.com

Jialin Sun [*]
jialinsun1996@hotmail.com

Yi Dai [*]
vmou@outlook.com

December 27, 2020

[*] Lassonde School of Engineering, York University, Toronto, Canada

1 Introduction

This report is about predicting if the income of a certain US adult exceeds 50,000 per year with three classification models based on the UCI dataset of 1994 US adult income. Throughout the entire project, we communicated well and everyone brought up the concerns clearly and openly. The tasks and time-line could be found in **Table** 1.

Task(s)	**Exp. date**	**Responsible person**	**Act. date**	**Note**
Create group chat	09/12	Xiaoyu Yin	09/12	Xiaoyu, group contact-person
Choose dataset	10/12	everyone	10/12	Adult income dataset
Assign subgroups	10/12	everyone	10/12	three groups formed
Choose predictors	10/12	everyone	10/12	random forest, KNN, neural network
Data preprocessing	11/12	Shiyi Du	11/12	data preprocessed at supergroup level
Model training & testing	13/12	subgroups	14/12	obtain predictors
Sub-reports writeup	15/12	subgroups	16/12	add contents on overleaf
Ensemble building	16/12	Xiaoyu Yin	17/12	test the ensemble
Intro writeup	17/12	Haoran Zhou	17/12	
Dataset writeup	17/12	Amir, Zijiang	17/12	
First draft & proofread	18/12	everyone	18/12	
Report finalize	19/12	Jing Li	19/12	

Table 1: Project planning and execution.

- **Random forest**: Jing Li - hyperparameter tuning; Haoran Zhou and Hong Chen - model training and testing.
- **K-nearest neighbor**: Huan and Shiyi - training and testing model; Amir and Zijiang - documentation.
- **Neural network**: Xiaoyu and Jialin - further data preprocess; Jialin and Yi Dai - model training and tuning.

Some tasks of the project, such as model training and testing (we were one day behind the schedule), took longer than we expected. The main reason was some predictors including random forest were new to us, so it took time to learn the concepts. In addition to that, we had not worked with machine learning libraries before. As such, we spent some time looking up the tutorials to get familiar with models and libraries in Python. Furthermore, many of us did not have sufficient hardware so certain tasks took longer than expected. For example, one run of hyperparameter tuning for the random forest predictor took 37 minutes, which limited the number of times we could run the optimization. Last but not least, the project was during the final period, and some team members had exams meanwhile. To deal with potential delays, we made sure that at least one person in each subgroup could work on the project and other members could catch up later. Overall, we maintained the work flow and were able to complete the assignment without much delay.

The remainder of this report is organized as follows: **Section 2** provides details of the dataset and data preprocessing. **Section 3** presents the random forest model. The k-nearest neighbor is described in **Section 4**. **Section 5** presents the neural network model. We compare three models and build an ensemble in **Section 6**.

2 Dataset and Preprocessing

The Adult Data Set [1] is found from the UCI Machine Learning Repository, and it contains 32,561 instances in 'adult.data' (we only used this dataset and split 20 per cent of it as test data). The dataset has a multivariate characteristic. Approximately, 2,399 of the instances from the dataset have at least one or more missing values; therefore, the first preprocessing we did at super-group level was deleting the instances that had at least one missing feature value. The reason of excluding those data is that our dataset is relatively large.

The dataset is containing 14 features, and the types of features are either categorical strings or integer values. The output of the dataset are >50k and <=50k, which represented whether an individual had an income greater than 50,000 USD. The second step that we did at super-group level was to convert the categorical string attributes into numerical values, because 8 out of the 14 attributes in the dataset were in the type of string. After mapping the string values to numerical values, we permuted the data and split it into 80% training datapoints (24,129) and 20% testing datapoints (6,033). One of the challenges of our dataset was that the classes was imbalanced, as the number of instances that were <=50k was much higher than the instances with >50k. Therefore, we decided to sub-sample datapoints from the training data. At the end, we had 12,094 number of balanced training datapoints (50% of it for each class).

3 Random Forest

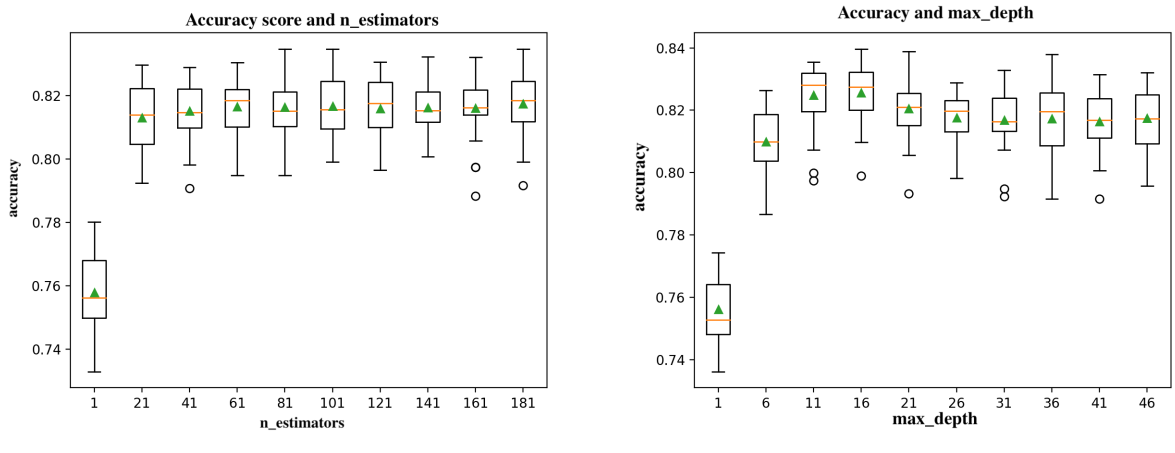
In this section, we predict whether a person's income is above 50k or below 50k through Random Forest, which is a widely used model in machine learning. Random Forest is an ensemble that consists of multiple decision trees in parallel [2]. We first tuned the hyperparameters using cross validation based on training data. Then we trained the model with the optimized parameters in random forest and evaluated the model based on testing data. We computed precision, recall, and f1-score in particular. Python 3.7 was used in all experiments and the predictor was trained and tested using scikit-learn library.

3.1 Hyperparameters tuning

Optimizing the hyperparameters is the key to the good prediction. As such, before we trained the model, we tuned the hyperparameters of random forest classifiers using cross validation [3]. The grid search approach was applied in our work. First, we created a hyperparameter grid that has different combinations of 5 hyper parameters of a random forest classifier: n_estimators, max_depth, min_samples_split, min_samples_leaf and and max_features. **Table** 2 shows the setting values of each hyper parameter for grid search. Particularly, [4] suggests that a range between 64 and 128 trees in a random forest works well regarding performance and running time. Thus, we experimented *n_estimators* in the range between 50 and 200. For other parameters, we consulted the analysis in [5]. Then, we employed the function *RandomizedSearchCV* to find the best hyperparameters by randomly searching. We used 10-fold cross validation, and 1000 different combinations are sampled. The best parameters we obtained from fitting the random search are shown in the last column in **Table** 2, with 82.5% accuracy on train data. The random grid search in Python could be realized:

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import RandomizedSearchCV
rf = RandomForestClassifier()
rf_random = RandomizedSearchCV(estimator=rf, param_distributions = random_grid, n_iter=1000, cv=10)
rf_random.fit(X_train, Y_train)
rf_random.best_params_
```

We also examined the effect of individual hyper parameters on model performance. **Figure** 1 shows how *n_estimators* and *max_depth* affect the model accuracy, respectively. We could clearly see that, with *n_estimators* increasing, the accuracy of the model sharply increases and then stagnates at a certain level, and *max_depth* has the similar property.



(a)

(b)

Figure 1: The effect of n_estimators and max_depth

3.2 Model training and testing

After obtaining the best hyper parameters of a random forest classifier fitting the training data, we trained the model using the training dataset and tested it using the testing dataset. Feature importance of our model is in **Figure** 2. We evaluated our model using accuracy, precision, recall and f1_score. The accuracy score we obtained was 80.34%. While

Parameters	Experimented Values	Best
n_estimators	50, 75, 105, 135, 160, 200	105
max_depth	2, 5, 10, 20, 40	20
min_samples_split	2, 10, 50, 100, 200	50
min_samples_leaf	1, 2, 4, 10	2
max_features	auto, sqrt	auto

Table 2: Hyper parameters experimented in Grid Search.

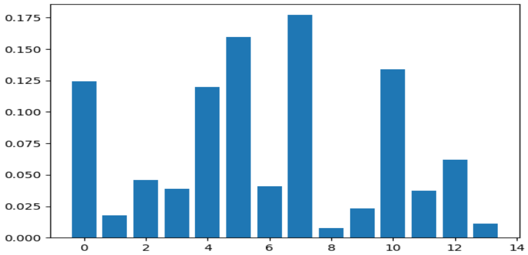


Figure 2: feature importance

the precision before and after tuning stayed similar, the recall increased from 83.3% to 86.31%, which means tuning helped reduce false negative. The model gave us the best value for f1_score of 68.16% after tuning its hyperparameters, while the score was 67.61% without tuning. The snippet of code is:

```
from sklearn.metrics import *
rf = RandomForestClassifier(n_estimators=105, max_depth=20, min_samples_leaf=2,
    min_samples_split=2, max_features='auto', random_state=42)
rf.fit(X_train, Y_train)
predicted = rf.predict(X_test)
accuracy = accuracy_score(y_test, predicted)
```

4 K-Nearest Neighbor

In this section, we used Python to train the K-nearest neighbor classifier on the dataset. The sklearn toolbox was used for implementing the KNN algorithm. We divided our methodology into three main stages: normalization approach, one-hot encoding approach with hyperparameter tuning, and genetic algorithm approach for finding a set of optimum parameters for KNN. We used 0/1 loss function for model evaluation. At the end, we decided to apply the hyperparameters we obtained from GA on a KNN classifier for the predictor.

4.1 Normalization

Since the original dataset may contain some extreme values, we decided to apply a normalization on the dataset. After the normalization, the bias should be reduced, and the model could have better accuracy [6]. The formula $y = \frac{x - x_{min}}{x_{max} - x_{min}}$ was used to transform the data, because the sample size was large, $Y \sim \mathcal{N}(0, 1)$. Then we used Euclidean distance for KNN with $K = 3$, and a brute-force algorithm to find k-nearest points. 10% of the training data was used for validation, and the rest of the training data was used to predict the validation data. Without normalization, the model had an accuracy of 61%. After the normalization, the model hit accuracy of 76.3% (the normalization was not used in the later stages).

4.2 One Hot Encoding

We split the training data (80% from the original data that we have agreed on in the super group) into a new training dataset and a validation dataset with the ratio of 4:1. Then we proceed on with the feature selection (especially the features with categorical values) and preprocessed the data with One Hot Encoding. We tuned hyper parameters on validation dataset. After that, we trained the model with a random permutation of the training data. We repeated this step 5 times for each combination of parameters ('n_neighbor', 'leaf_size' and 'weights'). The error was computed by averaging the total errors.

When training the model with the hyperparameters being tuned, we find that no model has the error less than 20%. In particular, when we selected Manhattan distance as the main distance function and 'kd_tree' as the main algorithm in KNN, we only achieved 62% and 77% accuracy in the validation and training phase, respectively. When we replaced Manhattan distance with Euclidean distance, we could achieve 62% and 74% as the best accuracy in the validation and training phase, respectively.

4.3 Genetic Algorithms

The KNN algorithm is slightly different from other traditional algorithms. Instead of learning from the training data, it only memorizes the data and tries to make predictions based on the votes of nearby points. This means that our goal is to find a set of parameters that we could tune, and try to optimize that set of parameters. And here's a set of parameters we wanted to tune (Note: one-hot encoding and normalization were not used in this part):

- The coefficient of the input features. Some features might have a much larger influence on the result than others, for example (x_1, x_2, x_3) might be transformed to $(a \times x_1, b \times x_2, c \times x_3)$ for better prediction.
- The coefficient of different distance functions. Since our feature is mixed with numeric values and categorical values, we decided to apply Euclidean distance on numeric values and Hamming distance on categorical values; then we added them up. This means we have 2 coefficient for Euclid and Hamming distance, for example, $Euclid(x, y) + Ham(x, y)$ might be transformed to $a \times Euclid(x, y) + b \times Ham(x, y)$ for better prediction.
- The k value. We also wanted to find the optimum k value that could give us the best result.

Adding them up, we had 17 discrete parameters we wanted to tune, which was perfect for GA. We coded the algorithm with all the common processes like Selection, Crossover, Mutation [7]. The gene pool consisted of integer values between [0, 9], and each chromosome consisted of 17 genes corresponding to 17 parameters. The population of each generation was 20 individuals; and for every new generation, we randomly selected around 4% data to remember and 1% data for validation. The fitness value of each individual was measured as correct rate on validation data. (Due to the page limit, more details of the procedure are described on our Github.)

After about 200 generations, we ended up with the following chromosome, with about 80% accuracy (75% - 85% during validation phase, and 79.5% on testing data), while we only had 60% accuracy using the default parameter (which was 1).

[3, 9, 0, 0, 9, 4, 2, 8, 5, 6, 6, 9, 2, 9, 3, 9, 7]

The red numbers are the coefficient for the features. Although it is unclear if the higher value is correlated to the higher importance, it is certain that 0 means this feature is absolutely useless in the model. Therefore, as can be seen, the third and fourth features do not affect the model. Note that the coefficient is applied before calculating Euclid distance while the coefficient of categorical features for Hamming distance is applied during the calculation; for example, (0, 1), (1, 1) may have a Hamming distance of $a \times 1 + b \times 0$. The blue numbers are the weights for 2 distance function. In our case, $result = 3 \times Euclid + 9 \times Hamming$. And the green number indicates the value for k. In our case $k = 2 \times 7 + 1$, which means it is always odd value.

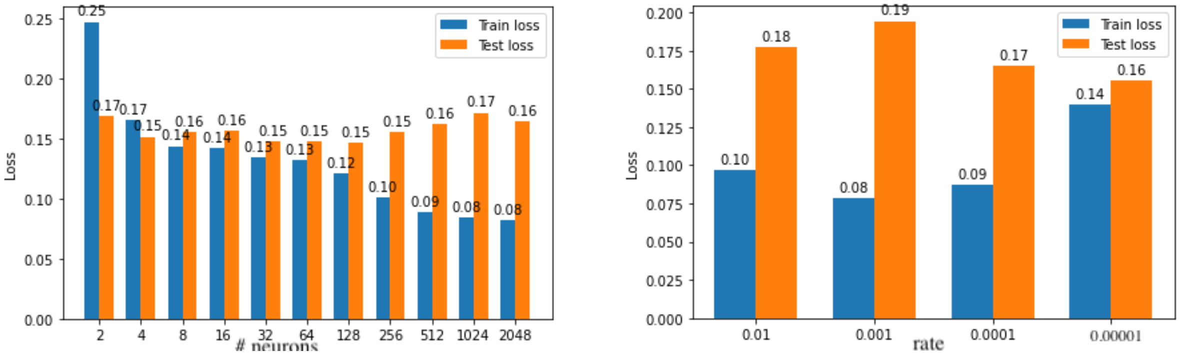
5 Neural Network

5.1 Additional data preprocessing

After obtaining the training and testing datasets from the super-group level, we spent some time to learn the property of the data. In order to have a better performance with neural network model, we needed two additional data preprocessing. In particular, we needed to decimal scale the continuous values [8] and one-hot encode the categorized values [9]. The reasons are as follows. First, we needed to decimal scale the continuous values, because the continuous values of some features that we were dealing with are at different scales. For example, *WorkingHoursPerWeek* and *EducationNum* have different scales, but, this does not mean the scales reflect their levels of importance. As such, we needed to perform decimal scaling to bring them to the same scale. Second, we needed to one-hot encode the categorized data. This is because the integer labels for categorized data only work in the situation that the categories actually "have a natural ordered relationship between each other and machine learning algorithms may be able to understand and harness this relationship", according to Brownlee [10]. However, our dataset does not have this property.

5.2 Model training and testing

We developed a pytorch skeleton of a neural network [11]. With this skeleton, we were able to easily train the model with different parameters such as learning rate, activation function and number of hidden layers. We used cross-validation to turn the hyperparameters, and **Figure** 3 presents the effects of neurons per layer and learning rate on model performance (Here we plotted both train data and test data, but, we only used train data in hyperparameters turning). we know that more hidden layers and more neurons per layer can improve the performance; at the same time, they also significantly affected the running time. It is a trade-off. And given that neural network could easily go overfitting, we finally chose 512 neurons per layer as hyperparameter. We used 0.001 learning rate in the model because it gave us the minimum loss, which is shown in **Figure** 3 (b).



(a)

(b)

Figure 3: The effect of # neurons and learning rate

The optimization algorithm we used was Adam, which could harness the power of adaptive learning rates methods to find individual learning rates for each parameter[12]. However, the effect of utilizing this function remains unclear because the effect is hard to visualize. The first activation function we selected is logistic. It yielded 0.2454 testing loss. We also tried other activation functions such as sigmoid and ReLu, of which ReLu had the lowest testing loss. We compared the training loss and the testing loss, the model did not have the overfitting problem. With 0.0001 learning rate, 512 neurons per layer, and two hidden layers in the ReLu activation function, the model achieved 77.408% accuracy.

6 Ensemble and Conclusion

We tested three predictors on the test data and reported the performance in each subsection. The accuracy scores were 79.43% (KNN), 80.34% (random forest) and 77.87% (neural network). Random forest had the highest accuracy. This might be because random forest is very effective at preventing overfitting and thus much more accurate [13].

We build an ensemble by assigning weights to three predictors, so ensemble becomes a voting system. The way how it works is straightforward. Let a, b and c where $a + b + c = 1$ denote the weights of predictors. For example, if the predicted labels of three predictors are 0, 1 and 1, the result of the voting system will be $a \times 0 + b \times 1 + c \times 1 = b + c$. If the result is greater than 0.5, then the ensemble will be a '1'. Otherwise, it will be a '0'.

An ensemble has the following advantage: in addition to the accuracy of individual predictor, it also leverages the performance of other predictors. As such, we assume that the average performance of the ensemble should be good. Meanwhile, the performance also depends on the weights that are assigned to predictors. We tried different weights and the accuracy scores can be found in **Table** 3.

KNN	**RF**	**NN**	**Accuracy**
0.3	0.5	0.2	79.33%
0.5	0.3	0.2	77.44%
0.2	0.7	0.1	80.37%
0.33	0.33	0.33	80.5%
0.2	0.2	0.6	77.87%

Table 3: Effect of different weights.

It shows that when 0.33 was assigned to each predictor, the ensemble (80.5% accuracy) outperformed any other individual predictor, which proved our assumption. What is also worth noticing is that the improvement was not significant. This might be because three predictors have fairly similar predictions. However, a small improvement could lead to a large number True cases given massive amounts of input data.

References

[1] UCI. Adult data set, 1996. data retrieved from UC Irvine Machine Learning Repository, `https://archive.ics.uci.edu/ml/datasets/Adult`.

[2] Siddharth Misra and Hao Li. Noninvasive fracture characterization based on the classification of sonic wave travel times. *Machine Learning for Subsurface Characterization*, page 243, 2019.

[3] Will Koehrsen. Hyperparameter tuning the random forest in python. [Online] Available from: `https://towardsdatascience.com/hyperparameter-tuning-the-random-forest-in-python-using-scikit-learn-28d2aa77dd74`, January 19 2018.

[4] Thais Mayumi Oshiro, Pedro Santoro Perez, and José Augusto Baranauskas. How many trees in a random forest? In *International workshop on machine learning and data mining in pattern recognition*, pages 154–168. Springer, 2012.

[5] Sharoon Saxena. A beginner's guide to random forest hyperparameter tuning. [Online] Available from: `https://www.analyticsvidhya.com/blog/2020/03/beginners-guide-random-forest-hyperparameter-tuning`, March 12 2020.

[6] DeFilippi Robert. Standardize or normalize? — examples in python. [Online] Available from: `https://medium.com/@rrfd/standardize-or-normalize-examples-in-python-e3f174b65dfc`, April 29 2018.

[7] John McCall. Genetic algorithms for modelling and optimisation. *Journal of computational and Applied Mathematics*, 184(1):205–222, 2005.

[8] Data normalization in data mining. [Online] Available from: `https://www.geeksforgeeks.org/data-normalization-in-data-mining`, June 25 2019.

[9] David Harris and Sarah Harris. *Digital design and computer architecture*. Morgan Kaufmann, 2010.

[10] Jason Brownlee. Why one-hot encode data in machine learning? [Online] Available from: `https://machinelearningmastery.com/why-one-hot-encode-data-in-machine-learning/`, July 28 2017.

[11] Defining a neural network in pytorch. `https://pytorch.org/tutorials/recipes/recipes/defining_a_neural_network.html`.

[12] Vitaly Bushaev. Adam — latest trends in deep learning optimization. [Online] Available from: `https://towardsdatascience.com/adam-latest-trends-in-deep-learning-optimization-6be9a291375c`, October 22 2018.

[13] Mark R Segal. Machine learning benchmarks and random forest regression. 2004.