EECS 4404: Project Report

(Machine Learning and Pattern Recognition)

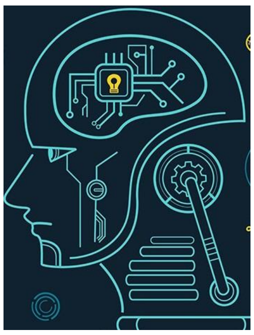

Table of Contents

PREPARED BY

Andrei Topala (210352144)
Arnesh Dawar (213721279)
Aziz Khan (215160120)
Chun Li (214908800)
Chu Yen Cao (215325343)
Muhammad Arifur Rahman (215858855)
Nicolas Jaramilli (211456639)
Nina Yanin (215850522)
Shovon Saha (215246473)

1. Introduction

To work on this project more efficiently, we divided the super-group into three subgroups each having three members. We chose to work on the Bank Marketing Data Set from the UCI repository [1]. We agreed to work on a random forest with XGBoost (subgroup 1), a NN (subgroup 2), and a Kernelized SVM (subgroup 3). We agreed on a train-test split of the data (80-20) using random permutations (random_state=42). We planned to finish the training of each model by December 18, 2020 and work on the evaluation of the performance and comparison between three predictors by December 19, 2020. Finally, we decided to describe our findings in our report on December 20, 2020.

Each subgroup was further divided into tasks based upon the individual's interest. For example, in subgroup 1, one member was responsible for implementation, another for optimal parameters / hyperparameters and one was responsible for reporting results. We arranged multiple Zoom meetings as well as messaging via Whatsapp to eliminate any confusion and to provide guidelines to subgroups on implementation. Our strategy to deal with any delays / bottleneck was to simplify the project or use different algorithms but we did not have to resort to this. Luckily, each subgroup finished their assigned tasks on time and our project finished as planned in the beginning. One contact person was chosen from the super-group to contact the course professor for any further clarifications.

2. Training of a XGBoost model by subgroup 1

2.1 Selecting the Predictor and toolbox

The group decided to train XGBoost on the bank dataset. XGBoost works by converting many weak-learners into a strong learner, which can then be further tuned and tweaked with the use of hyper parameters. XGBoost works by using a chain of small decision trees called ‘weak learners’. Each weak learner becomes a function where the tree is the function that takes an input and the leaves are the possible outputs. These leaves are then assigned a value of a weight, in order to improve performance. The weights are trained through gradient descent similar to gradient machines and what we learned in class. Additionally, XGBoost adds the regularizer that prevent the model from over fitting

2.2 Methodology

The boosting process begins by building trees. To begin with, we will introduce two hyper-parameters: λ and γ. These are part of the regularization $\gamma T + \frac{1}{2}\lambda\Sigma_{j=1}^{T} w_j^{2}$, λ is the regularizer while γ is used to prune the trees so they don’t overfit the data.

2.2.1 Data Exploration

While exploring the dataset we found the target values to be skewed as there are not many ‘yes’ values compared to the ‘no’ values. This can pose a problem as the model would form a bias against ‘yes’. As the yes class represents 13.24% of the data, the value of the mean square would be negligible.

We chose the Smote algorithm as it will increase the value of the smaller class and use F1 score as it provides a better model for skewed datasets. We also noticed the dataset to have missing data. We decided to use XGBoost as it is a great predictor for such scenarios. However we decided to decrease the feature space by using feature selection algorithm as it can increase the accuracy and speed of the process. We are trying to maintain the attributes which offer the highest information gain. We chose to use a decision tree method where we build a tree based and only keep the attributes chosen as a root.

2.2.2 Model Building

First, we split the data set the same way as the other groups with an 80 / 20 split. We used sklearn's RandomizedSearchCV to pick the hyper parameters of XGBoost such as gamma, learning rate, max_depth. We ran a 10 k-fold cross validation on the 80% training test.

Next, as mentioned before, we used the SMOTE algorithm to increase the value of the yes class and account for the skewness of the dataset.

Afterwards, we trained the model on the features we got from the decision tree which were ['age', 'balance', 'day', 'month', 'duration'].

2.3 Performance

The boosting process takes into account how the decision tree performs on each leaf node by placing emphasis on the predictors which perform well and tending to favor success. In the first case, Hyper Parameterization algorithm produced a F1 score of 0.8962398950072735. The Smote algorithm did not improve the performance of the model and the F1 score was 0.8855014502060525. And, finally the decision tree approach had the worst performance with a F1 score of 0.8799412634515239.

This gave us an idea. As XGBoost uses trees, what if we could use XGBoost for feature selection. After some research, we discovered that XGBoost does support feature selection and the features it selected where ['housing', 'contact', 'month', 'duration', 'pdays', 'poutcome']. By doing this we can use these features in the other models to see if they can improve the prediction or time of the other two algorithms, specially NN as they can take a long time to train.

3. Training of a NN model by subgroup 2

3.1 Selecting the predictor

We chose to take two different approaches to train a neural network: *scikit-learn*'s **MLPClassifier** and **Keras**.

3.2 Methodology

3.2.1 First approach - using sklearn library

We used following toolboxes to train our predictor:

- **neural_network.MLPClassifier**, a multi-layer perceptron classifier
- **preprocessing.OrdinalEncoder:** encode categorical features
- **Model_selection.train_test_split**: 20% testing and 80% training
- **model_selection.GridSearchCV:** test all specified hyperparameters and derive optimal solution
- **over_sampling.SMOTE:** handle bias in the data
- **preprocessing.StandardScaler:** standardize and center features to reduce variance

Once the data was preprocessed, we needed to get a sense of what parameter settings would yield the best result. We denoted all possible options for *solver*, *activation*, and *learning_rate* as they are closed sets. We also denoted three options for alpha and hidden_layer, received the results, updated the three options, and repeated the process. Having three options for these two hyperparameters allowed us to recognize towards which setting the model was leaning, and adjust the proposed options accordingly.

3.2.2 Second approach - using keras library:

We used the same technique to preprocess and encode the data, but instead of using GridSearch to look for the optimal parameters, we just tried different parameters and layers. We first drop the other features that are not important using feature selection from sklearn, standardize all the features, and train the following model:

Neural network - The sequential Model

- A 8-1 neural network
- The first hidden layer with input dimension 6 and relu activation
- The second hidden layer with sigmoid activation
- Loss function: binary cross entropy
- Optimizer: SGD with L2 Regularization
- 150 Epochs with batch size of 10

3.3 Training Performance

For our first approach, it seems that preprocessing the data yields a better predictor which can be seen in the table below. However, it is contingent on whether the bias was properly removed and whether there is no overfitting of data.

Results	solver	alpha	activation	hidden _layer	accuracy	MSE	f1_score
Preprocessed data	adam	0.0001	logistic	(150,)	0.943	0.058	0.942
Raw data	sgd	0.001	relu	(150,)	0.892	0.108	0.850

Table 1: Showing results of the algorithm ran on processed and unprocessed data

With Keras, we are able to visualize the training and testing process, also we are able to define the number of layers, activation function and regularizer on each layer. As a result, the neural network works best with 2 hidden layers with the accuracy of 89.78 and loss of 0.25.

To summarize, we noticed that the model performed better with more layers on the training data but performed really bad on the test data. This might be caused by the overfitting of the training set. Further, The number of epochs and batch size has a huge impact on how fast or how accurate the model will run. Finally, the feature selection helps speed up the model by disregarding features that have less impact on the result.

4. Training of a SVM model by subgroup 3

4.1 Selecting the Predictor and toolbox

The predictor we decided to train was a SVM using a Gaussian kernel. We choose to use **sklearn** that provides an implementation of this classifier via **sklearn.svm.SVC**. We felt using this kernel was a good choice due to the lack of knowledge about the data and domain.

4.2 Methodology

Before running the actual algorithm, we needed to preprocess the dataset. First, since SVM expects the features to be numerical as opposed to categorical, we transformed categorical values to numerical values using **sklearn.preprocessing.LabelEncoder().** In Addition, We converted manually our target values 'yes' and 'no' to 1 and 0 respectively.

Second, we standardized the dataset to avoid poor learning and domination by a single feature that has variance of higher magnitude than other features of the dataset [2]. For example, balance, duration and pdays have higher variance than other features. This standardization is especially important since we are using SVM with RBF Kernel and RBF Kernel presumes that all feature values are close to 0 and variances that are of same order [2]. To do this, we used **sklearn.preprocessing.StandardScaler().**

Finally, we extracted features to reduce the dimensionality and to remove redundant features that have less contribution to the learning process. We selected features that have strong correlations with the target values by evaluating the correlation matrix. We found that features housing, contact, month, duration, pdays, and poutcome have better correlations than other features and selected them for our features. We also evaluated Principle Component Analysis (PCA) with n_components = 6 and 8 for feature selection using **sklearn.decomposition.PCA**. However, we ran the algorithm with all features in the dataset to compare the performance with the dataset of selected features.

For selecting hyperparameters, we set the regularization parameter, C to discrete values of [0.01, 0.1, 1, 10, 100, 1000] and choose "gamma" within a log-uniform distribution between 1e-3 and 1e-1 using **loguniform** from **sklearn.utils.** The kernel chosen was 'rbf'. **RandomizedSearchCV** was used from **sklearn.model_selection** to search on hyperparameters for its better running time. To optimize these parameters, we applied 10-fold cross validation to our training dataset. Initially, we set the n-iterations to 20 and increased to 100 once the predictor converged.

4.3 Training Performance

We evaluated the performance of our predictor using the **f1_score** and **accuracy_score** from **sklearn.metrics**. The f1_score was around 0.88 for both the dataset with all features and the dataset with extracted features and the accuracy_score was around 0.89 for both of them with n_iterations = 20. However, we get f1_score around 0.89 and accuracy_score around 0.90 when we increased the n_iterations to 100.

Further, we did not notice any large change in the performance when selecting features using PCA or manually using the correlations. This might be due to the fact that features selection has less impact on the overall performance of the predictor in our dataset. To summarize, the f1_score was between 0.87 and 0.89 and the accuracy_score was 0.89 to 0.90 which we can conclude to say that it is a good performance achieved by our predictor.

5. Summary

For the XGBoost results, the group found that the tuning which provided the superior f1 score for predictions was when Hyper Parameterization was used, resulting in a value of 0.8962. Furthermore, XGBoost's capability of supporting feature selection was used in order to optimize the work of the other groups, specifically the neural network. When the neural network was run with preprocessed data which accounted for bias in the dataset, an f1 score of 0.942 was achieved which was the highest f1 score among all groups. Finally, the best SVM results came when n_iterations was set to 100, and this resulted in accuracy of 0.90 and an f1 score of 0.89.

Generally speaking, the accuracy and f1 scores did not vary significantly between different types of predictors or specific tweaks or tuning of each trained predictor. As the number of iterations or the amount of validation increased, all models performed better as expected. Some tweaks that the groups tried such as Group 1 implementing SMOTE to account for bias did not increase the performance and instead decreased the performance.

For the majority voting, the group decided to utilize the **sklearn**.ensemble.VotingClassifier. Unfortunately, the method took too long to run and the final result of the ensemble process was unable to be determined and reported on. The discussion will instead focus on how the ensemble method would work and what the expected results were. An ensemble method attempts to use the results from multiple predictors or algorithms in order to form a more likely or accurate prediction. A majority voting classifier attempts to predict using each of the given models and classifies according to the majority of the outputted predictions. In general, the combined output will have its variance reduced and therefore be considered more reliable or precise.

References

1. https://archive.ics.uci.edu/ml/datasets/Bank+Marketing
2. https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html