Application of Neural Network on Bank Marketing Dataset

Jun Wang 11238309
Mingliang Wei 11274244
Zihui Deng 11254652

December 10, 2020

Abstract

Our goal of the project is to build a neural network to identify potential buying customers in order to facilitate a successful bank direct marketing campaign. Our dataset was taken from UCI Machine learning repository which is related to a direct marketing campaings of a Portuguese banking institution. There are 41,188 instances in the dataset with 20 attributes. Through our experiment, we test on different hyperparameters and conclude that the best model is with 2 hidden layers and λ=10 which can achieve an accuracy of 91.5%. We will analyze the reasons behind the better performance and suggest possible future improvements.

1 Introduction and Overview

Due to economic pressures and competition, bank marketing managers had invest on directed campaigns such as phone calls, or e-mail contacts. Successful bank marketing campaigns rely on the use of a huge amount of customer electronic data. The contact are selected strict and rigorously in order to achieve a higher successful rate. To help with the decision-making process for the financial institutions, data mining models play an essential role in the performance of the marketing campaigns. Therefore, the objective of our team project is to build a neural network model to identify potential buying customers from bank direct marketing campaigns. The purpose of the model is to increase the campaign effectiveness by predicting whether the potential client would subscribe to a term deposit after the direct marketing campaigns.

Our dataset was taken from UCI Machine learning repository[1]. It is related to direct marketing campaigns of a Portuguese banking institution, and the marketing campaigns were based on phone calls. There are 41,188 instances in the dataset with 20 attributes in 4 categories. The attributes include:

- Information related to the client background such as age and job type;
- Information related to the social and economic context such as employment variation rate(numeric quarterly indicator) and consumer price index (numeric monthly indicator);
- Other information such as number of contacts performed during this campaign and for this client(numeric, includes last contact), and outcome of the previous marketing campaign(categorical);
- Since more than one contact to the same client was often required, there are attributes related to the last contact of the current campaign such as contact communication type(categorical), and last contact month of year(categorical).

Our goal is to predict the output variable – has the client subscribed a term deposit(binary: “yes”, “no”)

2 Methodology

The model we chose is neural network. There are potentially several advantages of neural networks to our dataset. First of all, neural networks are able to detect complex nonlinear relationships between dependent and independent variables, which is suitable for our dataset. Second, unlike many other prediction techniques, neural networks does not impose any restrictions on the input variables like how they should be distributed or scaled, so it's more convenient for data processing. Third. Neural networks can better model heteroscedasticity, which means data with high volatility and non-constant variance. Since neural networks are able to learn hidden relationships in the data without imposing any fixed relationships in the data[4].

2.1 The structure of neural networks

A simple neural network consists of an input layer, one or multiple hidden layers, and an output layer. Figure 1 figuratively demonstrates the structure of a neural network for binary classification with multiple hidden layers.

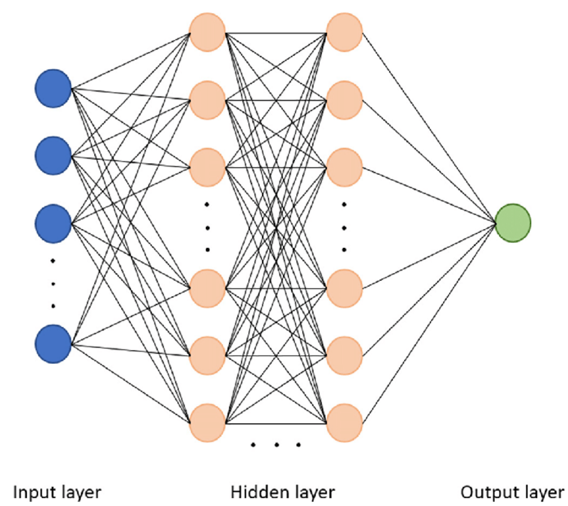

Figure 1: Example of a neural network for binary classification. Source:Dai et al., 2020[6]

The input layer is the input data, which means the independent variables.The hidden layers consists of one or multiple neurons.

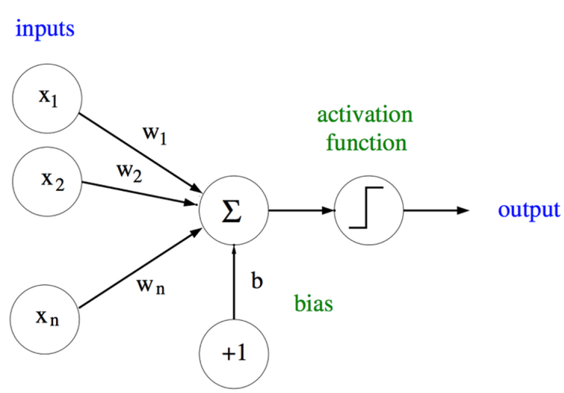

Figure 2: Example of a neuron in artificial neural network. Source:Galante and Banisch,2019[2]

Within each neuron, as shown in figure 2, the inputs will first go through a linear transformation: every input times a weight will be summed up and an addition of bias will be attached after the summation. Followed by the linear transformation is an activation function for nonlinear transformation. For multilayer neural networks, the output of the previous layer is the input of the next layer. Some common activation functions include:

- Sigmoid function: g(x) = 1/(1+exp(-x)) It maps the input to (0,1).
- Tanh function: g(x) = (1-exp(-2x))/(1+exp(-2x)) It maps the input to (-1,1).
- Rectified linear (ReLU) function: g(x) = max(0,x) It outputs the max between 0 and the input(after the linear transformation) and there's no upper bound.

The output layer is decided by the objective of the model. Linear units are for Gaussian output distributions, softmax units are for multinomial output distributions, and for Bernoulli output distributions(binary classification) like our project, we use a sigmoid unit.

2.2 Forward propagation

The forward propagation is the algorithm that takes the neural network and the initial input into the network and pushes the input through the network, it leads to the generation of an output hypothesis.

We use the cross-entropy cost function with an L2 regularized term to avoid over-fitting. Taking the negative log of the likelihood, the Cross-entropy error function with the L2 regularized term is defined as following:

$$J(\Theta) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log(h_\Theta(x^{(i)}))_k + (1-y_k^{(i)})\log(1-(h_\Theta(x^{(i)}))_k)\right] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(\Theta_{ji}^{(l)})^2$$

Figure 3: Corss-entropy function with an L2 regularized term

The first half of the function is the corss-entropy function. For m data points where yi is the true value 0 or 1 and $h_\theta(x)^{(i)}$ is the predicted probability for the ith data point. The second half of the function is the L2 regularized term. Increasing the lambda means increasing the regularization effect, but when it's too large, the model would underfit.

The process of the forward algorithm is therefore such: for each training example (x,y), calculate the output $h_\theta(x)^{(i)}$ based on current neural networks and the supervised loss with the regularized term: $L(h_\theta(x), y) + \lambda(\theta)^2$. We need to minimize the whole cost function which is the cross-entropy function plus the L2 regularization.

2.3 Back propagation

Back propagation uses a gradient descent algorithm. Basically it takes the output you got from your network, compares it to the real value (y) and calculates how wrong the network parameters were. It then, calculates that which way the weights should be altered so that the cost function can reach a minima. The process of the back propagation is as such: first calculate the gradients with respect to the parameters in each layer. Then back-calculate the errors associated with each unit from the preceding layer. This goes on until reaching the input layer.

These "error" measurements for each unit can be used to calculate the partial derivatives and We use the partial derivatives with gradient descent to try minimize the cost function and update all the weights.

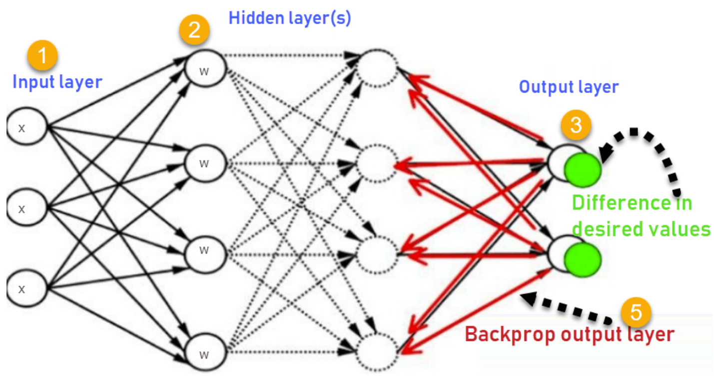

Figure 4: Back propagation. Source: Guru99[3]

3 Exploration Process

3.1 Data cleaning and pre-processing

Data cleaning is a crucial part before feeding model and pre-processing since any existence of outliers or missing value will make the interpretation meaningless and leading to incorrectness of the model.

There is no missing value or any outlier found in the data set after checking the basic information and box plots of it. But the response variable which represented as y in our model is **imbalanced**. The positive response represents that the client subscribed a term deposit only takes 11% of the whole data set whereas the negative response takes 89%. Therefore we applied two methods to balance positive and negative response in the training set which are "randomly under-sampling" and "randomly over-sampling".

While "randomly under-sampling" will randomly select a proportion of observations in the majority part and delete them, "randomly over-sampling" will randomly select some observations in the minority part(with replacement) and make repetitions of them. Both of the methods would be able to make a balance in the training data set in terms of the response (50% for each in our case). However, under-sampling could potentially make us lose some important information that should be used to improve the performance of the model, and over-sampling could lead to an possible over-fit in the minority part.

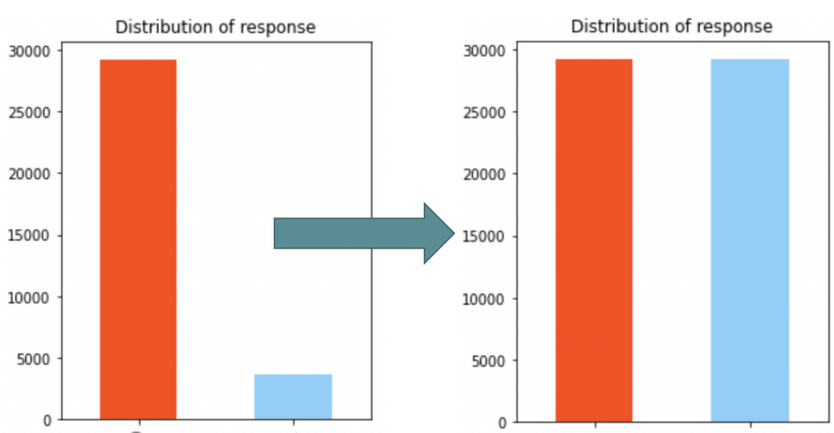

Figure 5: Distribution of response after over-sampling

In addition, there are 10 categorical variables in our data set in total. Before feeding into our neural network, we need to convert them into numerical variables. Since they are all nominal variables instead of having ordinal correlation between different levels within category, we will use **one hot encoding** here to create dummy variables where the number of dummies for each categorical variable will be equal to the number of different levels. After all, we will have 63 variables including 1 response(dependent variable), and then we are ready to feed the data to our neural network.

3.2 Neural network model details

The implementation of neural network is done with Python. The main goal is to build a neural network class object that can accomplish the tasks below[5]:

- **Task 1**: **This class object should be able to train the data set and get the performance of test set.** To be more specific, we defined 2 functions inside the neural network class object, "`get_cost`" and "`get_gradient`". The cross entropy with a regularization term is used to calculate the model cost; a forward and back propagation are preformed to calculate the gradient of our cost function.
- **Task 2**: **To improve efficiency.** Instead of using loop to calculate weights for every node on every layer, we choose to manipulate matrix to calculate weights on every layer, since matrix multiplication in "`Numpy`" (pre-compiled in optimized C code) are much more efficient in python than loop operation. We also use layer structure and lambda as input variable in function inside the class object, so it is easy to get the model cost and gradients with different weights to perform the optimization.
- **Task 3**: **This class object can be applied on different hyper-parameters (number of hidden layers, nodes and regularization parameters lambda).** Our solution is to save the weights matrix and their related size into a list so that we can call the list when needed. Besides, in the initialization of the class object, we define several variables that could be used frequently in the neural network class. For example, we store "`X`" as all input variables, "`y`" as output variables, "`num`" as an array of nodes in hidden layers, etc. The nomination makes our model flexible enough so that we can test our model and evaluate performance with different inputs of hyper-parameters.
- **Task 4**: **Necessary information should be stored for further performance analysis.** Our optimization process is done by using a python library – "`scipy.minimize`" with "`BGFS`" algorithm, which is an iterative method for solving unconstrained nonlinear optimization problems. The "`BGFS`" algorithm is a class of hill climbing optimization techniques that seek a local optimum with initial state given. Since we want to know if our algorithm works, we need to save the information in every iteration. The "`scipy.minimize`" algorithm allows us to save the new changed weights at every iteration, and with a "`save_step`" function with 2 global list variable, we are able to calculate accuracy with weights and save the accuracy level and time took in every iteration. It is also worth to mention that the "`scipy.minimize`" function only takes weights in type of "`ndarray`" with shape (n,). Therefore, it is essential to initialize weights into vector rather than into matrix, and then we can reshape the initial weights vector into weight matrix.

3.3 Performance comparison with different hyper-parameters

IT environment: 2.2 GHz Intel Core i7 with 4 cores and operated on Spyder.

As the minority class only takes up to 11.5% of the whole dataset, we can come up with our Naive model which can be used as a benchmark. Intuitively, this **Naive model** will predict the response in all of the test set as 0, so based on the distribution of the response, we can expect the accuracy to be around 88.5%.

Firstly, we set the hyperparameter $\lambda = 10$ arbitrarily and tried different numbers of hidden layers, then we can get the correlation between the accuracy and time consuming throughout the whole training process (as shown in Figure 6).

As we can see, the Neural network with two hidden layers acquires a higher accuracy after converge whereas the one with only one hidden layer can get to converge quicker than the complex one but with a lower accuracy. Also, after plugging into the validation set, the one with two hidden layers still performs better than the simpler one and got an accuracy of 91.8%, 91.22% on training set and validation set respectively.

The time consumed by both models are both around 70 seconds. Under the circumstance of a bank marketing decision, the manager does not need to train the data continuously and make an immediate decision, so the processing time of this algorithm is considered as reasonable. Since model with 2 hidden layers perform better in the training and validation set, we set hidden layers equals to 2 in following part.

Noted here that during the training process, we tried three types of training data which are the original one, the under sampling one and the over sampling one separately, however, the original one performs the best among all of them. Therefore, we conclude that it's probably because that the data imbalanceness in the dataset is not that extreme, so we decided to use the original dataset for the following part of the experiment to use the most of the information that is available.

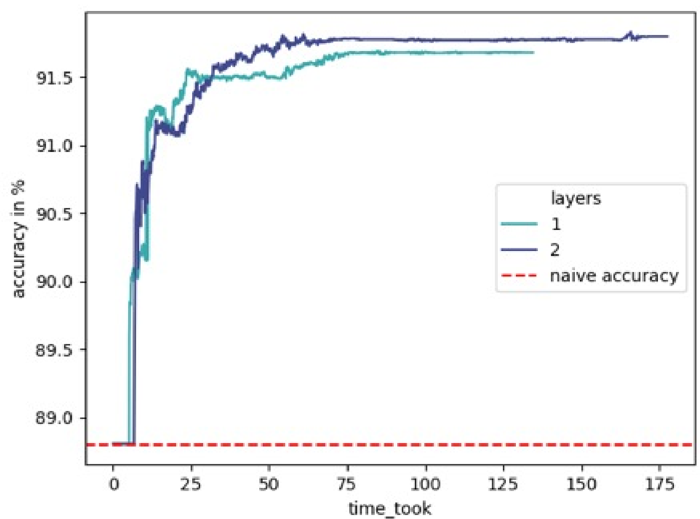

Figure 6: Neural network with different numbers of hidden layers

To further tune the hyperparameters and get a better performance, we will choose the λ in a set ranging from 0.001 to 100: $\lambda = [0.001, 0.005, 0.01, 0.05, 0.1, 0.5, 1, 5, 10, 50, 100]$. Then, as shown in Figure 7, we plot the accuracy of the training set(circles) and validation set(crosses).

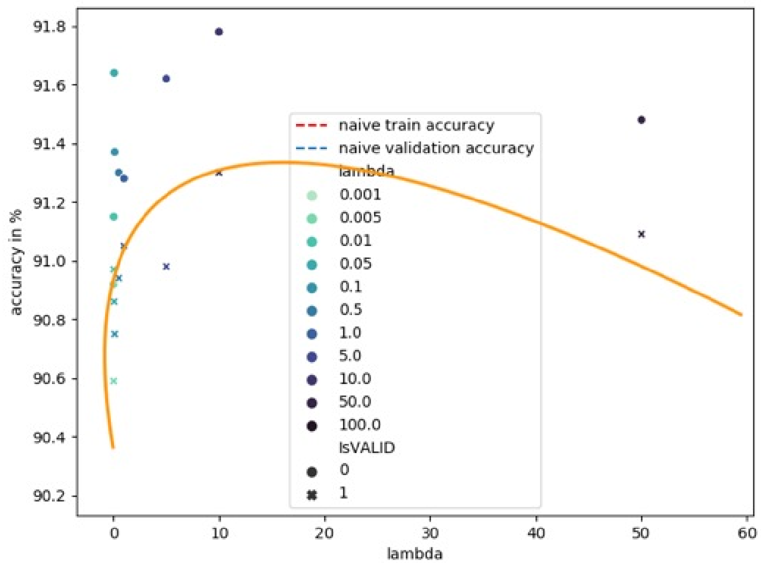

Figure 7: Train/Test accuracy with lambda from 0.001 to 100 [2 hidden layers]

As shown in Figure 7, it turns out that no matter what regularization parameter we chose, the training and validation accuracy are all above 88%. A downward parabola curve can also be possibly drew with the testing accuracy and we can expect to have the best settings of lambda between 1 to 30 interpreted from the graph.

Therefore, we took a finer set of lambda in the interval $[1, 30]$, and calculated the accuracy on the validation set.

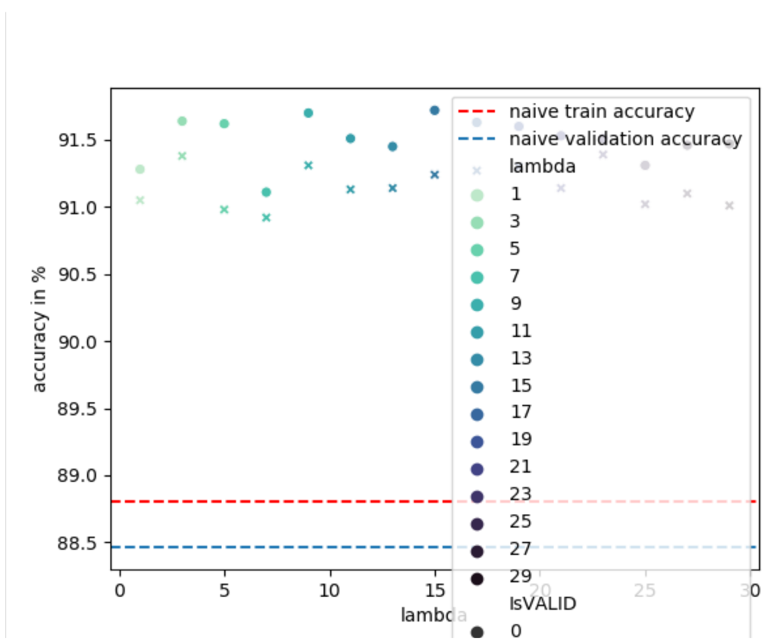

Figure 8: Test/Train accuracy with lambda from 1 to 30 [2 hidden layers]

As shown in Figure 8, the results were surprising: as the points of accuracy spread around a horizontal line, we cannot tell with which parameter the model would perform better out of sample. From another perspective, the linear shape of the accuracy shows little sensitive of the change in λ to the model. It is indeed a good property for a forecasting model as it indicates a lower risk that the model performs well in the training set while does not perform as well in the test set.

To trade-off between the performance and time consuming, we will choose lambda equals to 10 as our final parameter. In the process of choosing the hyper-parameters, there are indeed some parallel computation that could possibly be achieved in our algorithm. For example, the calibration of the hyper-parameters (number of nodes in each hidden layers, number of hidden layers, regularization parameters lambda) can be performed asynchronously using different computers.

4 Conclusions and Future work

4.1 Conclusions

The best performance of the model will be of 91.5% accuracy which is pretty good comparing to the naïve model we used. It only takes around 70 seconds for the training process [X: (39250,63), Y: (39250,1)] with a 4-cores 2.2 GHz Intel Core i7 computer.

After further exploration to the correlations between the inputs and outputs, and also the distribution of the response, we realized that the whole task is to classify those minority class within the population. Therefore, it might be more suitable to use a support vector machine (SVM) or other machine learning models to perform this task. Although the neural networks are robust to the noise, as it would mute those noise during the training process by decreasing the corresponding weights, we still think that a shrinkage method before training process such as using Lasso regression might help to remove the noise and improve the performance of the model.

Furthermore, although standardization will not influence the final performance too much, it can help to effectively reduce the time consumed for training process since it will make the corresponding weights converge quicker than without doing standardization.

4.2 Future work

As for the future work, since every set of weights is stored as a matrix, so all of the computation of derivatives on cost functions can be done simultaneously by using GPU for parallel computing or can be assigned as different computation tasks to different core within the CPU using parallel processing, which can both reduce the time consumed for the training process.

References

[1] Moro et al. *UCI Bank Marketing Data Set.* 2014. URL: https://archive.ics.uci.edu/ml/datasets/Bank+Marketing#.

[2] Luca Galante and Ralf Banisch. "A Comparative Evaluation of Anomaly Detection Techniques on Multivariate Time Series Data". PhD thesis. Jan. 2019. DOI: 10.13140/RG.2.2.18638.72001.

[3] *Guru99. How Backpropagation Works.* 2017. URL: https://www.guru99.com/backpropogation-neural-network.html.

[4] *Jahnavi Mahanta. Introduction to Neural Networks, Advantages and Applications.* 2017. URL: https://towardsdatascience.com/introduction-to-neural-networks-advantages-and-applications-96851bd1a207.

[5] Andrew Ng. *Lecture notes in Machine Learning*. URL: https://www.coursera.org/learn/machine-learning#about.

[6] Dai Xilei, Junjie Liu, and Xin Zhang. “A review of studies applying machine learning models to predict occupancy and window-opening behaviours in smart buildings”. In: *Energy and Buildings* 223 (May 2020), p. 110159. DOI: 10.1016/j.enbuild.2020.110159.

Appendix A Python Implementation

```python
# -*- coding: utf-8 -*-
# @author: Jun Wang, Mingliang Wei, Zihui Deng


# Import the library needed
import time
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from scipy.optimize import minimize
from imblearn.over_sampling import RandomOverSampler
from imblearn.under_sampling import RandomUnderSampler
from sklearn.model_selection import train_test_split


df=pd.read_csv("bank-additional-full.csv",sep=";")
df_test_final =pd.read_csv("bank-additional.csv",sep=";")

"""
No missing value found in dataset.**
There are 10 numeric inputs in total.**
Before feeding our Neural Network, we need to numeric those categorical variables in advance.
"""


df_dummy = pd.get_dummies(df, columns=["job", "marital", "education", "default", "housing",
                                       "loan", "contact", "month", "day_of_week", "poutcome", "y"],
                          prefix = ["job", "marital", "education", "default", "housing",
                                    "loan", "contact", "month", "day_of_week", "poutcome", "y"])
df_dummy = df_dummy.drop(['y_no'], axis = 1).rename(columns = {'y_yes':'y'})


X = df_dummy.drop('y', axis = 1)
y = df_dummy['y']

# split into train test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=2020)
y_train = np.array([y_train]).T
y_test = np.array([y_test]).T


#
print("The original distribution of training data response is: ", np.unique(y_train, return_counts=True) )
undersample = RandomUnderSampler(sampling_strategy='majority')
oversample = RandomOverSampler(sampling_strategy='minority')
# fit and apply the transform
X_train_under, y_train_under = undersample.fit_resample(X_train, y_train)
X_train_over, y_train_over = oversample.fit_resample(X_train, y_train)
print("The undersampling distribution of training data response is: ", np.unique(y_train_under, return_counts=True) )
print("The oversampling distribution of training data is: ", np.unique(y_train_over, return_counts=True) )


# distribution histogram for train set
pd.Series(y_train.flatten()).value_counts().plot(kind="bar",title="Distribution of response",color=['#FF4500','#87CEFA']
                        ,figsize=(4,5))
# distribution histogram for train set after over-sampling
```

```python
pd.Series(y_train_over.flatten()).value_counts().plot(kind="bar",title="Distribution of response",color=['#FF4500','#87CEFA']
                      ,figsize=(4,5))
# distribution histogram for test set
pd.Series(y_test.flatten()).value_counts().plot(kind="bar",title="Distribution of response",color=['#FF4500','#87CEFA']
                      ,figsize=(4,5))

df_test_final = pd.get_dummies(df_test_final, columns=["job", "marital", "education", "default", "housing",
                                       "loan", "contact", "month", "day_of_week", "poutcome", "y"],
                          prefix = ["job", "marital", "education", "default", "housing",
                                       "loan", "contact", "month", "day_of_week", "poutcome", "y"])
df_test_final = df_test_final.drop(['y_no'], axis = 1).rename(columns = {'y_yes':'y'})
df_test_final.head()



#%%
class NeuralNetwork():

  def __init__(self,input,output,num_nodes):
    # seeding for random number generation
    self.X = input.astype(float)
    self.y = output
    self.num = num_nodes # number of nodes for hidden layer (n2,n3,...,n(L-1))
    self.L = len(num_nodes) + 2 # total layers (include input and output)

    self.m = self.X.shape[0]  # number of training examples
    self.nx = self.X.shape[1]  # number of parameters

    # number of class in y
    if len(self.y.shape) == 1:
      self.ny = self.y.shape
    else:
      self.ny = self.y.shape[1]

    # number of parameters to estimate including numbers of bias units
    # i = current, j = next
    self.array_i = np.insert(self.num,0,self.nx)
    self.array_j = np.insert(self.num,len(self.num),self.ny)
    self.array_ntotal = (self.array_i+1) *self.array_j

    # number of parameters to estimate in total
    self.N = np.sum(self.array_ntotal)

  def rand_init(self,epsilon_init):
    # randomly initialize parameters to small values
    thetas = 2 * np.random.random((self.N,1))*epsilon_init - 1
    return thetas

  def get_thetas_without_bias_term(self,thetas):
    # sum should be np.sum(self.array_i * self.array_j)
    array_idx = np.cumsum(self.array_ntotal)

    # first
    thetas_without_bias = np.array(
        thetas[self.array_j[0]:array_idx[0],0])

    # then
    for idx in range(1,self.L-1):
      # exclude bias term
      thetas_ =  thetas[array_idx[idx-1]+self.array_j[idx]:array_idx[idx],0]
      thetas_without_bias = np.append(thetas_without_bias,thetas_)

    thetas_without_bias = np.reshape(thetas_without_bias,(-1,1))
```

```

    return thetas_without_bias

  # activation function
  def sigmoid(self,x):
    # Avoid overflow encountered in exp
    return np.exp(np.fmin(x, 0)) / (1 + np.exp(-np.abs(x)))

  # derivation of activation function
  def sigmoid_derivative(self,x):
    # computing the value of derivative to sigmoid function
    # g'(z_i) = g(z_i)*(1-g(z_i))
    return self.sigmoid(x) * (1 - self.sigmoid(x))


  # get cost function J_theta
  def get_cost(self,thetas,lambda_theta):

    if len(thetas.shape) == 1:
        thetas = np.reshape(thetas,(-1,1))

    # forward_propagation
    (a_L,lst_theta_i,lst_z_i,lst_a_i) = self.forward_propagation(thetas)

    # output = a_L
    output = a_L

    # regularization term
    thetas_without_bias = self.get_thetas_without_bias_term(thetas)

    # cost function
    J_theta = (
        - (1/self.m)*np.sum(self.y*np.log(output) + (1-self.y)*np.log(1-output))
        # add regulaized term
        + lambda_theta / (2*self.m) * np.sum(thetas_without_bias**2))
    return J_theta


  def get_gradient(self,thetas,lambda_theta):

    if len(thetas.shape) == 1:
        thetas = np.reshape(thetas,(-1,1))

    # forward_propagation
    (a_L,lst_theta_i,lst_z_i,lst_a_i) = self.forward_propagation(thetas)
    # backward_propagation to get the partial derivatives
    grad = self.back_propagation(a_L,lambda_theta,lst_theta_i,lst_z_i,lst_a_i)

    return grad

  def forward_propagation(self,thetas):
    array_idx = np.cumsum(self.array_ntotal)

    # forward_propagation
    a1 = np.concatenate((np.ones((self.m,1)),self.X),axis=1)
    theta1 = thetas[:array_idx[0]] # including the bias term
    theta1 = np.reshape(theta1,(self.array_i[0]+1,self.array_j[0]))

    # store z_i a_i theta_i
    lst_z_i = []
    lst_a_i = []

    lst_theta_i = []
    lst_theta_i.append(theta1)

    for i in range(1,self.L):
      if i == 1:
        # m * n2
```

```
        z_i = np.dot(a1,theta1) # z_2
        a_i = self.sigmoid(z_i) # a_2

        # num_next_layer * (num_current_layer + 1)
        # (n2,(n1+1))
        # theta_2
        theta_i = thetas[array_idx[i-1]:array_idx[i]]
        theta_i = np.reshape(theta_i,(self.array_i[1]+1,self.array_j[1]))
        lst_theta_i.append(theta_i)

        # add the bias unit
        a_i = np.concatenate((np.ones((self.m,1)),a_i),axis=1)

      else: # i = 2,3,4,...
        # hidden layers and output layer a_L
        z_i = np.dot(a_i,theta_i)
        a_i = self.sigmoid(z_i)

        if i < self.L -1:
          # only hidden layers
          theta_i = thetas[array_idx[i-1]:array_idx[i]]
          theta_i = np.reshape(theta_i,(self.array_i[i]+1,self.array_j[i]))
          lst_theta_i.append(theta_i)

          # add the bias unit
          a_i = np.concatenate((np.ones((self.m,1)),a_i),axis=1)

      # store z_i  a_i
      lst_z_i.append(z_i)
      lst_a_i.append(a_i)
    return (a_i,lst_theta_i,lst_z_i,lst_a_i)

  # computing the error used for back-propagation
  def back_propagation(self,output,lambda_theta,
                       lst_theta_i,lst_z_i,lst_a_i):
    # starting from output layer aL = y
    delta_L =  output - self.y

    # gradients
    gradients = []

    # backward look for errors = delta_i
    delta_iplus1 = delta_L
    for i in np.arange(self.L-1,1,-1): # i = l-1, l-2, 2
      # calculate gradient at layer i
      gradients_i = 1 / self.m * np.dot(np.transpose(delta_iplus1),lst_a_i[i-2])
      gradients.append(gradients_i)

      # parameters thetas at layer i
      thetas_i = lst_theta_i[i-1]

      # error for every layers
      # exclude bias term
      delta_i = np.dot(delta_iplus1,np.transpose(thetas_i[1:,:])) * self.sigmoid_derivative(lst_z_i[i-2])

      delta_iplus1 = delta_i

    # add gradient_1
    a1 = np.concatenate((np.ones((self.m,1)),self.X),axis=1)
    gradients_1 = 1 / self.m * np.dot(np.transpose(delta_iplus1),a1)
    gradients.append(gradients_1)

    # reverse order
    gradients.reverse()

    # unroll gradients
    for i in range(len(gradients)):
```

```
      gradients[i] = np.transpose(gradients[i])
      # add the regularization
      # exclude the bias term
      gradients[i][1:,:] = gradients[i][1:,:] + lambda_theta / self.m * lst_theta_i[i][1:,:]
      # put it into (n,1)
      gradients[i] = np.reshape(gradients[i],(-1,1))
    grad = np.ravel(np.concatenate(gradients,axis=0))

    return grad

  # accuracy
  def get_accuracy(self,theta):
    a_L = self.forward_propagation(theta)[0]
    # if a_L >=0.5, y_prct = 1
    # if a_L <0.5,  y_prct = 0
    y_prct = a_L>=0.5

    # accuracy in percentage %
    accuracy = (np.sum(self.y==y_prct) / len(y_prct))*100
    return accuracy

  # to save the accuracy rate for every step
  def save_step(self,k):
    global accuracy_steps
    accuracy = self.get_accuracy(k)
    accuracy_steps.append(accuracy)

    global time_steps
    time_took = time.perf_counter() - st
    time_steps.append(time_took)


  # train the neural network
  def train(self,lambda_theta,epsilon_init):
    # minimization with known gradient
    # objective function
    fun_cost = lambda thetas : self.get_cost(thetas,lambda_theta)
    fun_grad = lambda thetas : self.get_gradient(thetas,lambda_theta)

    # start point
    print("-"*80)
    print("Beginning Randomly Generated Weights: ")
    theta0 = self.rand_init(epsilon_init)
    print("theta0 size", theta0.shape)

    # minimization
    print("-"*80)
    print("Beginning training ---------- ")
    res = minimize(fun_cost, theta0, method='BFGS', jac=fun_grad,
                   callback = self.save_step,
                   options={'disp': True,'gtol': 1e-7, 'maxiter': 10})
    print("End training ---------------- ")

    return res.x

def get_step_accuracy_time(L_max,accuracy_steps,time_steps):
  l = 1
  lst = list(range(1,L_max+1))
  accuracy_steps_fn = {lst[i-1]:[] for i in lst}
  time_steps_fn = {lst[i-1]:[] for i in lst}

  accuracy_steps_fn[l].append(accuracy_steps[0])
  time_steps_fn[l].append(time_steps[0])

  for i in range(len(accuracy_steps)-1):
    if time_steps[i+1] < time_steps[i]:
      l = l+1

```

```
    accuracy_steps_fn[l].append(accuracy_steps[i+1])
    time_steps_fn[l].append(time_steps[i+1])

  return accuracy_steps_fn,time_steps_fn

def plot(L_max,accuracy_steps_fn,time_steps_fn,accuracy_naive):
  lst_df = []
  for i in range(L_max):
    df = pd.DataFrame({"accuracy in %":accuracy_steps_fn[i+1],
                       "time_took":time_steps_fn[i+1],
                       "layers":i+1})
    lst_df.append(df)

  df_fn = pd.concat(lst_df)

  # add accuracy level with naive prediction
  # all false
  plt.figure()
  palette = sns.color_palette("mako_r", L_max)
  sns.lineplot(data=df_fn, x="time_took", y="accuracy in %", hue="layers",palette=palette)
  plt.axhline(y=accuracy_naive,label="naive accuracy",ls="--",color="red")
  plt.legend()
  plt.show()


def plot_res(L_max,
             time_steps_fn_fn,dic_accuracy_train_fn,dic_accuracy_test_fn,
             accuracy_train_naive,accuracy_test_naive):
  lst_df = []
  df_train = pd.DataFrame({"accuracy in %":list(dic_accuracy_train_fn.values()),
                     "time_took":list(time_steps_fn_fn.values()),
                     "layers":np.arange(1,L_max+1),
                     "IsTest":0})

  df_test = pd.DataFrame({"accuracy in %":list(dic_accuracy_test_fn.values()),
                     "time_took":list(time_steps_fn_fn.values()),
                     "layers":np.arange(1,L_max+1),
                     "IsTest":1})


  lst_df.append(df_train)
  lst_df.append(df_test)
  df_fn = pd.concat(lst_df)


  # add accuracy level with naive prediction
  # all false
  plt.figure()
  palette = sns.color_palette("mako_r", L_max)
  sns.scatterplot(data=df_fn, x="time_took", y="accuracy in %", hue="layers",style="IsTest",
    palette=palette)
  plt.axhline(y=accuracy_train_naive,label="naive train accuracy",ls="--",color="red")
  plt.axhline(y=accuracy_test_naive,label="naive test accuracy",ls="--")
  plt.legend()
  plt.show()

def plot_lambda(lambda_thetas, dic_accuracy_train_fn,dic_accuracy_test_fn,
    accuracy_train_naive,accuracy_test_naive):
  lst_df = []
  df_train = pd.DataFrame({"accuracy in %":list(dic_accuracy_train_fn.values()),
                     "lambda":list(dic_accuracy_train_fn.keys()),
                     "IsTest":0})

  df_test = pd.DataFrame({"accuracy in %":list(dic_accuracy_test_fn.values()),
                     "lambda":list(dic_accuracy_train_fn.keys()),
                     "IsTest":1})

  lst_df.append(df_train)
```

```
  lst_df.append(df_test)
  df_fn = pd.concat(lst_df)


  # add accuracy level with naive prediction
  # all false
  plt.figure()
  palette = sns.color_palette("mako_r", len(lambda_thetas))

  sns.scatterplot(data=df_fn, x="lambda", y="accuracy in %", hue="lambda",style="IsTest",
    palette=palette)
  plt.axhline(y=accuracy_train_naive,label="naive train accuracy",ls="--",color="red")
  plt.axhline(y=accuracy_test_naive,label="naive test accuracy",ls="--")
  plt.legend()
  plt.show()

def find_best_lambda(lambda_thetas):
  dic_accuracy_train_fn = {lambda_thetas[i-1]:[] for i in range(len(lambda_thetas))}
  dic_accuracy_test_fn = {lambda_thetas[i-1]:[] for i in range(len(lambda_thetas))}

  for lambda_theta in lambda_thetas:
    # Initialization of neural network
    neural_network_train = NeuralNetwork(X_train, y_train,num_nodes)
    # train
    thetas_res =  neural_network_train.train(lambda_theta,epsilon_init)
    accuracy_train = neural_network_train.get_accuracy(thetas_res)
    dic_accuracy_train_fn[lambda_theta] = np.round(accuracy_train,2)
    # test
    neural_network_test = NeuralNetwork(X_test, y_test,num_nodes)
    accuracy_test = neural_network_test.get_accuracy(thetas_res)
    dic_accuracy_test_fn[lambda_theta] = np.round(accuracy_test,2)

  # print final accuracy
  print("-"*80)
  print("train accuracy == \n",dic_accuracy_train_fn, " in %")
  print("test accuracy == \n",dic_accuracy_test_fn, " in %")

  plot_lambda(lambda_thetas,
              dic_accuracy_train_fn,dic_accuracy_test_fn,
              accuracy_train_naive,accuracy_test_naive)


#%%
if __name__ == "__main__":

  """
  inputs
  """
  # parameters to make thetas_init small,
  epsilon_init = 0.12

  # max number of hidden layers
  L_max = 2

  # regularization parameter
  lambda_theta = 10

  """
  choice of number of layers
  """

  # number of nodes in the hidden layers
  matrix_num_nodes = np.tril(np.ones((L_max,L_max)) + 3)
  matrix_num_nodes = matrix_num_nodes.astype(int)
  print("Inputs: ")
  print("matrix of hidden layers nodes \n", matrix_num_nodes)

  # store accuracy of every iteration
```

```python
  accuracy_steps = []
  time_steps = []

  l = 1
  lst = list(range(1,L_max+1))
  dic_accuracy_train_fn = {lst[i-1]:[] for i in lst}
  dic_accuracy_test_fn = {lst[i-1]:[] for i in lst}

  for L in range(L_max):
    num_nodes = matrix_num_nodes[L,:]
    num_nodes = num_nodes[num_nodes != 0]
    print("-"*80)
    print("-"*80)
    print("parameter set: ")
    print("hidden layers: ",num_nodes)

    # Initialization of neural network
    neural_network_train = NeuralNetwork(X_train, y_train,num_nodes)


    # Perform neural network
    st = time.perf_counter()
    thetas_res =  neural_network_train.train(lambda_theta,epsilon_init)

    # train accuracy
    accuracy_train = neural_network_train.get_accuracy(thetas_res)
    dic_accuracy_train_fn[l] = np.round(accuracy_train,2)

    # test accuracy
    neural_network_test = NeuralNetwork(X_test, y_test,num_nodes)
    accuracy_test = neural_network_test.get_accuracy(thetas_res)
    dic_accuracy_test_fn[l] = np.round(accuracy_test,2)

    l = l+1

  # naive accuracy
  accuracy_train_naive = round((1 - np.sum(y_train)/y_train.shape[0])*100,2)
  accuracy_test_naive = round((1 - np.sum(y_test)/y_test.shape[0])*100,2)

  # print final accuracy
  print("-"*80)
  print("train accuracy == \n",dic_accuracy_train_fn, " %")
  print("test accuracy == \n",dic_accuracy_test_fn, " %")

  # steps accuracy and acculated time for every step
  accuracy_steps_fn,time_steps_fn = get_step_accuracy_time(
      L_max,accuracy_steps,time_steps)

  time_steps_fn_fn = {l:t[-1] for (l,t) in time_steps_fn.items()}

  # plot
  plot(L_max,accuracy_steps_fn,time_steps_fn,accuracy_train_naive)
  plot_res(L_max,time_steps_fn_fn,dic_accuracy_train_fn,dic_accuracy_test_fn,
           accuracy_train_naive,accuracy_test_naive)

  a = np.array(list(dic_accuracy_test_fn.items()))[:,1]
  np.mean(a - accuracy_test_naive)


#%%
  L = 1
  num_nodes = matrix_num_nodes[L,:]
  num_nodes = num_nodes[num_nodes != 0]

  """
  final test
  df_test_final
  """
```

```
  X_test_fn = df_test_final.loc[:,df_test_final.columns != 'y']
  y_test_fn = np.reshape(np.array(df_test_final['y']),(-1,1))
  accuracy_test_fn_naive = round((1 - np.sum(y_test_fn)/y_test_fn.shape[0])*100,2)

  # final test accuracy
  neural_network_train = NeuralNetwork(X_train, y_train,num_nodes)
  thetas_res =  neural_network_train.train(lambda_theta,epsilon_init)
  neural_network_fn = NeuralNetwork(X_test_fn, y_test_fn,num_nodes)
  accuracy_test = neural_network_fn.get_accuracy(thetas_res)
  print("final test accuracy == \n",np.round(accuracy_test,2), " %")



#%%
  """
  choice of regularization parameter
  """
  # we choose hidden layers = 2, max_iter = 1000
  L = 1
  num_nodes = matrix_num_nodes[L,:]
  num_nodes = num_nodes[num_nodes != 0]

  lambda_thetas = [0.001,0.005,0.01,0.05,0.1,0.5,1,5,10,50,100]
  find_best_lambda(lambda_thetas)

  # finer discretization
  # lambda between 1 - 50
  lambda_thetas = list(np.arange(1,30,2))
  find_best_lambda(lambda_thetas)




  #%%
  lambda_theta = 10
  y_train_under = np.reshape(y_train_under,(-1,1))
  neural_network_train_undersample = NeuralNetwork(X_train_under, y_train_under,num_nodes)
  thetas_res_under =  neural_network_train_undersample.train(lambda_theta,epsilon_init)
  accuracy_train_under = neural_network_train_undersample.get_accuracy(thetas_res_under)
  print("-"*80)
  print("train accuracy undersampling == ",np.round(accuracy_train_under,2), "%")
  neural_network_test_under = NeuralNetwork(X_test, y_test,num_nodes)
  accuracy_test_under = neural_network_test_under.get_accuracy(thetas_res)
  print("test accuracy undersampling == ",np.round(accuracy_test_under,2), "%")
```

Listing 1: Application of Neural Network on Bank Marketing Dataset