Report for FYS-STK4155: Project 2

Maksym Brilenkov

November 2019

Abstract

In this project, I was working on both classification and regression problems. First part of the project is dedicated to comparison of the Logistic Regression and Feed Forward Neural Network (NN) performances via classification problem on the Default Credit Card Data set [1]. In the second part I apply created NN to fit the Franke function described in the first project Ref [2] and compare the results with previous codes written during Project 1.

Contents

1 Introduction

This project is dedicated to one of the most popular topic nowadays - *Artificial Neural Networks* (NN). In its core, NN is a set of mathematical algorithms which helps us to solve a specific problem. This problem doesn't have specific nature and can be anything from voice/image recognition to stock market prediction. However, in this project, I will address two types of tasks typical to supervised learning - classification and regression. The main difference between them is that first one deals with the set of discrete variables (and most of the time concerned with the outcome of the type - true/false, positive/negative, yes/no), while the second one works with continuous ones (with the outcome limited by the shape of the function we are trying to fit). However, regardless of the types of variables, the mathematical treatment is practically the same and it is based on the same idea - minimization of the so-called *Cost Function*. This is the basis of *Feed Forward Neural Networks* (FFNN).

For classification task, I have chosen to work with *Default Credit Card* data set originally analyzed by [3]. It contains information about gender, age, education, marital status, payment history, payment amount history, history of bill payments and the default history on 30000 clients for the period of several months (starting from April 2005 till September 2005). The idea is to build the model which will be able to predict the probability of a default with high accuracy. For regression task, my aim is to fit the Franke function [4] and compare the results with the ones obtained during the first project [2]

The report organized as follows. In section 2, I briefly sintroduce the concept of NN and how to train them. In section 3, I explain how I implemented the algorithms and also who the results of the pipeline run. The codes are listed in the Appendix.

2 Artificial Neural Network

In this section I describe the ideas behind Artificial Neural Networks and briefly state training algorithm and some problems which can be encountered during its implementation.

2.1 What is NN?

Dr. Robert Hecht-Nielsen [5] defines it as: "*a computing system made up of a number of simple, highly interconnected processing elements, which process information by their dynamic state response to external inputs*". Similarly to the **neurons** in human brain, which are connected via **synapses** and communicate through electrical signals, NN are made up of *neurons* (also called *unit* or *node*), which are sitting in a specific *layer* and receive the input from all the neurons of previous layers, process it and outputs processed signal to subsequent layer. Mathematically we can think of it as mapping one function into another one.

There are several types of Neural Networks [6]:

- *Feed Forward* (FFNN - all from one layer are connected to all neurons from subsequent layer; the information passes only forward through all the layers. Often, there are only 3 layers in total: the *input* layer, the *hidden* layer and the *output* layer. However, the amount of hidden layers can be larger with nonlinear activation functions. In this case the NN is called *Multi Layered Perceptron* (MLP)
- *Recurrent* (RNN) - similar to FFNN, but here information is passed in cycles.
- *Convolutional* (CNN) - each neuron in a layer is connected only to a subset of the nodes in the previous layer. Often, CNNs consist of several convolutional layers that learn local features of the input, with a fully-connected layer at the end, which gathers all the local data and produces the outputs.
- NN for unsupervised learning (*Deep Boltzmann Machines*)

To analyze the data sets I have briefly described in introduction, I will be using MLP, which mathematically can be described as [6]:

$$y_i^l = f^{l+1}\left[\sum_{j=1}^{N_l} w_{ij}^3 f^l\left(\sum_{k=1}^{N_{l-1}} w_{jk}^{l-1}\left(...f^1\left(\sum_{n=1}^{N_0} w_{mn}^1 x_n + b_m^1\right)...\right) + b_k^2\right) + b_1^3\right], \tag{2.1}$$

where N_l is the number of nodes in the layer l, f is the activation function, w are weights and b are biases for each layer. x_n is the inputs (and the only independent variable). The output is calculated by a series of matrix-vector multiplications and vector additions that are used as input to the *activation functions*.

But why do we need activation functions in the first place? It is all due to the *universal approxamation theorem*, which states that [6]: "*a feed-forward network with a single hidden layer containing a finite number of neurons can approximate continuous functions on compact subsets of real functions*". This means that if we have some input (represented by variable x) and output (represented by y) and we believe that there exists some unknown connection between them ($y = f(x)$), we can approximate this connection via series of non-linear transformations (2.1). therefore, the choice of activation function is very important for a given NN and is inferred from the following restrictions [6]:

- Non-constant
- Bounded
- Monotonically-increasing
- Continuous

Activation functions is a part of neural network architecture and are specified before starting the network training. The most popular choices for activation functions are:

- Sigmoid (also called logistic)
- Hyperbolic Tangent
- Softmax
- Rectified Linear Unit (ReLU)
- Leaky ReLU
- Parametric ReLU

Activation function, together with total amount of layers and the number of neurons per layer describes the network *architecture*. The choice of the architecture is crucial and varies from task to task. However, for the most of the problems, 1 input, 1 hidden and 1 output layer is enough. Such NN is called *two-layer network* by convention [6]. Once the architecture is established, it is time to start training NN.

2.2 Training NN

As in Linear Regression case[2], we are interested in fitting the unknown function $f(x)$, whose values, given some points, we know. As I have written previously, the fitting procedure is

based on the approximation of this function as (in matrix notation):

$$\mathbf{z} = \mathbf{W}^T \mathbf{x} + \mathbf{b}, \tag{2.2}$$

where parameters are the ones we are interested in.

We start training our network by subsequently applying transformation of the form (2.1) on each node while propagating through each layer. This approach is called *Forward Propagation.*

But here the story is not finished - we have only approximated our function, while our goal is the same as in the project 1 [2], i.e. to find weights and biases. Similarly to the previous scenario, we are going to define *Cost function* and find the parameters which are minimize it. Since there are two problems I am considering during this project, I am going to define two different cost functions - one for classification and another one for regression.

The *binary cross entropy* defined as [6]

$$J = \sum \left(y_i \log a_i + (1 - t_i) \log (1 - a_i) \right), \tag{2.3}$$

will be used for classification, whereas familiar *Mean Square Error* [6]

$$J = \frac{1}{n} \sum \left(a_i^2 - y_i^2 \right), \tag{2.4}$$

will be used for regression. a_i are the values of the of the function after applying activation function into it. Mathematically speaking:

$$a_i^l = \frac{\exp(z_i^l)}{1 + \exp(z_i^l)}, \quad z_i^l = \sum w_{ij}^l a_j^{l-1} + b_i^l, \tag{2.5}$$

where I have used *sigmoid activation function.*

In the next subsection I am going to discuss various methods used to minimize this function.

2.2.1 Optimization Methods

Batch Gradient descent(GD) algorithm is an iterative process that takes us to the minimum of a function, and the direction of the fastest descrease will be the direction of the negative gradient $-\nabla J(\mathbf{w})$. So, we start with some initial guess for $\mathbf{w}_0$ for a local minimum of $J(\mathbf{w})$ and

then update the value of $\mathbf{w}_n$ according the rule:

$$w := w - \alpha \nabla J(w), \quad J(w) = \frac{1}{n} \sum_{i=1}^{n} J_i(w), \quad n \geq 0, \tag{2.6}$$

We iterate this function over the entire data set and for given amount of epochs.

This approach may not be very efficient if we have say millions of data points. It can be very troublesome to calculate gradient in this case; thus, we can use an approximation to it on a single example (x_i, y_i are the training example and corresponding label):

$$w := w - \alpha \nabla J_i(w, x_i, y_i), \tag{2.7}$$

This algorithm is called *Stochastic Gradient Descent* (SGD) and it performs the above update for each training example. The data usually shuffled to prevent application of the algorithm on the same data.

This algorithm essentially jumps to a new (potentially better) local minima, while simple GD converges to the minimum of basin, where our parameters have been placed. This is an advantage. However, there is a risk of overshooting the global minima, if the learning rate is not tuned properly. That is why, it is better to use another approach, which calls *Mini-Batch Gradient Descent* (MBGD).

MBGD takes the best of the two above and performs an update for every mini-batch of n training examples:

$$w := w - \alpha \nabla J_i(w, x_{i;i+n}, y_{i;i+n}), \tag{2.8}$$

In this way it

- reduces the variance of the parameter updates; thus, we get better convergence;
- can make use of highly optimized matrix optimizations common to state-of-the-art deep learning libraries that make computing the gradient *w.r.t.* a mini-batch very efficient

The sizes of Mini-Batches vary from application to application, but it is common practice to use the some power of two, such as: 2, 4, 8, 16, ..., 256 and so on. This algorithm is typical choice then we deal with NN, however it also has its drawbacks:

- It does not guarantee good convergence - we can be trapped in the local minima and stay there for good.
- The choice of learning rate α may be very difficult. If α is too small, we will get slow convergence, while large α values can hinder convergence or even cause divergence;

- We apply the same learning rate to all weights. If our data is sparse and our features have very different frequencies, we might not want to update all of them to the same extent, but perform a larger update for rarely occurring features.

That is why we are tempted to use even more profound algorithms (I will drop $x_{i;i+n}$, $y_{i;i+n}$ notation for simplicity).

Momentum [7] is a method that helps accelerate SGD in the relevant direction and dampens oscillations. To make such algorithm work, we add a fraction γ of the update vector of the past time step to the current update vector:

$$v_t = \gamma v_{t-1} + \alpha \nabla_w J(w)\,, \tag{2.9}$$

$$w = w - v_t\,, \tag{2.10}$$

The momentum term is usually set to $\gamma = 0.9$.

When we are using moment it is like pushing the ball down the hill - it will accumulate momentum and will move faster and faster until it reached its terminal velocity if there is air resistance, i.e. $\gamma < 1$. on practive, our momentum term momentum term increases for dimensions whose gradients point in the same directions and reduces updates for dimensions whose gradients change directions. Consequently, we are getting faster convergence and reducing oscillation.

However, in this case we cannot control our ball, which is not very good thing. Thus, we need to think of something even better.

Nesterov accelerated gradient (NAG) [8] is a way to give our momentum term the kind of prescience of the direction it needs to move in. here, instead of computing γv_{t-1}, we will compute $w - \gamma v_{t-1}$, which will give us an approximation of the next position of the parameters. this means, that we know compute the gradient not *w.r.t.* w, but with the approximate future position of the parameters:

$$v_t = \gamma v_{t-1} + \alpha \nabla_w J\left(w - \gamma v_{t-1}\right), \tag{2.11}$$

$$w = w - v_t\,, \tag{2.12}$$

However, what if we want not only this, but also to adapt our updates to each individual parameter to be able to perform larger or smaller updates depending on their importance.

Adagrad [9] is an algorithm for gradient-based optimization that adapts the learning rate to the parameters, performing smaller updates (i.e. low learning rates) for parameters associated with frequently occurring features, and larger updates (i.e. high learning rates) for parameters associated with infrequent features.

In its update rule, Adagrad modifies the general learning rate α at each time step t for every parameter w_i based on the past gradients that have been computed for w_i:

$$w_{t+1,i} = w_{t,i} - \frac{\alpha}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i}, \quad g_{t,i} = \nabla_w J(w_{t,i}), \tag{2.13}$$

where $G_t \in \mathbb{R}^d \times d$ here is a diagonal matrix where each diagonal element $i, i4$ is the sum of the squares of the gradients *w.r.t.* w_i up to time step t [12], while ϵ is a smoothing term that avoids division by zero (usually on the order of 10^{-8}).

In vectorized form:

$$w_{t+1} = w_t - \frac{\alpha}{\sqrt{G_t + \epsilon}} \odot g_t, \tag{2.14}$$

Now we do not need to care about tuning the learning rate α.

As it is seen from equation (**??**), the main weakness of this approach is accumulation of the squared gradients in the denominator, which cause learning rate to become very small, which results in algorithm not acquiring any knowledge.

There are a lot of other methods and improvements, e.g.:

- Adadelta
- RMSProp
- Adam
- AdaMax
- Nadam
- AMSGrad

Each and one for them has its own strengths and weaknesses, but their exact formulations are beyond the scope of this report.

Regardless of the algorithm we chose, the gradients are calculated through the process called *Back propagation*. The idea is that once we know the values of a_i for each layer, we can propagate backwards to learn the error associated with each layer and thus calculate the gradients of the cost function.

So, if we have a values of a_i, which corresponds to the last layer L, we can write the gradient of the cost function as follows [6]:

$$\frac{\partial J}{\partial w_{jk}^L} = \left(a_j^L - t_j\right) \frac{\partial a_j^L}{\partial w_{jk}^L}, \quad \frac{\partial a_j^L}{\partial w_{jk}^L} = a_j^L \left(1 - a_j^L\right) a_k^{L-1}, \tag{2.15}$$

Defining the error as:

$$\delta_j^L =_j^L \left(1 - a_j^L\right)\left(a_j^L - t_j\right), \tag{2.16}$$

With the definition of δ, we now can write the derivative in the more compact form:

$$\frac{\partial J}{\partial w_{jk}^L} = \delta_j^L a_k^{L-1}, \tag{2.17}$$

hence we obtained the value of ∂J for the last layer, L. It is not hard to show [6] that the derivative of all intermediate levels can be found as:

$$\delta_j^l = \sum_k \delta_k^{l+1} w_{kj}^{l+1} f'(z_j^l), \tag{2.18}$$

where $f'(z_j)$ is the derivative of the activation function.

With this, the update rule for simple GD algorithm can be written as follows:

$$w_{jk}^l := w_{jk}^l - \alpha \delta_j^l a_k^{l-1}, \tag{2.19}$$

$$b_j^l := b_j^l - \alpha \delta_j^l, \tag{2.20}$$

With this in mind, we are ready to go and to write the NN. However, there is one more thing worth mentioning and it is weights initialisation.

2.2.2 Weights initialisation

Weights initialisation is a very crucial step for every NN network. Badly pre-configured weights may result in variety of problems such as *vanishing* and *exploding gradient problems*. Basically, the first one is encountered when the gradient becomes really small (smaller then numerical values stored in computer memory), while the second one is when gradients become extremely large.

For quite some time, the most popular weights initialisation technique was sampling from normal distribution with zero mean and unit variance. However, it was explicitly shown that such initialisation can lead to the exploding gradient problems. *Xavier et al* [10] showed that it is practically reasonable to initialise weights from Gaussian distribution with zero mean and variance of the form:

$$\sigma = \sqrt{\frac{2}{n_{in} + n_{out}}}. \tag{2.21}$$

or from uniform distribution on the interval $[-r, r]$

$$r = \sqrt{\frac{6}{n_{in} + n_{out}}}, \tag{2.22}$$

For He [11] initialization the variance should be multiplied by the factor of 2, and that is it! Xavier is mostly used for logistic function, while He initialisation is used for ReLU and its variants.

Once weights are initialized, we can proceed to the next step - forward propagation and then to all subsequent steps.

Overall, the entire training algorithm can be described as follows:

- Define the neural network structure;
- Initialize the model's parameters;
- Loop over epochs:
  - Implement forward propagation;
  - Compute loss;
  - Implement backward propagation to calculate the gradients;
  - Update parameters (via specific optimization algorithm);

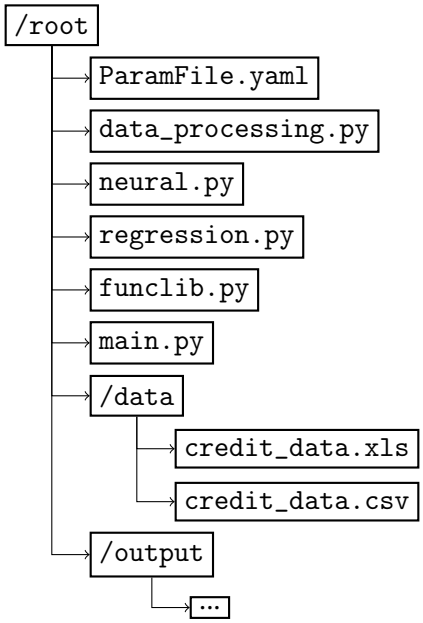
In the next section, I will talk about code implementation and the results I have obtained.

3 Algorithms implementation

In this section, I describe the pipelines written to, first, process the data sets and, second, to apply techniques described in previous section.

3.1 File Organisation

Overall, the program's structured as follows:



- `root` - root folder, contains all the files;
- `data` - folder which contains all data used in the current project;
- `output` - the folder which contains all output (e.g. plots) of the resulting fitting etc.;
- `ParamFile.yaml` - parameter file used to configure neural network;
- `data_processing.py` - the module used to process the Credit Card data and Franke function;
- `neural.py` - module to instantiate and apply all techniques necessary for Neural Network to run successfully;
- `regression.py` - module to run linear and logistic regression analysis;
- `funclib.py` - module which contains all functions used in the program;

- `main.py` (`main.ipynb`) - the entry point of the program (`jupyter` notebook for nicer output).

Before running the program, choose your preferred configuration inside the parameter file. You can decide on type of the task (Classification or Regression), the input data and output folder, the number of hidden layers, type of activation function for hidden layers and the output layer, number of neurons in hidden layer and output layer (for classification it should be 2, for regression 1), number of epochs to loop over, regularisation parameter and learning rate and so on.

Run the program either with main.py or main.ipynb. The latter will give you nicer view on results as it supports the markup.

3.2 Data Preparation

It is essential to all neural networks to clean and pre-process your data. I will start with Default Credit Card data. Let's first look if it contains missing or NaN values:

```
data.info()
```

The output is given tells us that everything is in place.

I am also renaming 'PAY_0' to 'PAY_1', make all the heading lower case and drop the 'ID' column:

```
dataFormat = dataPath[-4:]
if dataFormat == '.csv':
    data = pd.read_csv(dataPath, index_col=False)#, header = None,
        ↪ index_col=False)
    data.rename(columns={'default.payment.next.month': 'default'},
        ↪ inplace = True)
elif dataFormat == '.xls':
    nanDict = {}
    data = pd.read_excel(dataPath, header=1, skiprows=0, index_col=False,
        ↪  na_values=nanDict)
    data.rename(index=str, columns={"default␣payment␣next␣month": '
        ↪ default'}, inplace=True)

data = data.drop('ID', axis = 1)
data.rename(columns={'PAY_0': 'PAY_1'}, inplace = True)
data.rename(columns=lambda x: x.lower(), inplace = True)
```

To understand which feature plays the major 'predictor' role, I am plotting the correlation matrix:

```
corr = data.corr()#data.drop('ID', axis = 1).corr()
fig, ax = plt.subplots(figsize=(22, 22))
# Generate a custom diverging colormap
cmap = sbn.diverging_palette(220, 10, as_cmap=True)
# Draw the heatmap with the mask and correct aspect ratio
sbn.heatmap(corr, cmap = cmap, vmin=0,vmax=1, center=0, square=True,
    ↪ annot=True, linewidths=.5)
```

which output can be seen on figure 3.1. The most dense regions are the most important ones; thus, from the plot we can infer that 'pay_i' together 'bill_amti' are the most important features of the data set. However, I would like to look into data in more detail.

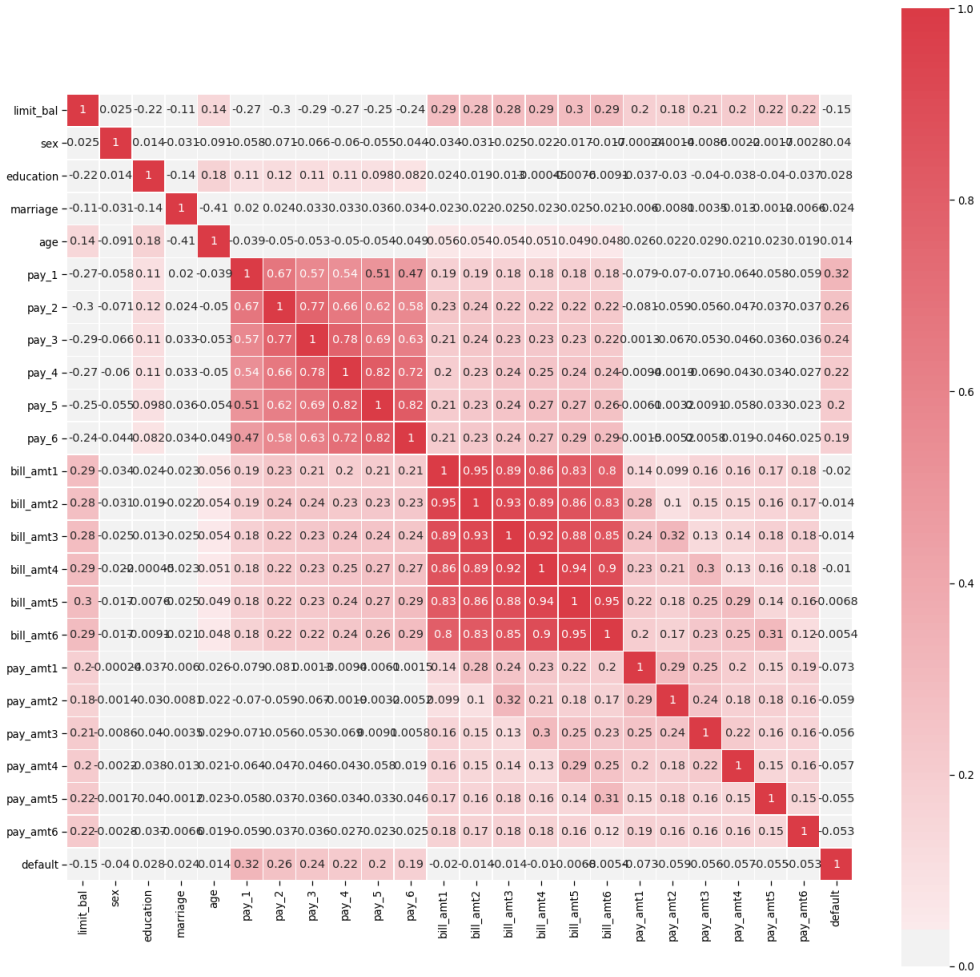


Figure 3.1: The correlation matrix of the full data set, the indices on the x- and y-axis are corresponding to the data types.

I will print the random sample of the data:

```
display(data.sample(10))
```

There we can spot some abnormalities. If we compare the description of the data provided by [3], we can see that some values doesn't make sense. For instance: 'age' column has one more value '0', which is not clear what it suppose to represent. Also, 'education' ranges from $0-6$, which means 0, 5 and 6 are unlabeled data. There are not that many of then, however, that is why I decided to add these unlabeled points to the value, which corresponds to the 'other' for their respective columns. The code is as follows:

```
vals = [0, 4, 5, 6]
for val in vals:
    print("We␣have␣{}␣with␣education={}".format(len(data.loc[ data["
        ↪ education"]==val]), val))

fill = (data['education'] == 5) | (data['education'] == 6) | (data['
    ↪ education'] == 0)
data.loc[fill, 'education'] = 4
# counting education values after
print('After␣cleaning')
display(data['education'].value_counts())
# doing the same for marriage
fill = (data['marriage'] == 0)
data.loc[fill, 'marriage'] = 3
print('After␣cleaning')
```

The strange thing is that 'pay_i' also contains unlabeled data:

```
display(data[['pay_1', 'pay_2', 'pay_3', 'pay_4', 'pay_5', 'pay_6']].
    ↪ describe())
vals = [1, 2, 3, 4, 5, 6]
for val in vals:
    display(data['pay_'+str(val)].value_counts())
```

These are '0', which are approximately half of the size of the entire data set. I could throw them away, but eventually decided to leave it as it is, because such operation can obscure our view. However, I will put all '-2' values inside '-1':

```
for val in vals:
    fill = (data['pay_' + str(val)] == -2)
    data.loc[fill, 'pay_' + str(val)] = -1
    display(data['pay_'+str(val)].value_counts())
```

Similar operation were done with all other variables. Later I decided to see how many clients actually had default (figure 3.2):

```
yes = data.default.sum()
no = len(data)-yes
```

```
# Percentage
yes_perc = round(yes/len(data)*100, 1)
no_perc = round(no/len(data)*100, 1)

fig = plt.figure(figsize=(10, 5))
sbn.set_context('notebook', font_scale=1.2)
sbn.countplot('default',data=data, palette="cool")
plt.annotate('Non-default: {}'.format(no), xy=(-0.3, 15000), xytext
    ↪ =(-0.3, 3000), size=12)
plt.annotate('Default: {}'.format(yes), xy=(0.7, 15000), xytext=(0.7,
    ↪ 3000), size=12)
plt.annotate(str(no_perc)+" %", xy=(-0.3, 15000), xytext=(-0.1, 8000),
    ↪ size=12)
plt.annotate(str(yes_perc)+" %", xy=(0.7, 15000), xytext=(0.9, 8000),
    ↪ size=12)
plt.title("CREDIT CARDS' DEFAULT COUNT", size=14)
```

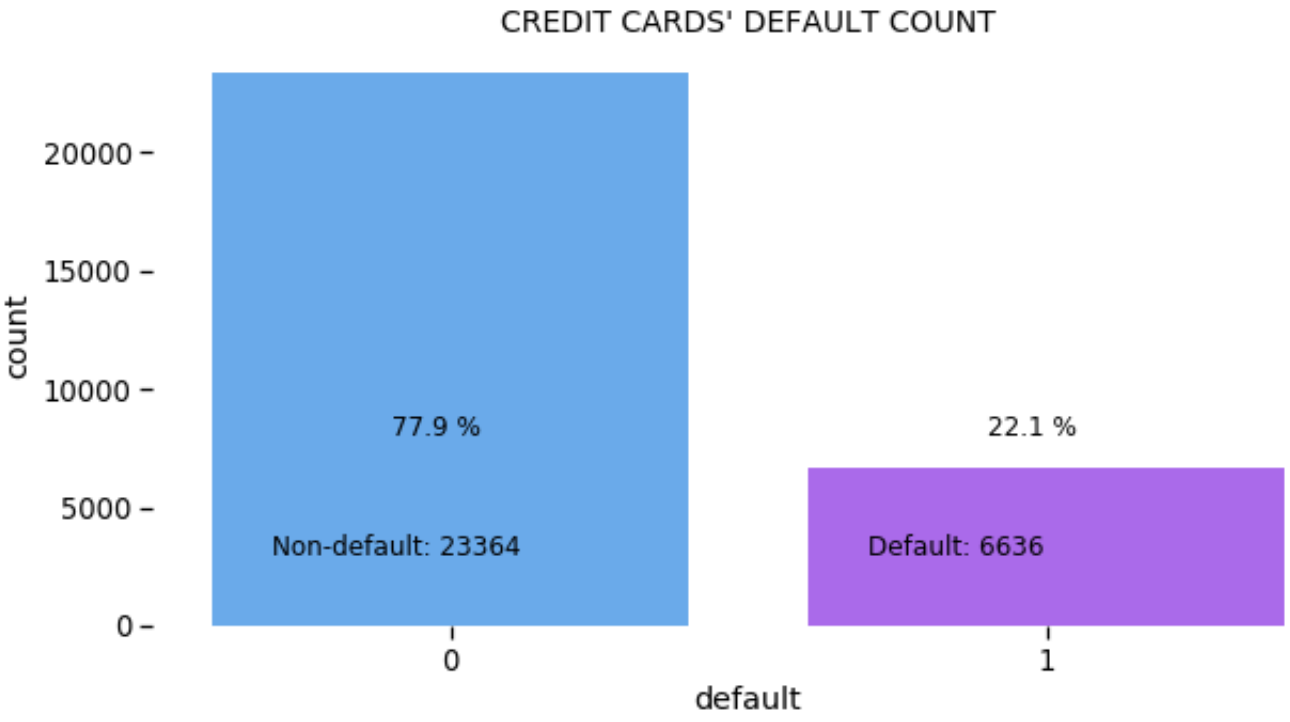


Figure 3.2: The count of the amount of clients who had a default.

And also, I would like to see the frequency of categorical variables (figure 3.3):

```
labels = ['sex', 'education', 'marriage', 'pay_1', 'pay_2',
       'pay_3', 'pay_4', 'pay_5', 'pay_6', 'default']
subset = data[labels]
fig, axes = plt.subplots(3,3, figsize=(20,15), sharey = 'all', facecolor=
    ↪ 'white')
axes = axes.flatten()
fig.suptitle('FREQUENCY OF CATEGORICAL VARIABLES (BY TARGET)')
for axe, catplot in zip(axes, labels):
    order=np.unique(subset.values))
    sbn.countplot(x=catplot, hue="default", data = subset, palette = "
        ↪ cool", ax=axe)
    axe.legend(loc='upper right', bbox_to_anchor=(1.0, 1.00),title='
        ↪ default')
```

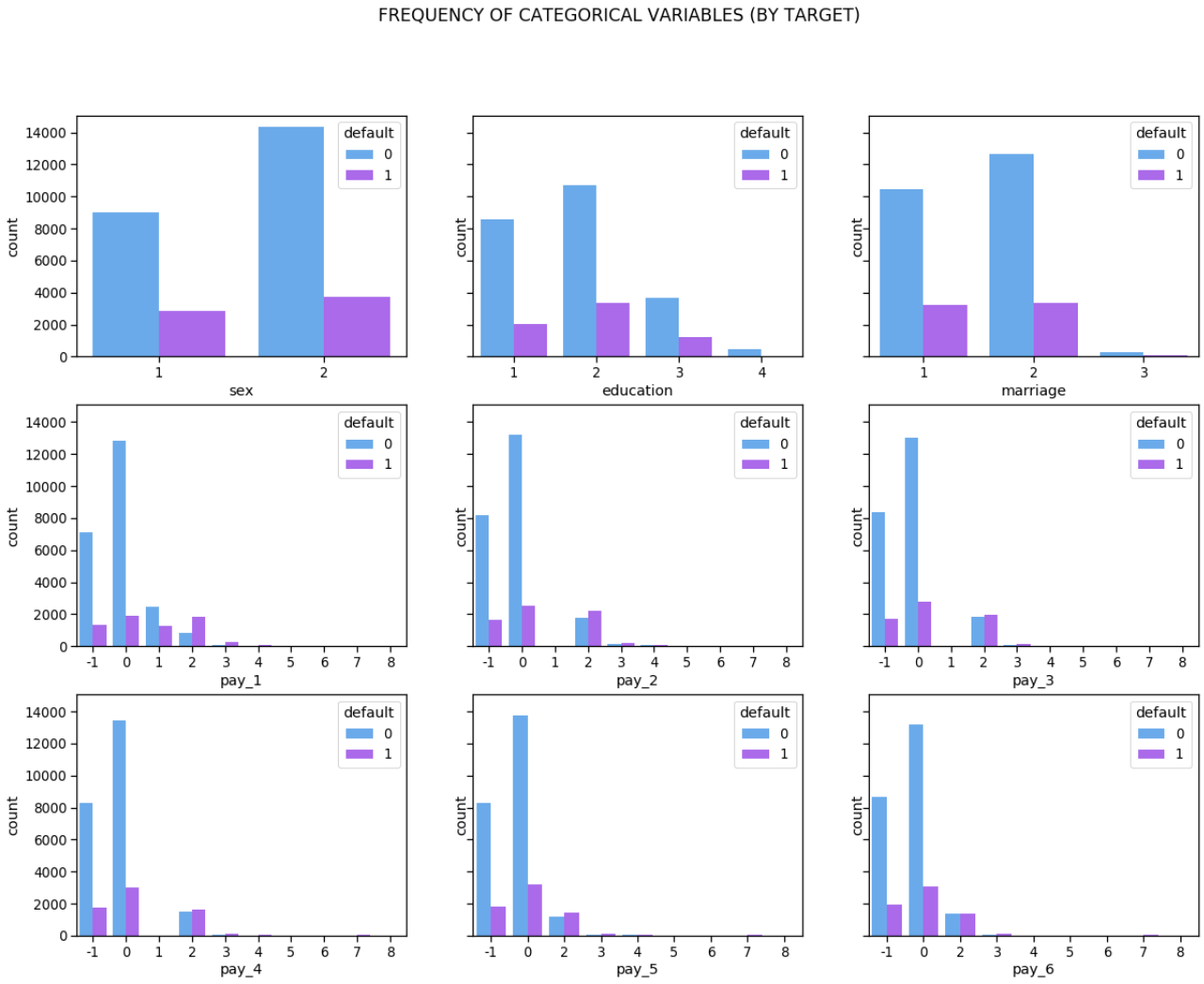


Figure 3.3: Frequency of categorical variables.

Next, I will use pandas to create dummy features:

```
data['grad_school']  = (data['education']==1).astype('int')
data['university']   = (data['education']==2).astype('int')
data['high_school']  = (data['education']==3).astype('int')
data['others']       = (data['education']==4).astype('int')
data.drop('education', axis=1, inplace=True)
```

```
data['male'] = (data['sex']==1).astype('int')
data.drop('sex', axis=1, inplace=True)
# dumping all singles and others in one category
data['married'] = (data['marriage']==2).astype('int')
data.drop('marriage', axis=1, inplace=True)
print(data.head(10))
```

Next, I am creating *One Hot Encoder* and plit my data set into train and test sets:

```
X = data.loc[:, data.columns != "default"].values
Y = data.loc[:, data.columns == "default"].values

onehotencoder = OneHotEncoder(sparse=False, categories="auto")
X = ColumnTransformer([("", onehotencoder, [3])], remainder="passthrough"
    ↪ ).fit_transform(X)

X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size =
    ↪ testSize, random_state = 1)

Y_onehot = onehotencoder.fit_transform(Y)
Y_train_onehot = onehotencoder.fit_transform(Y_train)#.reshape(len(
    ↪ Y_train), -1))
Y_test_onehot =  onehotencoder.fit_transform(Y_test)
```

I also normalise the data:

```
ss = StandardScaler()
rs = RobustScaler()
X_train = ss.fit_transform( X_train )
X_test = ss.transform( X_test )
```

and then pass everything to the next method, which will pre-configure the NN. with this I am all set to start doing regression and train my NN.

3.3 Logistic Regression

Due to the nature of the task, I have split the gradient descent for logistic regression (which is basically a neural network without hidden layer) into separate class. To tune the parameters I have writtn the small method, which is based on multiprocessing:

```
myResults = Parallel(n_jobs=nproc, verbose=10)(delayed(
    ↪ DoGriSearchClassification)\
(logisticData, i, j, alpha, lmbd) for i, alpha in enumerate(alphas) for j
    ↪ , lmbd in enumerate(lambdas))
```

where

```
def DoGriSearchClassification(logisticData, i, j, alpha, lmbd):
    # passing data values
    NNType, NNArch, nLayers, nFeatures, \
        nHidden, nOutput, epochs, \
        X, Y, X_train, X_test, Y_train, Y_test, Y_onehot, Y_train_onehot,
            ↪ Y_test_onehot,\
        m, nInput, seed, onehotencoder,\
        BatchSize, Optimization = logisticData

    pipeline = regression.RegressionPipeline()
    # scores to be evaluated
    precision = []
    recall = []
    accuracy = []
    f1 = []
    roc_auc = []
    costsTrain = []
    # Getting cost function averaged over all epochs
    # Doing KFold Cross validation - getting appropriate indexes
    kf = KFold(n_splits=5, random_state=1, shuffle=True)
    for train_index, test_index in kf.split(X):
        #print("TRAIN:", train_index, "TEST:", test_index)
        X_train, X_test = X[train_index], X[test_index]
        #Y_train, Y_test = Y[train_index], Y[test_index]
        Y_train, Y_test = Y[train_index], Y[test_index]#Y_onehot[
            ↪ train_index-1][:], Y_onehot[test_index-1][:]
        # Training Logistic Regression Model
        costs, modelParams = pipeline.DoLogisticRegression(X_train, \
                                                Y_train, epochs, lmbd, alpha)
        # making prediction (for each splitted set) - fitting
        Y_pred = pipeline.PredictLogisticRegression(X_test, modelParams)
        # getting scores - accuracy, recall, precision, f1, auc
        precision.append(precision_score(Y_test, Y_pred))
        recall.append(recall_score(Y_test, Y_pred))
        accuracy.append(accuracy_score(Y_test, Y_pred, normalize=True))
        f1.append(f1_score(Y_test, Y_pred))
        roc_auc.append(roc_auc_score(Y_test,Y_pred))
        costsTrain.append(costs)

    # averaged scores for give set of alphas and lambdas
    APrecision = sum(precision)/len(precision)
    ARecall = sum(recall)/len(recall)
    AAccuracy = sum(accuracy)/len(accuracy)
    AF1 = sum(f1)/len(f1)
```

```
    AROC = sum(roc_auc)/len(roc_auc)

    results = {AROC: ['lambda='+str(lmbd)+', alpha='+str(alpha)]}
    myResult = pd.DataFrame(results)

    return myResult
```

method does KFold cross validation to tune parameters to maximize `roc_auc_score`. The GD algorithm for logistic regression is:

```
def DoLogisticRegression(self, *args):
    X_train = args[0]
    Y_train_onehot = args[1]
    epochs   = args[2]
    lmbd = args[3]
    alpha = args[4]
    activeFuncs = funclib.ActivationFuncs()
    costFuncs = funclib.CostFuncs()
    theta = np.zeros((X_train.shape[1], 1))
    epochs1 = range(epochs)
    m = len(Y_train_onehot)
    costs = []
    for epoch in epochs1:
        Y_pred = np.dot(X_train, theta)
        A = activeFuncs.CallSigmoid(Y_pred)#CallSigmoid(Y_pred)
        # cost function
        J, dJ = costFuncs.CallLogistic(X_train, Y_train_onehot, A)
        # Adding regularisation
        J = J + lmbd / (2*m) * np.sum(theta**2)
        if np.isnan(J):
            print("J is ", J)
        dJ = dJ + lmbd * theta / m
        # updating weights
        theta = theta - alpha * dJ #+ lmbd * theta/m
        # updating cost func history
        costs.append(J)

    return costs, theta
```

In this way, I found to different values for hyper parameters. the plot of cost function after training with these parameters can be found in figure 3.4.

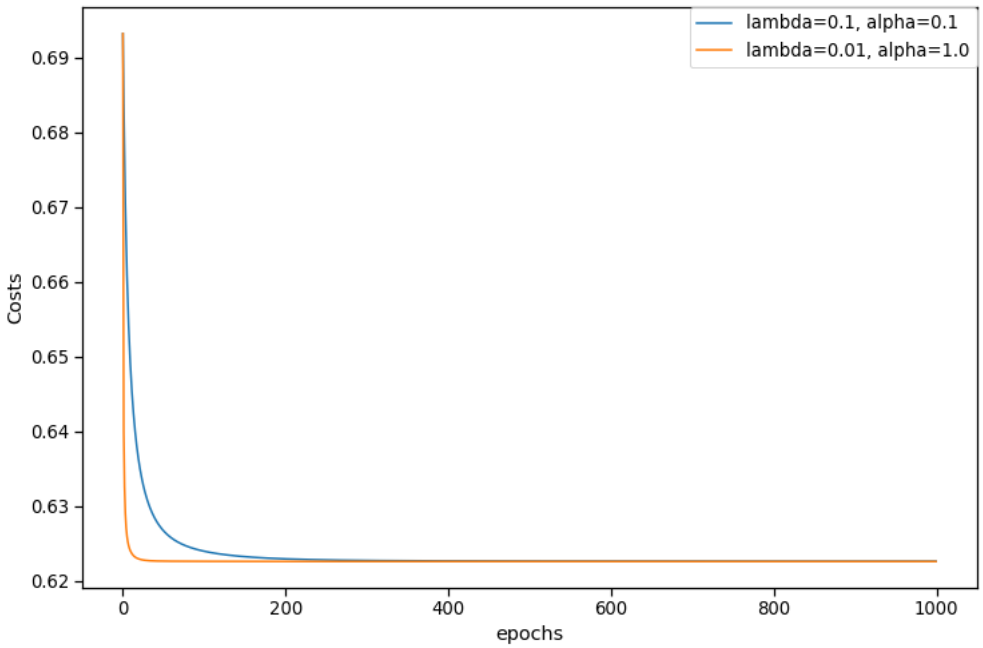


Figure 3.4: The plot of cost function for different hyper parameters, after tuning the algorithm with manual approach.

The corresponding ROC AUC curve can be found on figure [**?**], while the classification report with predicted values from manual algorithm gives the following result for sets of parameters after testing:

```
              precision    recall  f1-score   support

           0       0.87      0.81      0.84     11637
           1       0.47      0.59      0.52      3363

    accuracy                           0.76     15000
   macro avg       0.67      0.70      0.68     15000
weighted avg       0.78      0.76      0.77     15000

              precision    recall  f1-score   support

           0       0.87      0.81      0.84     11637
```

1	0.47	0.59	0.52	3363
accuracy			0.76	15000
macro avg	0.67	0.70	0.68	15000
weighted avg	0.78	0.76	0.77	15000

As we can see from the table, the accuracy is 76% which is not bad. The corresponding Scikit Learn result is:

	precision	recall	f1-score	support
0	0.82	0.95	0.88	11637
1	0.65	0.29	0.40	3363
accuracy			0.81	15000
macro avg	0.74	0.62	0.64	15000
weighted avg	0.78	0.81	0.78	15000

Which results in slightly better accuracy of 81%.

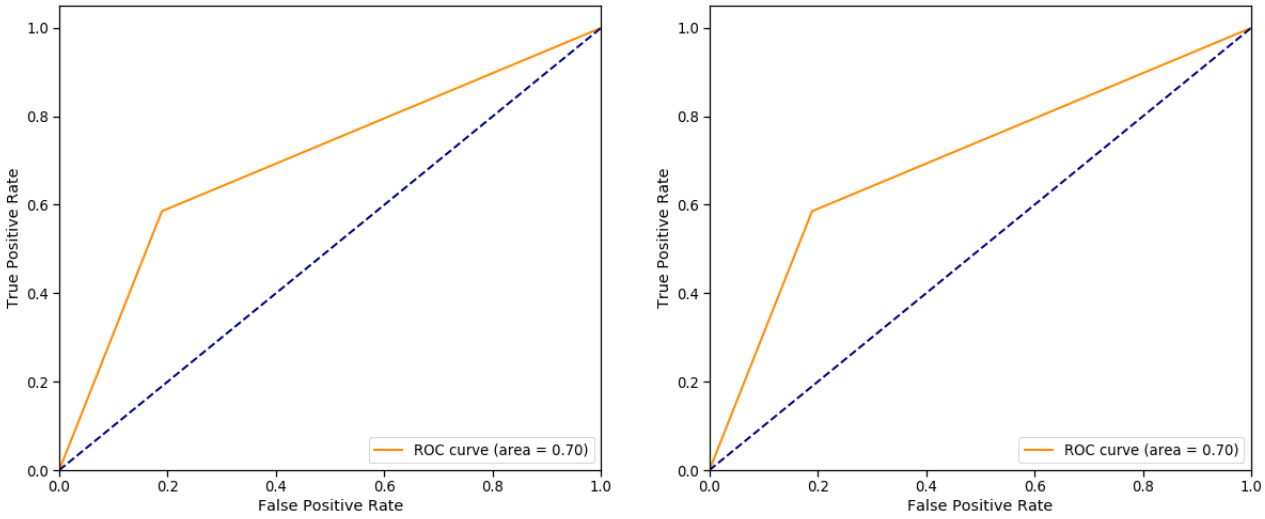


Figure 3.5: ROC AUC curve for the manual implementation for gradient descent algorithm. Left plot: $\alpha = 0.1$, $\lambda = 0.1$. Right plot: $= 0.01$, $\alpha = 1.0$.

3.4 Neural Network - logistic Regression

I am using 2-layer NN (1 hidden layer), with 40 hidden neurons, $\alpha = 0.01$, $\lambda = 0.0001$, the optimization algorithm is Adagrad with batch size of 4056. Activation function are sigmoid for hidden layer and softmax for output layer. The code For NN listed in the appendix, as it is quite large.

With such initialisation, the results for the manual algorithm are:

	precision	recall	f1-score	support
0	0.83	0.95	0.89	11637
1	0.67	0.33	0.44	3363
accuracy			0.81	15000
macro avg	0.75	0.64	0.67	15000
weighted avg	0.79	0.81	0.79	15000

The cost history is plotted on figure 3.6, while the ROC AUC Curve is given on figure [**?**]. As we can see the parameters definitely requires some tuning, but the accuracy result of 81% is quite good already.

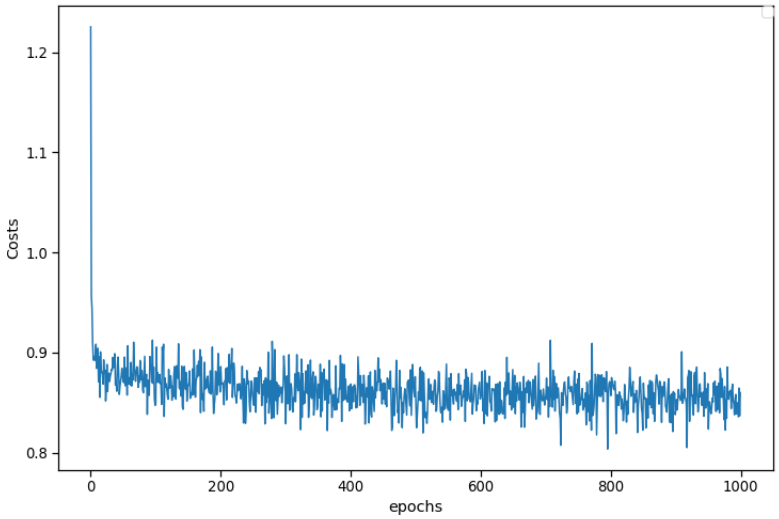


Figure 3.6: The plot of cost function for different hyper parameters, after tuning the algorithm with manual approach.

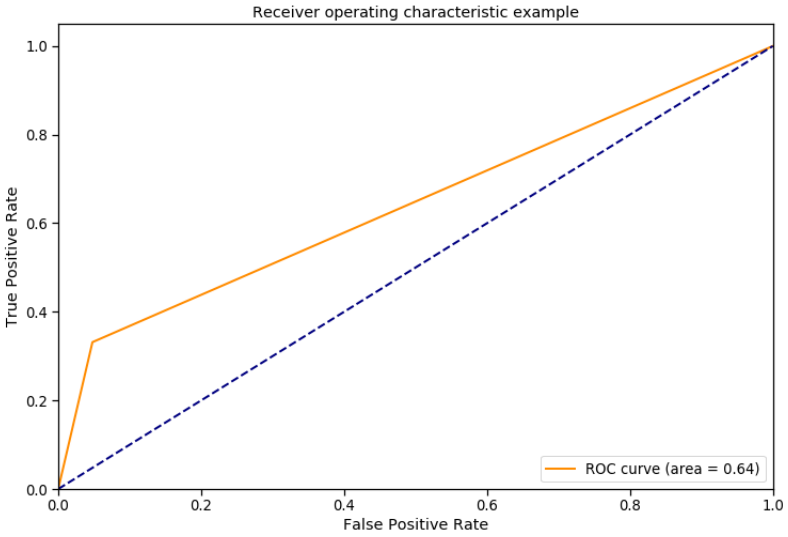


Figure 3.7: ROC AUC curve for the manual implementation for gradient descent algorithm. Left plot: $\alpha = 0.1$, $\lambda = 0.1$. Right plot: $= 0.01$, $\alpha = 1.0$.

The corresponding Scikit Learn analog gives:

```
              precision    recall  f1-score   support

           0       0.83      0.93      0.88     11637
           1       0.60      0.35      0.44      3363

    accuracy                           0.80     15000
   macro avg       0.72      0.64      0.66     15000
weighted avg       0.78      0.80      0.78     15000

Roc auc:  0.6415844843230899
```

As we can see, although I haven't specifically tuned my algorithm, it gives the comparable results to Scikit.

3.5 Neural Network - Linear Regression

I have encountered a lot of issues with Linear Regression. Specifically, the preferred functions for LR are ReLU for hidden layers and Linear for output layer. However, with such implementation, some values are exploding or becoming infinitely small (exploding and vanishing gra-

dient problems). I have tried to change the weights initialisation with Xavier and He, but it didn't help me.

After several days of debugging, I have changed the MBGD method for Adagrad and the program seem started to work properly (at least it doesn't crash anymore). Therefore, I will be presenting results I am having obtained with Adagrad.

The Franke Function is defined on 40×40 grid. This values should give enough data points.

I am using 2-layer NN (1 hidden layer), with 21 hidden neurons, $\alpha = 0.1$, $\lambda = 0.0001$, the optimization algorithm is Adagrad with batch size of 4. Activation function are ReLU for hidden layer and Linear for output layer. The code For NN listed in the appendix, as it is quite large.

The corresponding cost function and comparison of fitted surfaces can be found in figures [**?**] and [**?**] respectively.

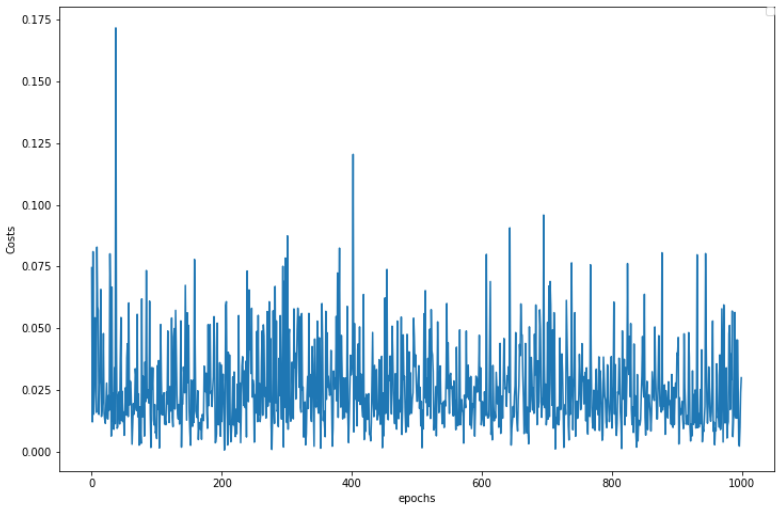


Figure 3.8: The plot of cost function for different hyper parameters, after tuning the algorithm with manual approach.

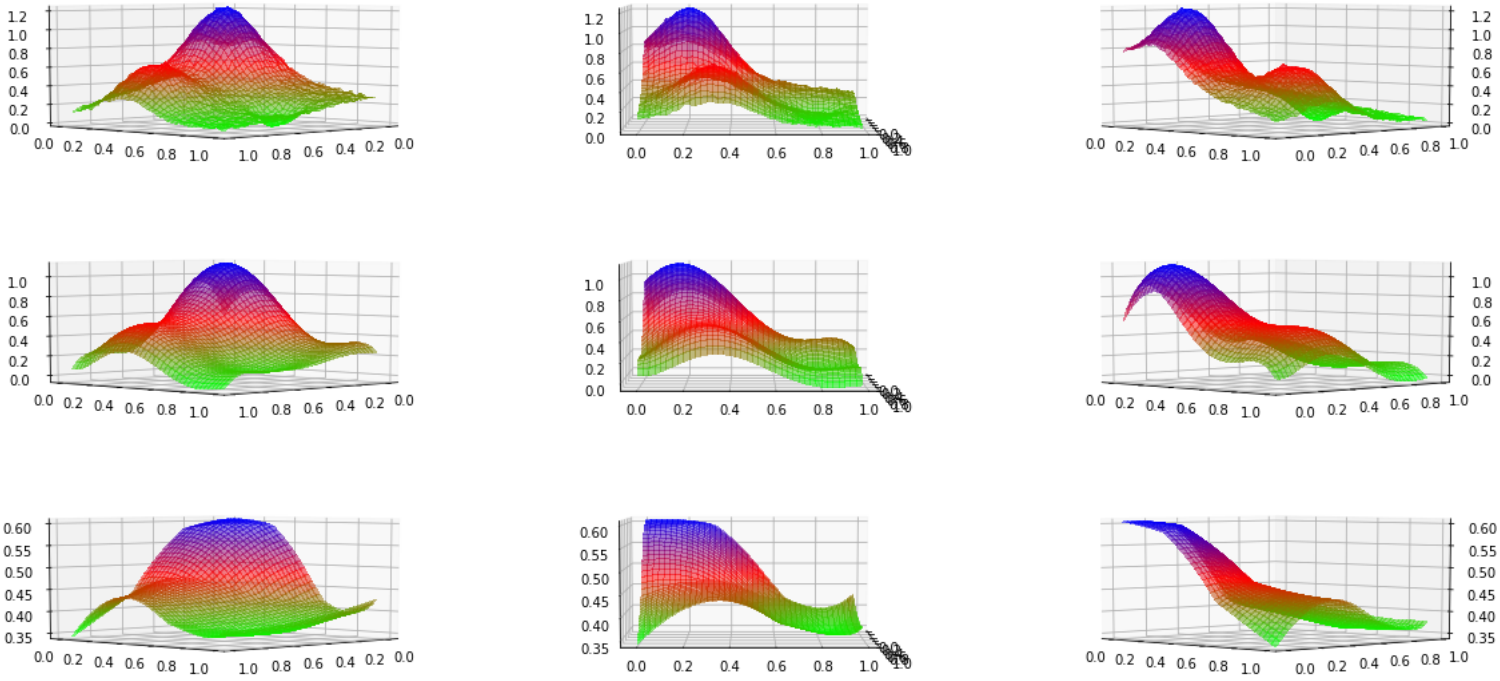


Figure 3.9: The plot of surfaces for Franke function. The top row is the original noisy function, the middle row the fitted function via linear regression algorithm introduced in project 1 and bottom row is the function obtained via Neural Network run.

Although it seems like algorithm doesn't learn, as cost function doesn't change, the shape of the function is preserved.

Overall, the Neural Network Algorithms seems to work fine, but some parameters require tuning.

References

[1] UCI University of California Irvine. Default of credit card clients Data Set. 2016. url: https: //archive.ics.uci.edu/ml/datasets/ default + of + credit + card + clients (visited on 11/12/2019).

[2] Brilenkov, M., Report for FYS-STK4155: Project 1; October 2019.

[3] Yeh, I. C., Lien, C. H. (2009). The comparisons of data mining techniques for the predictive accuracy of probability of default of credit card clients. Expert Systems with Applications, 36(2), 2473-2480.

[4] Franke, R., A critical comparison of some methods for interpolation of scattered data. Tech. rep. Monterey, California: Naval Post- graduate School., 1979.

[5] Caudill, M., "Neural Network Primer: Part I" , AI Expert, Feb. 1989

[6] Hjorth-Jensen, M.. Lectures Notes in FYS-STK4155. Data Analysis and Machine Learning: Neural networks, from the simple perceptron to deep learning. Oct. 2019.

[7] Qian, N. (1999). On the momentum term in gradient descent learning algorithms. Neural Networks : The Official Journal of the International Neural Network Society, 12(1), 145–151. http://doi.org/10.1016/S0893-6080(98)00116-6

[8] Nesterov, Y. (1983). A method for unconstrained convex minimization problem with the rate of convergence o(1/k2). Doklady ANSSSR (translated as Soviet.Math.Docl.), vol. 269, pp. 543– 547.

[9] Duchi, J., Hazan, E., Singer, Y. (2011). Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. Journal of Machine Learning Research, 12, 2121–2159. Retrieved from http://jmlr.org/papers/v12/duchi11a.html

[10] Xavier, G. and Yoshua, B.. “Understanding the difficulty of training deep feed-forward neural networks”. In: Proceedings of the thirteenth international conference on artificial intelligence and statistics. 2010, pp. 249-256.

[11] Kaiming He et al. “Delving deep into rectifiers: Surpassing human-level performance on imagenet classification”. In: Proceedings of the IEEE international conference on computer vision. 2015, pp. 1026–1034.

Code

Listing 1: "Parameter File"

```
#============================================================================#
# Parameter file for configuring NeuralNetwork
#============================================================================#
# Type of task to address
type:           'Classification' #['Classification', 'Regression']
# Path to data
dataPath:       'data/credit_data.xls'
# Output Path - where to save all the files (phg's etc.)
outputPath:     'output/'
# Specify the seed
RandomSeed:     1
# Choose number of layers (can be any number)
nHiddenLayers:  1 # type 0 to run Logistic Regression
# Activation functions for hidden and output layers
# (['sigmoid', 'tanh', 'relu', 'softmax'])
hiddenFunc:     'sigmoid' #'sigmoid'
outputFunc:     'softmax' #'identity' #'softmax'
# Size of the test sample
testSize:       0.5
# Number of Neurons for hidden and output layers
nHiddenNeurons: 40 # 21 # 4  # 30
nOutputNeurons: 2 # classification = 2, Regression = 1
# Epochs to train the algorithm
epochs:         1000 #200 #170
# Optimization Algorithm: choose it wisely :)
#['MBGD', 'Adagrad' 'GD'] <= for minibatch, if you choose 1 you will get just
    ↪ stochstic GD
# if you choose simply GD, then it willl ignore batchSize parameter and will use
    ↪ the whole data set
Optimization:   'Adagrad' # please use Adagrad for linear regression (as it may
    ↪ crush overwise
# Batch size for Gradient Descent => if 0, will use simple gradient descent
BatchSize:      4056 #16 # 1 #10000 <= increase batch size and it will be good
# Learning rate
alpha:          0.01 # 0.01 # 0.01 #0.0001 #np.logspace(-5, 1, 7)
# Regularisation
lambda:         0.0001 #0.0000001 # np.logspace(-5, 1, 7) # if 0 then no
    ↪ regularisation used
# The range of alphas to apply grid search
alphas:         [-5, 1, 7] # np.logspace(-5, 1, 7)
lambdas:        [-5, 1, 7]
#============================================================================#
# I am going to use Franke function, that is why this part of the parameters
# configuring the franke data. In theory it it possible to use variable dataPath
# and work with the real data set but again some cleaning and data processing
# would be required in that step. I think this can be ommitted for Franke data,
# because we know more or less its shape and also because it is alredy normalized
```

```
# (in a sense) :)
#
# Please, bear in mind that, you still need to specify the type of analysis
# and also all other necessary parameters.
function:       'Franke' # ['Franke', 'Paraboloid', 'Beale']
# number of independent variables
nVars:          2
# polynomial degree to approximate
degree:         5
# number of grid points (e.g. if 100, we get grid 100x100)
nPoints:        40 # 100 #<= overfitting?
# noise level, aka sigma/psilon/whatever
noise:          0.01
# number of processors to use - useless parameter
nProc:          10
# Decide whether we want to get best parameters for our model



# Use very small lamdba for Linear regression, like 0.000001
```

Listing 2: "Entry point to the program"

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Wed Nov 13 11:21:05 2019

@author: maksymb
"""

# importing libraries
import os
import sys
import math as mt
import numpy as np
# for polynomial manipulation
import sympy as sp
# from sympy import *
import itertools as it
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import seaborn as sbn
import pandas as pd
import multiprocessing as mp
from joblib import Parallel, delayed

from IPython.display import display, Latex, Markdown

from collections import Counter

# to read parameter file
import yaml


# Scikitlearn imports to check results
from sklearn.linear_model import LogisticRegression, LinearRegression
from sklearn.model_selection import GridSearchCV, train_test_split, KFold
from sklearn.metrics import confusion_matrix, roc_curve, auc, roc_auc_score
from sklearn.metrics import recall_score, precision_score, accuracy_score,
    ↪ f1_score, classification_report
from sklearn.preprocessing import LabelEncoder, OneHotEncoder, StandardScaler
from numpy import argmax

# We'll need some metrics to evaluate our models
from sklearn.neural_network import MLPClassifier, MLPRegressor


import keras
# stochastic gradient descent
from keras.optimizers import SGD
from keras.models import Sequential
from keras.layers import Dense
```

```
from keras.callbacks import History


# importing manually created libraries
import funclib
import neural
import regression
import data_processing

import time


def DoGriSearchRegression(logisticData, i, j, alpha, lmbd):

    '''
    1. Big Plot of Franke Function and Corresponding Gradient analysis
    2. Plots of MSE for different parameters (lambda, alpha, bumber of neurons?)
    3. Plots of R2 for different values of the same parameters
    4. Plots of cost function?
    '''

    return print(i, j, k, l)

def DoGriSearchClassification(logisticData, i, j, alpha, lmbd):
    # passing data values
    NNType, NNArch, nLayers, nFeatures, \
        nHidden, nOutput, epochs, \
        X, Y, X_train, X_test, Y_train, Y_test, Y_onehot, Y_train_onehot,
            ↪ Y_test_onehot,\
        m, nInput, seed, onehotencoder,\
        BatchSize, Optimization = logisticData


    pipeline = regression.RegressionPipeline()
    # scores to be evaluated
    precision = []
    recall = []
    accuracy = []
    f1 = []
    roc_auc = []
    costsTrain = []
    # Getting cost function averaged over all epochs
    # Doing KFold Cross validation - getting appropriate indexes
    kf = KFold(n_splits=5, random_state=1, shuffle=True)
    for train_index, test_index in kf.split(X):
        #print("TRAIN:", train_index, "TEST:", test_index)
        X_train, X_test = X[train_index], X[test_index]
        #Y_train, Y_test = Y[train_index], Y[test_index]
        Y_train, Y_test = Y[train_index], Y[test_index]#Y_onehot[train_index
            ↪ -1][:], Y_onehot[test_index-1][:]
        # Training Logistic Regression Model
```

```
        costs, modelParams = pipeline.DoLogisticRegression(X_train, \
                                            Y_train, epochs, lmbd, alpha)
        # making prediction (for each splitted set) - fitting
        Y_pred = pipeline.PredictLogisticRegression(X_test, modelParams)
        # getting scores - accuracy, recall, precision, f1, auc
        precision.append(precision_score(Y_test, Y_pred))
        recall.append(recall_score(Y_test, Y_pred))
        accuracy.append(accuracy_score(Y_test, Y_pred, normalize=True))
        f1.append(f1_score(Y_test, Y_pred))
        roc_auc.append(roc_auc_score(Y_test,Y_pred))
        costsTrain.append(costs)

    # averaged scores for give set of alphas and lambdas
    APrecision = sum(precision)/len(precision)
    ARecall = sum(recall)/len(recall)
    AAccuracy = sum(accuracy)/len(accuracy)
    AF1 = sum(f1)/len(f1)

    AROC = sum(roc_auc)/len(roc_auc)

    # for given set of lambda and alpha parameters (regularization and learning
        ↪ rate)
    # I return the set of precision, recall, accuracy and f1 scores
    #myResult = {'lambda='+str(lmbd): [APrecision, ARecall, AAccuracy, AF1],
    #            'alpha='+str(alpha):[APrecision, ARecall, AAccuracy, AF1]}
    '''
    results = {'lambda='+str(lmbd): [APrecision, ARecall, AAccuracy, AF1],
               'alpha='+str(alpha):[APrecision, ARecall, AAccuracy, AF1]}
    #alphas  = {alpha:[APrecision, ARecall, AAccuracy, AF1]}
    myResult = pd.DataFrame(results)
    '''
    # decided to go only with accuracy to find the highest value
    #results = {AAccuracy: ['lambda='+str(lmbd)+', alpha='+str(alpha)]}#{'lambda
        ↪ ='+str(lmbd)+', alpha='+str(alpha): [AAccuracy]}#, 'alpha='+str(alpha):
        ↪  [AAccuracy]}
    results = {AROC: ['lambda='+str(lmbd)+',␣alpha='+str(alpha)]}
    myResult = pd.DataFrame(results)


    return myResult#print(i, j, alpha, lambd)

# Feed Forward Neural Network
def CallKerasModel(NNArch, nLayers):
    classifier = Sequential()
    print("nHidden", nHidden)

    for layer in range(1, nLayers):
        if layer == 0:
            classifier.add(Dense(nHidden, activation=NNArch[layer]['AF'], \
                                 kernel_initializer='random_normal', input_dim=
                                     ↪ nFeatures))
```

```
        elif layer == nLayers-1:
            classifier.add(Dense(nOutput, activation=NNArch[layer]['AF'], \
                                 kernel_initializer='random_normal'))
        else:
            classifier.add(Dense(nHidden, activation=NNArch[layer]['AF'], \
                                 kernel_initializer='random_normal'))
    return classifier

'''
Main Body of the Program
'''

if __name__ == '__main__':

    # Estimate how much time it took for program to work
    startTime = time.time()

    # Getting parameter file
    paramFile = 'ParamFile.yaml'
    # getting NN configuration - a lot of parameters to trace T_T
    dataProc = data_processing.NetworkArchitecture(paramFile)

    with open(paramFile) as f:
        paramData = yaml.load(f, Loader = yaml.FullLoader)

    NNType = paramData['type']
    outputPath = paramData['outputPath']
    if not os.path.exists(outputPath):
        os.makedirs(outputPath)

    '''
    So, If the total number of layers = 2, we can switch to Simple Logistic
        ↪ regression.
    However, once the number of layers is > 2, we are going to use Neural Network.
    The number of hidden layers can be specified in the parameter file above
    (this is done for simplicity). In principle, the Neural network without hidden
        ↪  layers,
    should produc ethe same results as logistic regression (at least, i think so).
    '''
    #acts_hidden = ["logistic", "tanh", "relu"]
    #act_o = "identity"
    # initialising range of lambdas to be searched for
    alphas = paramData['alphas'] #np.logspace(-5, 1, 7)
    lambdas = paramData['lambdas']
    alphas = np.logspace(alphas[0], alphas[1], alphas[2])
    lambdas = np.logspace(lambdas[0], lambdas[1], lambdas[2])

    #print(alphas)
    #print(lambdas)
    #[print(i, alpha) for i, alpha in enumerate(alphas)]
```

```
    #sys.exit()


    # Getting the number of processors
    nproc = int(paramData['nProc'])
    # Checking which Problem we are facing
    if (NNType == 'Classification'):
        NNType, NNArch, nLayers, nFeatures, \
        nHidden, nOutput, epochs, alpha, lmbd, \
        X, Y, X_train, X_test, Y_train, Y_test, Y_onehot, Y_train_onehot,
            ↪ Y_test_onehot,\
        nInput, seed, onehotencoder,\
        BatchSize, optimization = dataProc.CreateNetwork()

        if optimization == 'GD': #if BatchSize == 0:
            # If batch size is zero, we will use standard Gradient Descent Method
            m = nInput
            #print("m is",m)
        elif optimization == 'MBGD': #elif BatchSize > 0:
            m = BatchSize
        elif optimization == 'Adagrad':
            m  = BatchSize
        else:
            print("Incorrect BatchSize. Check Parameter File!")
            sys.exit()

        # passing data to the function
        logisticData = NNType, NNArch, nLayers, nFeatures, \
        nHidden, nOutput, epochs, \
        X, Y, X_train, X_test, Y_train, Y_test, Y_onehot, Y_train_onehot,
            ↪ Y_test_onehot,\
        m, nInput, seed, onehotencoder,\
        BatchSize, optimization
        '''
        # If we want to explore parameter space, we should get data from
        # the parameter file. So, we need to do a Grid Search for the best
        parameters. As it is expensive operation
        '''
        '''
        Checking the number of layers. If 2 then it is simple classification
        problem. If > 2, we are facing Neural Network. If < 2, you should go
        and check parameter file
        '''
        if (nLayers == 2):
            print('''
                  ====================================
                  Logistic Regression Via Manual Coding
                  ====================================
                  Activation Function is:     {}
                  No.of test data points:     {}
                  No of epochs to learn:      {}
```

```
        Learning Rate, \u03B1:     {}
        Regularization param, \u03BB: {}
                    '''.format('function',
                                X_test.shape[0],
                                epochs,
                                alpha,
                                lmbd)+
        '''
        ====================================
        ''')


# enabling parallel processing - returning the list of pandas data
    ↪ frames
myResults = Parallel(n_jobs=nproc, verbose=10)(delayed(
    ↪ DoGriSearchClassification)\
 (logisticData, i, j, alpha, lmbd) for i, alpha in enumerate(alphas)
     ↪ for j, lmbd in enumerate(lambdas))

df = pd.concat(myResults)
# fill all NaN values with 0
df.fillna(0)
filename = outputPath + '/logistic_regression_table_best_accuracy.csv'
df.to_csv(filename)
# getting the maximum for each column and saving it inside another
    ↪ data frame
df1 = pd.DataFrame(df.max(skipna = True))
#print(df1.max(axis=0,skipna=True))
#filename = '1.csv'
#df1.to_csv(filename)
'''
The highest value of accuracy given in the last column (as its name)
and the cell values are values for lambda and alpha. So, for given
number of epochs, I am getting the following lambda and alphas
'''
#print(df.iloc[:,-1])
params = df.iloc[:,-1].values
params = [x for x in params if str(x) != 'nan']
#print(params)
costsList = []
paramList = []
epochs1 = range(epochs)
fig = plt.figure(figsize=(20, 8))
axe = []
# okay, I've got a set of parameters  for which I can now plot stuff
for i in range(len(params)):
    # getting the best parameters - converting string to floats
    par = params[i].split(",") # <= depending on the parameters we
        ↪ choose from this, it will converge slower or faster
    lmbd = par[0].split("=")
    lmbd = float(lmbd[1])
```

```
        alpha = par[1].split("=")
        alpha = float(alpha[1])
        # trainging model on best parameters
        pipeline = regression.RegressionPipeline()
        costs, theta = pipeline.DoLogisticRegression(X_train, \
                                                     Y_train, epochs, lmbd,
                                                         ↪ alpha)
        # fitting parameters to the data and evaluate all tests etc.
        Y_pred = pipeline.PredictLogisticRegression(X_test, theta)
        # some values gives very slow learning rate, so witha
        # given time frame (epochs) the algorithm wouldn't achieve
        # good results
        if any(cost > 1 for cost in costs):
            continue
        else:
            # getting all the parameters again
            print(classification_report(Y_test, Y_pred))
            costsList.append(costs)
            paramList.append(params[i])
            # plotting ROC AUC
            axe.append(fig.add_subplot(1,2,i+1))
            # Calculating Roc_auc
            fpr, tpr, thresholds = roc_curve(y_true = Y_test, y_score =
                ↪ Y_pred, pos_label = 1) #positive class is 1; negative
                ↪ class is 0
            roc_auc = auc(fpr, tpr)
            lw = 2
            axe[i].plot(fpr, tpr, color='darkorange', lw=lw, label='ROC␣
                ↪ curve␣(area␣=␣%0.2f)' % roc_auc)
            axe[i].plot([0, 1], [0, 1], color='navy', lw=lw, linestyle='--
                ↪ ')
            axe[i].set_xlim([0.0, 1.0])
            axe[i].set_ylim([0.0, 1.05])
            axe[i].set_xlabel('False␣Positive␣Rate')
            axe[i].set_ylabel('True␣Positive␣Rate')
            axe[i].legend(loc="lower␣right")

#plt.title('Receiver operating characteristic example')
#plt.legend(loc="lower right")
# saving figure
filename = outputPath + '/'+ 'logreg_rocauc_e' + str(epochs).zfill(4)+
    ↪ '.png'
fig.savefig(filename)

#print(costsList)
fig,ax = plt.subplots(figsize=(12, 8))
#print(paramList)
# plotting costs
#epochs1 = range(epochs)
for i in range(len(costsList)):
    plt.plot(epochs1, costsList[i], label = paramList[i])
```

```
plt.legend(loc='upper right', bbox_to_anchor=(1.0, 1.00),
    ↪ borderaxespad=0.)
plt.ylabel("Costs")
plt.xlabel("epochs")
plt.show()
# saving cost functions
filename = outputPath + '/'+ 'logreg_costs_e' + str(epochs).zfill(4)+'
    ↪ .png'
fig.savefig(filename)

# Calculating Roc metrics
#fpr, tpr, thresholds = roc_curve(y_true = Y_test, y_score = Y_pred,
    ↪ pos_label = 1) #positive class is 1; negative class is 0
#roc_auc = auc(fpr, tpr)

#fig = plt.figure()
#lw = 2
#plt.plot(fpr, tpr, color='darkorange',
#         lw=lw, label='ROC curve (area = %0.2f)' % roc_auc)

#plt.plot([0, 1], [0, 1], color='navy', lw=lw, linestyle='--')
#plt.xlim([0.0, 1.0])
#plt.ylim([0.0, 1.05])
#plt.xlabel('False Positive Rate')
#plt.ylabel('True Positive Rate')
#plt.title('Receiver operating characteristic example')
#plt.legend(loc="lower right")
#plt.show()

#sys.exit()

'''
param_grid = dict(batch_size=batch_size, epochs=epochs)
grid = GridSearchCV(estimator=model, param_grid=param_grid, n_jobs=-1,
    ↪  cv=3)
grid_result = grid.fit(X, Y)
# summarize results
print("Best: %f using %s" % (grid_result.best_score_, grid_result.
    ↪ best_params_))
'''

# Accuracy: checking for 0 occurence
#Accuracy = (TP + TN) / float(len(Y_true)) if Y_true else 0
# Precision:
#Precision = TP / (TP + FP)
# Recall (should be above 0.5 than it is good)
#Recall = TP / (TP + FN)
# F1, could've used 2 * (PRE * REC) / (PRE + REC), but this one doesn'
    ↪ t suffer
# from 0 devision issue
#F1 = (2 * TP) / (2 * TP + FP + FN)
```

```
#PSP = np.sum(np.where(Y_pred==1,1,0))
#PSN = np.sum(np.where(Y_pred==0,1,0))
#SP  = np.sum(np.where(Y_true==1,1,0))
#SN  = np.sum(np.where(Y_true==0,1,0))
#ppv=[tn*1.0/pcn, tp*1.0/pcp]
#trp=[tn*1.0/cn, tp*1.0/cp]
#Accuracy=(TP + TN)*1.0/(SP + SN)

# Evaluating scores
#Accuracy, Precision, Recall, F1 = funclib.ErrorFuncs.CallF1(Y_train,
    ↪ Y_pred)
#print('nknfsnflibsfksbkfbls')
# adding result to the dictionary
#myResult[alpha] = [Accuracy, Precision, Recall, F1]
#myResult[lmbd] = [Accuracy, Precision, Recall, F1]
# enabling parallel processing
#myResult = Parallel(n_jobs=nproc, verbose=10)(delayed(
    ↪ DoGriSearchClassification)\
# (logisticData, i, j, alpha, lambd) for i, alpha in enumerate(alphas)
    ↪  for j, lambd in enumerate(lambdas))
print('''
      ====================================
      Logistic Regression Via Scikit Learn
      ====================================
      Activation Function is:     {}
      No.of test data points:     {}
      No of epochs to learn:      {}
      Learning Rate, \u03B1:      {}
      Regularization param, \u03BB: {}
                  '''.format('function',
                             X_test.shape[0],
                             epochs,
                             alpha,
                             lmbd)+
      '''
      ====================================
      ''')

XTrain = X_train
yTrain = Y_train
XTest = X_test
yTest = Y_test
print("Doing logreg using sklearn")
#%Setting up grid search for optimal parameters of Logistic regression
'''
From Scikit Learn Documentation about scoring:

    accuracy           metrics.accuracy_score
     balanced_accuracy          metrics.balanced_accuracy_score
     average_precision          metrics.average_precision_score
     brier_score_loss           metrics.brier_score_loss
```

```
    f1          metrics.f1_score          for binary targets
    f1_micro    metrics.f1_score          micro-averaged
    f1_macro    metrics.f1_score          macro-averaged
    f1_weighted metrics.f1_score          weighted average
    f1_samples  metrics.f1_score          by multilabel sample
    neg_log_loss metrics.log_loss         requires predict_proba
    ↪ support
    precision   etc.        metrics.precision_score suffixes apply as
    ↪ with    f1
    recall    etc.    metrics.recall_score    suffixes apply as with
    ↪    f1
    jaccard   etc.    metrics.jaccard_score   suffixes apply as with
    ↪    f1
    roc_auc           metrics.roc_auc_score
'''

#lambdas=np.logspace(-5,7,13)
parameters = [{'C': 1./lambdas, "solver":["lbfgs"]}]##len(parameters)
    ↪ }]
scoring = ['accuracy','precision', 'recall', 'f1', 'roc_auc']
logReg = LogisticRegression()
# Finds best hyperparameters, then does regression.
gridSearch = GridSearchCV(logReg, parameters, cv=5, scoring=scoring,
    ↪ refit='roc_auc')

# Fit stuff
gridSearch.fit(XTrain, yTrain.ravel())
yTrue, yPred = yTest, gridSearch.predict(XTest)
print(classification_report(yTrue,yPred))
rep = pd.DataFrame(classification_report(yTrue,yPred,output_dict=True)
    ↪ ).transpose()
display(rep)

logreg_df = pd.DataFrame(gridSearch.cv_results_) # Shows behaviour of
    ↪ CV
display(logreg_df[['param_C','mean_test_accuracy', 'rank_test_accuracy
    ↪ ','mean_test_roc_auc', 'rank_test_roc_auc']])

logreg_df.columns
logreg_df.plot(x='param_C', y='mean_test_accuracy', yerr='
    ↪ std_test_accuracy', logx=True)
logreg_df.plot(x='param_C', y='mean_test_roc_auc', yerr='
    ↪ std_test_roc_auc', logx=True)
plt.show()


#print(myResult)
'''
Plotting Results - Cost Function
'''
#epochs1 = range(epochs)
```

```
        # Plotting results
        #fig,ax = plt.subplots(figsize=(12,8))

        #ax.set_ylabel('Costs')
        #ax.set_xlabel('Iterations')


        #fig,ax = plt.subplots(figsize=(12, 8))
        #print(paramList)
        # plotting costs
        #ax.plot(epochs1, myResult['costs'],'b.')
        #plt.legend(loc='upper right', bbox_to_anchor=(1.0, 1.00),
            ↪ borderaxespad=0.)
        #plt.ylabel("Costs")
        #plt.xlabel("epochs")
        #plt.show()

        '''
        m = np.size(Y_train)
        neuralNet = neural.NeuralNetwork(NNType, NNArch, \
                                         nLayers, nFeatures, \
                                         nHidden, nOutput, \
                                         epochs, alpha, \
                                         lmbd, nInput, seed)
        modelParams, costs = neuralNet.TrainNetwork(X_train, Y_train, m)
        '''

    '''
    Doing Neural Networks if the amount of layeres is more than 2
    '''

    if (nLayers > 2):

        '''
        Switching to Neural Network
        '''



        #m = np.size(Y_train)
        print('Neural Network')
        # passing configuration
        neuralNet = neural.NeuralNetwork(NNType, NNArch, \
                                         nLayers, nFeatures, \
                                         nHidden, nOutput, \
                                         epochs, alpha, \
                                         lmbd, nInput, seed, BatchSize,
                                             ↪ optimization)


        '''
```

```
            1. Add Parallelisation
            2. Add plots
            3. Save all the values for scores etc. to table or whatever
            '''
            #myResult = Parallel(n_jobs=nproc, verbose=10)(delayed(
                ↪ DoGriSearchClassification)\
            # (logisticData, i, j, alpha, lmbd) for i, alpha in enumerate(alphas)
                ↪ for j, lmbd in enumerate(lambdas))



            #print(type(BatchSize))
            if (BatchSize==0):
                print("Gradient␣Descent")

                # evaluating model parameters on test data
                modelParams, costs = neuralNet.TrainNetworkGD(X_train,
                    ↪ Y_train_onehot, m)
                # making a prediction
                Y_pred = neuralNet.MakePrediction(X_test, modelParams)
                print('''
                  ================================
                  Model Evaluation     -   Manual
                  ================================
{}
Accuracy score on test set: {}
                  ================================
                  Model Evaluation     -    Keras
                  ================================
                      '''.format(classification_report(Y_test, Y_pred),
                      accuracy_score(Y_test, Y_pred)))

                fig,ax = plt.subplots(figsize=(12, 8))
                #print(paramList)
                # plotting costs
                epochs1 = range(epochs)
                #for i in range(len(costsList)):
                plt.plot(epochs1, costs)
                plt.legend(loc='upper␣right', bbox_to_anchor=(1.0, 1.00),
                    ↪ borderaxespad=0.)
                plt.ylabel("Costs")
                plt.xlabel("epochs")
                plt.show()
                sys.exit()
                #print("Accuracy score on test set: ", accuracy_score(Y_test,
                    ↪ Y_pred))
                #print()
            elif (BatchSize > 0):
                # just to make sure m is equal to batch size
                #m = BatchSize
                print("Mini␣Batches")
```

```
                # evaluating costs
                modelParams, costs = neuralNet.TrainNetworkMBGD(X_train,
                    ↪ Y_train_onehot, m)#TrainNetworkMBGD(X_train, Y_train_onehot
                    ↪ , m)
                # making aprediction
                Y_pred = neuralNet.MakePrediction(X_test, modelParams)
                print('''
                  ================================
                  Model Evaluation     -   Manual
                  ================================
{}
Accuracy score on test set: {}
                  ================================
                  Model Evaluation     -    Keras
                  ================================
                      '''.format(classification_report(Y_test, Y_pred),
                      accuracy_score(Y_test, Y_pred)))

                #print("Accuracy score on test set: ", accuracy_score(Y_test,
                    ↪ test_predict))

                # Calculating Roc metrics
                fig,ax = plt.subplots(figsize=(12, 8))
                fpr, tpr, thresholds = roc_curve(y_true = Y_test, y_score = Y_pred
                    ↪ , pos_label = 1) #positive class is 1; negative class is 0
                roc_auc = auc(fpr, tpr)

                #fig = plt.figure()
                lw = 2
                plt.plot(fpr, tpr, color='darkorange',
                         lw=lw, label='ROC␣curve␣(area␣=␣%0.2f)' % roc_auc)

                plt.plot([0, 1], [0, 1], color='navy', lw=lw, linestyle='--')
                plt.xlim([0.0, 1.0])
                plt.ylim([0.0, 1.05])
                plt.xlabel('False␣Positive␣Rate')
                plt.ylabel('True␣Positive␣Rate')
                plt.title('Receiver␣operating␣characteristic␣example')
                plt.legend(loc="lower␣right")
                plt.show()

                filename = outputPath + '/'+ 'nnlog_rocauc_e' + str(epochs).zfill
                    ↪ (4)+'.png'
                fig.savefig(filename)

                # the cost function
                fig,ax = plt.subplots(figsize=(12, 8))
                #print(paramList)
                # plotting costs
                epochs1 = range(epochs)
                #for i in range(len(costsList)):
```

```
        plt.plot(epochs1, costs)
        plt.legend(loc='upper right', bbox_to_anchor=(1.0, 1.00),
            ↪ borderaxespad=0.)
        plt.ylabel("Costs")
        plt.xlabel("epochs")
        plt.show()

        filename = outputPath + '/'+ 'nnlog_costs_e' + str(epochs).zfill
            ↪ (4)+'.png'
        fig.savefig(filename)

        #sys.exit()


    #modelParams,
    #print(nHidden)
    # Classify using sklearn
    clf = MLPClassifier(solver="lbfgs", alpha=alpha, hidden_layer_sizes=
        ↪ nHidden)
    clf.fit(X_train, Y_train)
    yTrue, yPred = Y_test, clf.predict(X_test)
    print(classification_report(yTrue, yPred))
    print("Roc auc: ", roc_auc_score(yTrue, yPred))

    print('''
          Initialising Keras
          ''')
    # decoding from keras
    #print(np.shape(Y_train))
    #Y_train = np.argmax(Y_train, axis=1)#.reshape(1,-1)
    #print(np.shape(Y_train))

    classifier = CallKerasModel(NNArch, nLayers)

    '''
    To optimize our neural network we use Adam. Adam stands for Adaptive
    moment estimation. Adam is a combination of RMSProp + Momentum.
    '''
    #print(np.shape(Y_train))
    #Y_train = Y_train.reshape(1,-1)
    #decoded = Y_train.dot(onehotencoder.active_features_).astype(int)
    # invert the one hot encoded data
    #inverted = onehotencoder.inverse_transform([argmax(Y_train[:, :])])
    #print(Y_train)
    #Y_train = Y_train.reshape(-1,1)
    #print(inverted)
    if BatchSize == 0:
        BatchSize = 100
    # Stochatic gradient descent
    sgd = SGD(lr=alpha)
    '''
```

```
            classifier.compile(optimizer = sgd, loss='binary_crossentropy',
                ↪ metrics =['accuracy'])
            #Fitting the data to the training dataset
            classifier.fit(X_train, Y_train_onehot, batch_size=BatchSize, epochs =
                ↪  epochs)
            #np.set_printoptions(precision=4, suppress=True)
            eval_results = classifier.evaluate(X_test, Y_test_onehot, verbose=0)
            print("\nLoss, accuracy on test data: ")
            print("%0.4f %0.2f%%" % (eval_results[0], eval_results[1]*100))
            '''


            #print("m is ", m)

        #else:
        #    '''
        #    Raise an exception, if number of layers is smaller than 2. It shouldn
            ↪ 't be the case,
        #    because in param file I am specifying number of hidden layers and not
            ↪  the total layers.
        #    Then I add 2 to that number in the code. But better safe than sorry
            ↪ :)
        #    '''
        #    raise Exception('No. of Layers should be >= {}! Check Parameter File
            ↪ .'.format(nLayers))


    elif (NNType == 'Regression'):
        NNType, NNArch, nLayers, \
        nFeatures, nHidden, nOutput, \
        epochs, alpha, lmbd, X, Y, X_train, \
        X_test, Y_train, Y_test, \
        x, y, z,\
        x_rav, y_rav, z_rav, zshape,\
        nInput, seed, BatchSize, optimization, z_lin = dataProc.CreateNetwork()


        #print("X is ", X)

        #print("Y is ",  Y)

        #sys.exit()

        #print("nInput", nInput)
        if optimization == 'GD': #if BatchSize == 0:
            # If batch size is zero, we will use standard Gradient Descent Method
            m = nInput
            #print("m is",m)
        elif optimization == 'MBGD': #elif BatchSize > 0:
            m = BatchSize
        elif optimization == 'Adagrad':
```

```
        m  = BatchSize
    else:
        print("Incorrect BatchSize. Check Parameter File!")
        sys.exit()



    '''
    Polynomial Regression on Franke Function
    '''
    if (nLayers == 2):
        print('''
              ====================================
               Linear Regression Via Manual Coding
              ====================================
              Activation Function is:
              No.of test data points:
              No of epochs to learn:
              Learning Rate, \u03B1:
              Regularization param, \u03BB:
                          '''#.format('function',
                             #          X_test.shape[0],
                             #          epochs,
                             #          alpha,
                             #          lmbd)+
              '''
              ====================================
              ''')
        #dataProc.ProcessData()



        print('''
              ====================================
               Linear Regression Via Scikit Learn
              ====================================
              Activation Function is:
              No.of test data points:
              No of epochs to learn:
              Learning Rate, \u03B1:
              Regularization param, \u03BB:
                          '''#.format('function',
                             #          X_test.shape[0],
                             #          epochs,
                             #          alpha,
                             #          lmbd)+
              '''
              ====================================
              ''')
```

```
        # If layer more than 2, we are using Neural Network
        if (nLayers > 2):
            '''
            Switching to Neural Network
            '''

            #m = np.size(Y_train)
            print('Neural Network')
            # passing configuration
            neuralNet = neural.NeuralNetwork(NNType, NNArch, \
                                             nLayers, nFeatures, \
                                             nHidden, nOutput, \
                                             epochs, alpha, \
                                             lmbd, nInput, seed, BatchSize,
                                                 ↪ optimization)

            #print(type(BatchSize))
            #print(X_train)
            if (BatchSize==0):
                print("Gradient Descent")

                #print("m is ", m)

                modelParams, costs = neuralNet.TrainNetworkGD(X_train, Y_train, m)
                #print(modelParams)
                #print(modelParams)

                #print('Y_train_onehot', np.shape(Y_train))
                #print('Y_train_', np.shape(Y_train))
                #print("X_train", "Y_train_onehot")



                Y_test_pred = neuralNet.MakePrediction(X_test, modelParams)


                #print(Y_test_pred)

                #print('test_predict',np.shape(Y_test))
                #print(Y_test_onehot)
                #print(test_predict)
                #print("Accuracy score on test set: ", accuracy_score(Y_test,
                    ↪ test_predict))
                #x = x_rav
                #y = y_rav
                #print("X is", X.shape)
                Y_pred = neuralNet.MakePrediction(X, modelParams)

                #print("Y_pred is ", Y_pred)

                z = Y_pred.reshape(zshape)
```

```
            #print("x is", x.shape)
            #print("y is", y.shape)
            #print("z is", z.shape)
            # takes an array of z values
            #zarray = Y_test_pred
            # output dir
            #output_dir = args[3]
            # filename
            #filename = args[4]
            #print(filename)
            # Turning interactive mode on
            #plt.ion()
            fig = plt.figure()
            axe = fig.add_subplot(1,1,1, projection = '3d')
            #axe.view_init(5,50)
            #axes = [fig.add_subplot(1, 3, i, projection='3d') for i in range
                ↪ (1, len(zarray) + 1)]
            #axes[0].view_init(5,50)
            #axes[1].view_init(5,50)
            #axes[2].view_init(5,50)
            surf = axe.plot_surface(x, y, z, alpha = 0.5,\
                                    cmap = 'brg_r', label="Franke␣function",
                                        ↪ linewidth = 0, antialiased = False
                                        ↪ )

            plt.show()
        elif (BatchSize > 0):


            #  Keras
            # to obtain the history
            #history = History()
            '''
            To optimize our neural network we use Adam. Adam stands for
                ↪ Adaptive
            moment estimation. Adam is a combination of RMSProp + Momentum.
            '''
            '''
            classifier = CallKerasModel(NNArch, nLayers)

            sgd = SGD(lr=alpha)
            classifier.compile(optimizer = sgd, loss='mean_squared_error',
                ↪ metrics =['mse'])
            #Fitting the data to the training dataset
            history = classifier.fit(X, Y, validation_split=0.2, batch_size =
                ↪ BatchSize, epochs = epochs)

            fig,ax = plt.subplots(figsize=(12, 8))
            # Plot training & validation loss values
            plt.plot(history.history['loss'])
            #plt.plot(history.history['val_loss'])
```

```
plt.title('Model loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()
filename = outputPath + '/'+ 'linreg_manual_costs_e' + str(epochs)
    ↪ .zfill(4)+'.png'
fig.savefig(filename)

# making prediction and plotting franke function
Y_pred = classifier.predict_classes(X)
z = Y_pred.reshape(zshape)
# show the inputs and predicted outputs
#print("X=%s, Predicted=%s" % (Xnew[0], ynew[0]))
fig = plt.figure(figsize=(10, 5))
axe = fig.add_subplot(1,1,1, projection = '3d')
axe.view_init(5,0)
surf = axe.plot_surface(x, y, z, alpha = 0.5,\
                        cmap = 'brg_r', label="Franke function",
                            ↪ linewidth = 0, antialiased = False
                            ↪ )
filename = outputPath + '/'+ 'linreg_kers_surf_e' + str(epochs).
    ↪ zfill(4)+'.png'
fig.savefig(filename)


plt.show()
'''

print("Mini␣Batches")
modelParams, costs = neuralNet.TrainNetworkMBGD(X_train, Y_train,
    ↪ m)#TrainNetworkMBGD(X_train, Y_train_onehot, m)
Y_test_pred = neuralNet.MakePrediction(X_test, modelParams)
#print('test_predict',np.shape(Y_test))
#print(Y_test_onehot)
#print(test_predict)
#print("Accuracy score on test set: ", accuracy_score(Y_test,
    ↪ test_predict))
#x = x_rav
#y = y_rav
#print("X is", X)
Y_pred = neuralNet.MakePrediction(X, modelParams)

#print("Y_pred is ", Y_pred)


fig,ax = plt.subplots(figsize=(12, 8))
#print(paramList)
# plotting costs
epochs1 = range(epochs)
#for i in range(len(costsList)):
```

```
plt.plot(epochs1, costs)
plt.legend(loc='upper right', bbox_to_anchor=(1.0, 1.00),
    ↪ borderaxespad=0.)
plt.ylabel("Costs")
plt.xlabel("epochs")
plt.show()
# saving cost functions
filename = outputPath + '/'+ 'nnlin_manual_costs_e' + str(epochs).
    ↪ zfill(4)+'.png'
fig.savefig(filename)

znn_pred = Y_pred.reshape(zshape)



#print("x is", x.shape)
#print("y is", y.shape)
#print("z is", z.shape)
# takes an array of z values
#zarray = Y_test_pred
# output dir
#output_dir = args[3]
# filename
#filename = args[4]
#print(filename)
# Turning interactive mode on
#plt.ion()

def PlotSurface(x, y, z, filename):

    # output dir
    #output_dir = args[3]
    # filename
    #filename = args[4]
    #print(filename)
    # Turning interactive mode on
    #plt.ion()
    fig = plt.figure(figsize=(20, 10))
    axes = [fig.add_subplot(3, 3, i, projection='3d') for i in
        ↪ range(1, len(zarray) + 1)]
    angles = [45, 0, -45, 45, 0, -45, 45, 0, -45]
    #axes[0].view_init(5,50)
    #axes[1].view_init(5,50)
    #axes[2].view_init(5,50)
    view = [axes[i].view_init(5, angles[i]) for i in range(9)]
    surf = [axes[i].plot_surface(x, y, zarray[i], alpha = 0.5,
                                 cmap = 'brg_r', label="Franke
                                     ↪ function", linewidth = 0,
                                     ↪ antialiased = False) for i
                                     ↪  in range(len(zarray))]
    # saving figure with corresponding filename
```

```
    #fig.savefig(output_dir + filename)
    # close the figure window
    #plt.close(fig)
    plt.show()

    #filename = outputPath + '/'+ 'linreg_manual_surf_e' + str(
        ↪ epochs).zfill(4)+'.png'
    fig.savefig(filename)


# getting values from previous project to compare


zarray = [z,z,z, z_lin, z_lin, z_lin, znn_pred, znn_pred, znn_pred
    ↪ ]
filename = outputPath + '/'+ 'nnlin_manual_surf_e' + str(epochs).
    ↪ zfill(4)+'.png'
PlotSurface(x, y, zarray, filename)


#fig = plt.figure(figsize=(5, 5))
#axe = fig.add_subplot(1,1,1, projection = '3d')
#axe.view_init(5, 90)
#axes = [fig.add_subplot(1, 3, i, projection='3d') for i in range
    ↪ (1, len(zarray) + 1)]
#axes[0].view_init(5,50)
#axes[1].view_init(5,50)
#axes[2].view_init(5,50)
#surf = axe.plot_surface(x, y, z, alpha = 0.5,\
#                        cmap = 'brg_r', label="Franke function",
    ↪  linewidth = 0, antialiased = False)
#filename = outputPath + '/'+ 'linreg_manual_surf_e' + str(epochs)
    ↪ .zfill(4)+'.png'
#fig.savefig(filename)


#plt.show()

# Scikit Learn

#mlpr = MLPRegressor(solver="lbfgs", alpha=alpha,
#                    batchsize = m, hidden_layer_sizes = nHidden)
#mlpr.fit(X_train, Y_train)
#yTrue, yPred = Y_test, clf.predict(X_test)
#print(classification_report(yTrue, yPred))
#print("Roc auc: ", roc_auc_score(yTrue, yPred))


#mlpr= MLPRegressor(hidden_layer_sizes=hn,
#    activation=act_h,
#    solver="adam",
```

```
                #     alpha = lmbd,
                #     learning_rate_init=eta
                #     )
                #mlpr.fit(XTrain, yTrain)

                #yTrue, yPred = yTest, mlpr.predict(XTest)
                #ypred = mlpr.predict(X)

                #R2 = mlpr.score(XTest,yTrue.ravel())



    '''
    Plotting Results
    '''
    #epochs1 = range(epochs)
    # Plotting results
    #fig,ax = plt.subplots(figsize=(12,8))

    #ax.set_ylabel('J(Theta)')
    #ax.set_xlabel('Iterations')
    #ax.plot(epochs1, costs,'b.')

    # End time of the program
    endTime = time.time()
    print("--␣Program␣finished␣at␣%s␣sec␣--" % (endTime - startTime))
```

Listing 3: "Neural Network Class"

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Wed Nov 13 11:29:25 2019

@author: maksymb
"""

import os
import sys
import numpy as np
# for polynomial manipulation
import sympy as sp
# from sympy import *
import itertools as it
import matplotlib
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sbn
# to read parameter file
import yaml

import time

import collections
from collections import Counter
# for nice progress bar
from tqdm import tqdm_notebook

# For Data preprocessing
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import OneHotEncoder, LabelEncoder, StandardScaler,
    ↪ RobustScaler
from sklearn.compose import ColumnTransformer

from sklearn.metrics import accuracy_score, mean_squared_error, log_loss

# One Hot Encoder from Keras
from keras.utils import to_categorical
import tensorflow as tf
# initialising pretty printing with sympy
# for Latex characters and more
from IPython.display import display, Latex, Markdown
from sympy import * #init_printing
#from sympy.printing.latex import print_latex
#init_printing(use_latex='mathjax')
import funclib


'''
```

```
The class used for both classification and regression
depending on the cost function and the user's desire
'''
class NeuralNetwork:
    # constructor
    def __init__(self, *args):
        # type of neuralnetwork
        self.NNType = args[0]
        # Network Architecture
        self.NNArch = args[1]
        # Total Number of layers
        self.nLayers = args[2]
        # Neurons in input layer
        self.nInputNeurons = args[3]
        # Neurons in hidden layer
        self.nHiddenNeurons = args[4]
        # Neurons in output layer
        self.nOutputNeurons = args[5]
        # number of iterations for optimization algorithm
        self.epochs = args[6]
        # learning rate
        self.alpha = args[7]
        # regularisation (hyper) parameter
        self.lambd = args[8]

        # Data
        self.nInput = args[9]
        #self.nFeatures =
        seed = args[10]
        # Batch Size
        self.BatchSize = args[11]
        #print(self.BatchSize)
        #print(type(self.BatchSize))
        # random seed, to make the same random number each time
        np.random.seed(seed)
        #self.Y = args[10]
        #self.m = args[11]
        self.optimization = args[12]
        # Only for printing purpose
        #if (self.BatchSize == 0):
        #    algorithm = 'Gradient Descent'
        #elif (self.BatchSize > 0):
        #    algorithm = 'Mini Batch Gradient Descent'
        #display(Markdown())
        print((u'''
␣␣␣␣␣␣␣␣==========================================
␣␣␣␣␣␣␣␣␣␣␣␣Start␣{}␣Neural␣Network
␣␣␣␣␣␣␣␣==========================================
␣␣␣␣␣␣␣␣No.␣of␣hidden␣layers:␣␣␣␣␣␣␣␣␣{}
␣␣␣␣␣␣␣␣No.␣of␣input␣data:␣␣␣␣␣␣␣␣␣␣␣{}
␣␣␣␣␣␣␣␣No.␣of␣input␣neurons:␣␣␣␣␣␣␣␣{}
```

```
␣␣␣␣␣␣␣␣No.␣of␣hidden␣neurons:␣␣␣␣␣␣␣{}
␣␣␣␣␣␣␣␣No.␣of␣output␣neurons:␣␣␣␣␣␣␣{}
␣␣␣␣␣␣␣␣Activ.␣Func␣in␣Hidden␣Layer:␣{}
␣␣␣␣␣␣␣␣Activ.␣Func␣in␣Output␣Layer:␣{}
␣␣␣␣␣␣␣␣No.␣of␣epochs␣to␣see:␣␣␣␣␣␣␣␣{}
␣␣␣␣␣␣␣␣Optimization␣Algorithm:␣␣␣␣␣␣{}
␣␣␣␣␣␣␣␣Learning␣Rate,␣\u03B1:␣␣␣␣␣␣␣␣␣␣␣{}
␣␣␣␣␣␣␣␣Regularization␣param,␣\u03BB:␣␣␣␣␣{}
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣''').format(self.NNType,
                                    self.nLayers-2,
                                    self.nInput,
                                    self.nInputNeurons,
                                    self.nHiddenNeurons,
                                    self.nOutputNeurons,
                                    self.NNArch[1]['AF'],
                                    self.NNArch[self.nLayers-1]['AF'],
                                    self.epochs,
                                    self.optimization,
                                    self.alpha,
                                    self.lambd)))

    '''
    To address these issues, Xavier and Bengio (2010) proposed the
        Xavier     initialization which considers the size of the network
    (number of input and output units) while initializing weights.
    This approach ensures that the weights stay within a reasonable
    range of values by making them inversely proportional to the square
    root of the number of units in the previous layer (referred to as fan-in).
    '''
    '''
    To prevent the gradients of the networks activations from vanishing or
    exploding, we will stick to the following rules of thumb:

        The mean of the activations should be zero.
        The variance of the activations should stay the same across every layer.
    '''
    '''
    Used for tanh, sigmoid etc.
    '''
    def CallXavier(self, *args):
        n_l = args[0]
        n_next = args[1]
        #xav = np.random.uniform(-1, 1, size=(n_l, n_next)) * np.sqrt(6.0 / (n_l +
            ↪  n_next))
        xav = np.random.randn(n_l, n_next)*np.sqrt(2/(n_l+n_next))
        return xav

    '''
    Used for ReLU
    '''
    def CallKaiming(self, *args):
```

```
        m = args[0]
        h = args[1]
        kaim = np.random.rand(m, h) * np.sqrt(2./m)
        return kaim#torch.randn()

    '''
    Initialising weights' for training. I am using Xavier and He
    (here I call it Kaiming from Kaiming He <= you got the idea :),
    but, for some reason, it doesn't help much for Linear Regression,
    so later I am trying to do batch Normalisation.
    '''
    def InitParams(self, *args):
        NNArch = args[0]
        nInput = args[1]
        # biases and weights for hidden and output layers
        # dictionary to contain all parameters for each layer
        # (i.e. "W1", "b1", ..., "WL", "bL", except inpur one)
        self.modelParams = {}
        self.update_params={}

        for l in range(1, self.nLayers):
            #print(self.NNArch[l]["LSize"])
            if self.NNArch[l]['AF'] == 'sigmoid':
                self.modelParams['W' + str(l)] = self.CallXavier(self.NNArch[l-1][
                    ↪ "LSize"],self.NNArch[l]["LSize"])
                self.modelParams['b' + str(l)] = np.random.randn(NNArch[l]["LSize"
                    ↪ ]) #+ 1
            elif self.NNArch[l]['AF'] == 'tanh':
                self.modelParams['W' + str(l)] = self.CallXavier(self.NNArch[l-1][
                    ↪ "LSize"],self.NNArch[l]["LSize"])
                self.modelParams['b' + str(l)] = np.random.randn(NNArch[l]["LSize"
                    ↪ ]) #+ 1
            elif self.NNArch[l]['AF'] == 'softmax':
                self.modelParams['W' + str(l)] = self.CallXavier(self.NNArch[l-1][
                    ↪ "LSize"],self.NNArch[l]["LSize"])
                self.modelParams['b' + str(l)] = np.random.randn(NNArch[l]["LSize"
                    ↪ ]) #+ 1
            elif self.NNArch[l]['AF'] == 'relu':
                self.modelParams['W' + str(l)] = self.CallKaiming(self.NNArch[l
                    ↪ -1]["LSize"],self.NNArch[l]["LSize"])
                self.modelParams['b' + str(l)] = np.random.randn(NNArch[l]["LSize"
                    ↪ ]) #+ 1
            elif self.NNArch[l]['AF'] == 'elu':
                self.modelParams['W' + str(l)] = self.CallKaiming(self.NNArch[l
                    ↪ -1]["LSize"],self.NNArch[l]["LSize"])
                self.modelParams['b' + str(l)] = np.random.randn(NNArch[l]["LSize"
                    ↪ ]) #+ 1
            else:
                self.modelParams['W' + str(l)] = self.CallKaiming(self.NNArch[l
                    ↪ -1]["LSize"],self.NNArch[l]["LSize"])
                self.modelParams['b' + str(l)] = np.random.randn(NNArch[l]["LSize"
```

```
                ↪ ]) #+ 1

            self.update_params["v_w"+str(l)]=0
            self.update_params["v_b"+str(l)]=0
        #return self.modelParams



    # Getting output for each layer
    def GetA(self, *args):
        Z = args[0]
        l = args[1]
        # (['sigmoid', 'tanh', 'relu', 'softmax'])
        #print('AF is ' + self.NNArch[l]['AF'])
        if self.NNArch[l]['AF'] == 'sigmoid':
            A = funclib.ActivationFuncs().CallSigmoid(Z)
        elif self.NNArch[l]['AF'] == 'relu':
            # manual version:
            A = funclib.ActivationFuncs().CallReLU(Z)
            #print(A)
            # tensor flow version:
            #A = tf.nn.relu(Z)
            #print(A)
        elif self.NNArch[l]['AF'] == 'tanh':
            A = funclib.ActivationFuncs().CallTanh(Z)
        elif self.NNArch[l]['AF'] == 'softmax':
            A = funclib.ActivationFuncs().CallSoftmax(Z)
        elif self.NNArch[l]['AF'] == 'linear':
            A = funclib.ActivationFuncs().CallIdentity(Z)
        elif self.NNArch[l]['AF'] == 'elu':
            A = funclib.ActivationFuncs().CalleLU(Z)
        return A

    '''
    Doing BAtch Normalisation - in case of simple gradient
    descent the batch size is the entire data set.
    This method is used during Feed Forward, for back propagation
    there is another one below!
    '''
    '''
    def DoBatchNormalisation(self, *args):
        X = args[0] # <= these are not actually X, but Z values
        gamma = args[1]
        beta = args[2]
        # parameters
        eps = 1e-5

        #print(X.shape)

        # checking for matrix to be of the shape 2
        #if X.shape == 2:
```

```
        # mean of the mini-batch
        batchMean = np.mean(X, axis=0)
        # variance of the mini-batch
        batchVar = np.var(X, axis=0)
        # normalising
        X_norm = (X - batchMean) * 1.0 / np.sqrt(batchVar + eps)
        # scaled output
        X_out = gamma * X_norm + beta
        # people are usually storing stuff in cache so I will do the same thing
        cache = (X, X_norm, batchMean, batchVar, gamma, beta)

        return X_out, cache, batchMean, batchVar
        #else:
        #    print("Batch Normalisation: Check your X shape!")
        #    sys.exit()
    '''

    '''
    This one to implement Batch Normalisationwith Back Propagation
    '''
    '''
    def DoDBatchNormalisation(self, *args):
        dout = args[0]
        cache = args[1]
        eps = 1e-5
        # getting values from cache
        X, X_norm, mu, var, gamma, beta = cache

        N, D = X.shape

        X_mu = X - mu
        std_inv = 1. / np.sqrt(var + eps)

        dX_norm = dout * gamma
        dvar = np.sum(dX_norm * X_mu, axis=0) * -.5 * std_inv**3
        dmu = np.sum(dX_norm * -std_inv, axis=0) + dvar * np.mean(-2. * X_mu, axis
            ↪ =0)

        dX = (dX_norm * std_inv) + (dvar * 2 * X_mu / N) + (dmu / N)
        dgamma = np.sum(dout * X_norm, axis=0)
        dbeta = np.sum(dout, axis=0)

        return dX, dgamma, dbeta
    '''

    '''
    Feed Forward seems to return correct (!) shapes
    '''

    # Method to Feed Forward Propagation
    def DoFeedForward(self, *args):
```

```
        X = args[0] # <= it will be of batch size (for Mini-Batch GD)
        #self.modelParams = args[1]
        # parameters for batch normalisation
        gamma = 0.01
        beta  = 0.01
        A = {}
        Z = {}
        # Values for Input Layer
        A['0'] = X
        Z['0'] = X

        #cache['0'] = 0

        # compute model for each layer and
        # apply corresponding activation function
        for l in range(1, self.nLayers):
            #print("W"+str(l)+" is", np.shape(modelParams['W' + str(l)]))
            #print("b"+str(l)+" is", modelParams['b' + str(l)])
            # z for each layer
            Z[str(l)] = np.matmul(A[str(l-1)], self.modelParams['W' + str(l)]) +
                ↪ self.modelParams['b' + str(l)]



            # Doing batch Normalisation <= updating Z values for each batch
            # We need to do normalisation only for training

            #Z[str(l)], cache, mu, var = self.DoBatchNormalisation(Z[str(l)],
                ↪ gamma, beta)

            #if np.isnan(Z[str(l)]):
            #print("Z is", Z[str(l)])
            #print(Z[str(l)])
            # applying corresponding activation function
            A[str(l)] = self.GetA(Z[str(l)], l)


        # returning dictionary of outputs
        return A, Z

    # Method which decides, whioch
    # derivative of cost function to use
    def GetdAF(self, *args):
        A = args[0]
        l = args[1]
        # (['sigmoid', 'tanh', 'relu', 'softmax'])
        #print('dAF is ' + self.NNArch[l]['AF'])
        if self.NNArch[l]['AF'] == 'sigmoid':
            dAF = funclib.ActivationFuncs().CallDSigmoid(A)
        elif self.NNArch[l]['AF'] == 'relu':
            dAF = funclib.ActivationFuncs().CallDReLU(A)
```

```
        elif self.NNArch[l]['AF'] == 'tanh':
            dAF = funclib.ActivationFuncs().CallDTanh(A)
        elif self.NNArch[l]['AF'] == 'softmax':
            dAF = funclib.ActivationFuncs().CallDSoftmax(A)
        elif self.NNArch[l]['AF'] == 'linear':
            dAF = funclib.ActivationFuncs().CallDIdentity(A)
        elif self.NNArch[l]['AF'] == 'elu':
            dAF = funclib.ActivationFuncs().CallDeLU(A)

        return dAF

    # Method to do Back Propagation
    # (to calculate gradients of J)
    def DoBackPropagation(self, *args):
        # getting values for each layer
        #A, Z = self.DoFeedForward()
        Y = args[0]
        A = args[1]
        #Z = args[2]
        #self.modelParams = args[3]
        m = args[4]

        X = args[5]
        # errors for output layer
        delta = {}
        #delta[str(self.nLayers-1)] = A[str(self.nLayers-1)] - self.Y
        # gradients of the cost function
        # (for each layer)
        dJ = {}
        #print(self.NNType)
        # Calculating gradients of the cost function for each layer
        # (going from last to first hidden layer)
        if self.NNType == "Classification":
            for l in reversed(range(1, self.nLayers)):
                #print(l)
                # calculating error for each layer
                if (l == self.nLayers - 1):
                    delta[str(l)] = A[str(l)] - Y
                    # gradients of output layer (+ regularization)
                    # W^{l} = A^{l-1} * delta^{l}
                    '''
                    dJ['dW'+str(l)] = (1 / m) * np.matmul(A[str(l-1)].T, delta[str
                        ↪ (l)]) \
                    + self.lambd * modelParams['W' + str(l)] / m
                    dJ['db'+str(l)] = (1 / m) * np.sum(delta[str(l)], axis=0)#,
                        ↪ keepdims=True)
                    '''
                    dJ['dW'+str(l)] = (np.matmul(A[str(l-1)].T, delta[str(l)]) \
                    + self.lambd * self.modelParams['W' + str(l)]) #/ m
                    dJ['db'+str(l)] = np.sum(delta[str(l)], axis=0) #/ m#,
                        ↪ keepdims=True)
```

```
            else:
                dAF = self.GetdAF(A[str(l)], l)
                delta[str(l)] = np.multiply(np.matmul(delta[str(l+1)], self.
                    ↪ modelParams['W' + str(l+1)].T), dAF)
                # gradients of the hidden layer
                # W^{l} = A^{l-1} * delta^{l}
                '''
                dJ['dW'+str(l)] = (1 / m) * np.matmul(A[str(l-1)].T, delta[str
                    ↪ (l)]) \
                + self.lambd * modelParams['W' + str(l)] / m
                dJ['db'+str(l)] = (1 / m) * np.sum(delta[str(l)], axis=0)#,
                    ↪ keepdims=True)
                '''
                dJ['dW'+str(l)] = (np.matmul(A[str(l-1)].T, delta[str(l)]) \
                + self.lambd * self.modelParams['W' + str(l)]) #/ m
                dJ['db'+str(l)] = np.sum(delta[str(l)], axis=0) #/ m#,
                    ↪ keepdims=True)
            #print("dW"+str(l), dJ['dW'+str(l)].shape)
            #print("db"+str(l), dJ['db'+str(l)].shape)
            #print(dJ['dW'+str(l)])
            #if np.isnan(delta[str(l)].columns.values):
        #    print('NaN values spotted')
    elif self.NNType == 'Regression':
        for l in reversed(range(1, self.nLayers)):
            #print(l)
            # calculating error for each layer
            if (l == self.nLayers - 1):
                #delta[str(l)] = 0
                #print("Y", Y.shape)
                # delta_L = (A_L - Y) * A_L <= notice matrix multiplication
                delta[str(l)] = np.matmul((A[str(l)] - Y), A[str(l)])#, A[str(
                    ↪ l)]) #self.lambd * (A[str(l)] - Y) * A[str(l)] / m
                #print('delta'+str(l), delta[str(l)].shape)
                #beta = beta - alpha * (X.T.dot(X.dot(beta)-y)/m)
                # gradients of output layer (+ regularization)
                # W^{l} = A^{l-1} * delta^{l}
                dJ['dW'+str(l)] = (np.matmul(A[str(l-1)].T, delta[str(l)]) \
                + self.lambd * self.modelParams['W' + str(l)])# / m
                #print(dJ['dW'+str(l)])
                dJ['db'+str(l)] = np.sum(delta[str(l)], axis=0) / m#, keepdims
                    ↪ =True)
                # clipping gradients:
                #if dJ['dW'+str(l)].any() > 10 or dJ['dW'+str(l)].any() < -10:
                #    dJ['dW'+str(l)] = np.slip(dJ['dW'+str(l)], 0, 1)
                #if dJ['db'+str(l)].any() > 10 or dJ['db'+str(l)].any() < -10:
                #    dJ['db'+str(l)] = np.slip(dJ['db'+str(l)], 0, 1)
            else:
                #dAF = funclib.ActivationFuncs().CallDSigmoid(A[str(l)])
                # Calling derivative of current activation function
                dAF = self.GetdAF(A[str(l)], l)
                delta[str(l)] = np.multiply(np.matmul(delta[str(l+1)], self.
```

```
                    ↪ modelParams['W' + str(l+1)].T), dAF)
                #print("delta"+str(l), delta[str(l)].shape)
                # gradients of the hidden layer
                # W^{l} = A^{l-1} * delta^{l}
                dJ['dW'+str(l)] = (np.matmul(A[str(l-1)].T, delta[str(l)]) \
                + self.lambd * self.modelParams['W' + str(l)]) / m
                dJ['db'+str(l)] = np.sum(delta[str(l)], axis=0) / m#, keepdims
                    ↪ =True)

                # clipping gradients:
                #if dJ['dW'+str(l)].any() > 10 or dJ['dW'+str(l)].any() < -10:
                #    dJ['dW'+str(l)] = np.slip(dJ['dW'+str(l)], 0, 1)
                #if dJ['db'+str(l)].any() > 10 or dJ['db'+str(l)].any() < -10:
                #    dJ['db'+str(l)] = np.slip(dJ['db'+str(l)], 0, 1)

            #print("dW"+str(l), dJ['dW'+str(l)])
            #print("db"+str(l), dJ['db'+str(l)].shape)

    return dJ

# Method to Update Weights
# (on each iteration)
def UpdateWeights(self, *args):
    dJ = args[0]
    #self.modelParams = args[1]
    m = args[2]

    gamma = 0.9
    eps=1e-8

    algorithm = self.optimization

    # Different versions of weights updates :)
    if algorithm == 'GD':
        for l in range(1, self.nLayers):
            self.modelParams['W' + str(l)] -= dJ['dW' + str(l)] * self.alpha /
                ↪  (m)# * np.sqrt(6.0 / (n_l + n_next)) # <= this one should
                ↪ be correct
            self.modelParams['b' + str(l)] -= dJ['db' + str(l)] * self.alpha /
                ↪  (m)# * np.sqrt(6.0 / (n_l + n_next)) # <= this one should
                ↪ be correct
    elif algorithm == 'MBGD':
        for batch in range(self.BatchSize):
            for l in range(1, self.nLayers):
                self.modelParams['W' + str(l)] -= dJ['dW' + str(l)] * self.
                    ↪ alpha / (m)# * np.sqrt(6.0 / (n_l + n_next)) # <= this
                    ↪ one should be correct
                self.modelParams['b' + str(l)] -= dJ['db' + str(l)] * self.
                    ↪ alpha / (m)# * np.sqrt(6.0 / (n_l + n_next)) # <= this
                    ↪ one should be correct
    elif algorithm == "Momentum":
```

```
            for l in range(1, self.nLayers):
                self.update_params["v_w"+str(l)] = gamma *self.update_params["v_w"
                    ↪ +str(l)] + self.alpha * (dJ['dW' + str(l)]/m)
                self.update_params["v_b"+str(l)] = gamma *self.update_params["v_b"
                    ↪ +str(l)] + self.alpha * (dJ['db' + str(l)]/m)
                self.modelParams["W"+str(l)] -= self.update_params["v_w"+str(l)]
                self.modelParams["b"+str(l)] -= self.update_params["v_b"+str(l)]
        elif algorithm == "Adagrad":
            for l in range(1, self.nLayers):
                self.update_params["v_w"+str(l)] += (dJ['dW' + str(l)]/m)**2
                self.update_params["v_b"+str(l)] += (dJ['db' + str(l)]/m)**2
                self.modelParams["W"+str(l)] -= (self.alpha/(np.sqrt(self.
                    ↪ update_params["v_w"+str(l)])+eps)) * (dJ['dW' + str(l)]/m)
                self.modelParams["b"+str(l)] -= (self.alpha/(np.sqrt(self.
                    ↪ update_params["v_b"+str(l)])+eps)) * (dJ['db' + str(l)]/m)


        #return self.modelParams

    # Train Neural Network using Gradient Descent
    def TrainNetworkGD(self, *args):
        X_train = args[0]
        Y_train = args[1]
        #print(Ytrain)
        m = args[2]

        #print("NN m is", m)

        # Initialising parameters
        #self.modelParams = self.InitParams(self.NNArch, self.nInput)
        self.InitParams(self.NNArch, self.nInput)

        costs =  []
        if self.NNType == 'Classification':
            #tqdm_notebook(range(epochs), total=epochs, unit="epoch")
            # Running Optimisation Algorithm
            for epoch in tqdm_notebook(range(self.epochs), total=self.epochs, unit
                ↪ ="epoch"):#range(0, self.epochs, 1):
                # Propagating Forward
                A, Z = self.DoFeedForward(X_train, self.modelParams)
                # Calculating cost Function
                J = funclib.CostFuncs().CallNNLogistic(Y_train,\
                                        A[str(self.nLayers-1)],\
                                        self.modelParams,\
                                        self.nLayers,\
                                        m,\
                                        self.lambd, X_train)
                #print(J)
                # Back propagation - gradients
                dJ = self.DoBackPropagation(Y_train, A, Z, self.modelParams, m,
                    ↪ X_train)
```

```
                #print(dJ)
                # updating weights
                #modelParams = self.UpdateWeights(dJ, self.modelParams, m)
                self.UpdateWeights(dJ, self.modelParams, m)
                # getting values of cost function at each epoch
                if(epoch % 1 == 0):
                    print('Cost after iteration# {:d}: {:f}'.format(epoch, J))
                costs.append(J)
            # returning set of optimal model parameters
            return self.modelParams, costs

        elif self.NNType == 'Regression':
            # Running Optimisation Algorithm
            for epoch in tqdm_notebook(range(self.epochs), total=self.epochs, unit
                ="epoch"):#for epoch in range(1, self.epochs+1, 1):
                # Propagating Forward
                A, Z = self.DoFeedForward(X_train, self.modelParams)
                # Calculating cost Function
                J = funclib.CostFuncs().CallNNMSE(Y_train,\
                                    A[str(self.nLayers-1)],\
                                    self.modelParams,\
                                    self.nLayers,\
                                    m,\
                                    self.lambd, X_train)
                #print(J)
                # Back propagation - gradients
                dJ = self.DoBackPropagation(Y_train, A, Z, self.modelParams, m,
                    X_train)
                #print(dJ)
                # updating weights
                #self.modelParams = self.UpdateWeights(dJ, self.modelParams, m)
                self.UpdateWeights(dJ, self.modelParams, m)

                #print(modelParams)
                #print(epoch, J, dJ)
                # getting values of cost function at each epoch
                #if(epoch % 1 == 0):
                #    print('Cost after iteration# {:d}: {:f}'.format(epoch, J))
                costs.append(J)
            # returning set of optimal model parameters
            return self.modelParams, costs
        else:
            Exception('It is neither Regression nor Classification task! Check
                Parameter File.')



    # Train Neural Network using Mini Batch Stochastic Gradient Descent
    def TrainNetworkMBGD(self, *args):
        X_train = args[0]
```

```
        Y_train = args[1]
        #print(Ytrain)
        m = self.BatchSize
        #print(self.CreateMiniBatches(Xtrain, Ytrain, self.BatchSize))
        indices = np.arange(self.nInput)
        costs =  []
        # sum the values with same keys
        #counter = collections.Counter()
        cost = 0
        self.nBatches = self.nInput // m

        print("nBatches", self.nBatches)

        #print(nBatches)
        # getting all the indecis from inputs
        indices = np.arange(self.nInput)
        # Initialising parameters - ensuring
        '''
        Let theta = model parameters and max_iters = number of epochs.

        for itr = 1, 2, 3,     , max_iters:
              for mini_batch (X_mini, y_mini):

        Forward Pass on the batch X_mini:
        Make predictions on the mini-batch
        Compute error in predictions (J(theta)) with the current values of the
            ↪ parameters
        Backward Pass:
        Compute gradient(theta) = partial derivative of J(theta) w.r.t. theta
        Update parameters:
        theta = theta      learning_rate*gradient(theta)
        '''
        # that we starting at the spot in the parameter space
        #self.modelParams = self.InitParams(self.NNArch, m)
        self.InitParams(self.NNArch, m)

        if self.NNType == 'Classification':
            # Running Optimisation Algorithm
            for epoch in tqdm_notebook(range(self.epochs), total=self.epochs, unit
                ↪ ="epoch"):#for epoch in range(1, self.epochs+1, 1):
                # looping through all batches
                for j in range(self.nBatches):
                    # chosing the indexes for playing around
                    points = np.random.choice(indices, size=m, replace=False)
                    X_train_batch = X_train[points]
                    Y_train_batch = Y_train[points]
                    #print(np.shape(X_train_batch))
                    # Propagating Forward
                    A, Z = self.DoFeedForward(X_train_batch, self.modelParams)
                    # Calculating cost Function
                    J = funclib.CostFuncs().CallNNLogistic(Y_train_batch,\
```

```
                                        A[str(self.nLayers-1)],\
                                        self.modelParams,\
                                        self.nLayers,\
                                        m,\
                                        self.lambd)
                    # trying scikit lea     <= didn't work out very well
                    #J = log_loss(np.argmax(Y_train_batch, axis=1), A[str(self.
                        ↪ nLayers-1)])
                                       # Back propagation - gradients
                    dJ = self.DoBackPropagation(Y_train_batch, A, Z, self.
                        ↪ modelParams, m, X_train)
                    # updating weights
                    #self.modelParams = self.UpdateWeights(dJ, self.modelParams, m
                        ↪ )
                    self.UpdateWeights(dJ, self.modelParams, m)

                    cost += J/self.nInput
                if(epoch % 1 == 0):
                    print('Cost after iteration# {:d}: {:f}'.format(epoch, J))
                costs.append(J)

        return self.modelParams, costs

    elif self.NNType == 'Regression':
        # Running Optimisation Algorithm
        for epoch in tqdm_notebook(range(self.epochs), total=self.epochs, unit
            ↪ ="epoch"):#for epoch in range(1, self.epochs+1, 1):
            #points = np.random.permutation(indices, size=m)
            # looping through all batches
            for j in range(self.nBatches):
                # chosing the indexes for playing around
                points = np.random.choice(indices, size=m, replace=False)
                X_train_batch = X_train[points]
                Y_train_batch = Y_train[points]
                #print(np.shape(X_train_batch))
                # Propagating Forward
                A, Z = self.DoFeedForward(X_train_batch, self.modelParams)
                # Calculating cost Function
                J = funclib.CostFuncs().CallNNMSE(Y_train_batch,\
                                        A[str(self.nLayers-1)],\
                                        self.modelParams,\
                                        self.nLayers,\
                                        m,\
                                        self.lambd, X_train_batch)
                                       # Back propagation - gradients
                dJ = self.DoBackPropagation(Y_train_batch, A, Z, self.
                    ↪ modelParams, m, X_train_batch)
                # updating weights
                #self.modelParams = self.UpdateWeights(dJ, self.modelParams, m
                    ↪ )
                self.UpdateWeights(dJ, self.modelParams, m)
```

```
                cost += J/self.nInput

            #print(modelParams)
            #print(epoch, dJ)
            #if(epoch % 1 == 0):
            #    print('Cost after iteration# {:d}: {:f}'.format(epoch, J))
            costs.append(J)
        # returning set of optimal model parameters
        return self.modelParams, costs
    else:
        Exception('It is neither Regression nor Classification task! Check
            ↪ Parameter File.')

'''
Function which will fit the test data
'''
def MakePrediction(self, *args):
    # making prediction for the data set
    X_test = args[0]

    modelParams = args[1]

    # making a prediction
    A, Z = self.DoFeedForward(X_test, modelParams)#self.feed_forward_out(X)

    if self.NNType == 'Classification':
        # outputting the values which corresponds to the maximum probability
        return np.argmax(A[str(self.nLayers-1)], axis=1)
    elif self.NNType == 'Regression':
        return A[str(self.nLayers-1)]
```

Listing 4: "Regression Class"

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Tue Nov 19 09:56:48 2019

@author: maksymb
"""

import numpy as np
# for polynomial manipulation
import sympy as sp
# from sympy import *
import itertools as it

import multiprocessing as mp
from joblib import Parallel, delayed

from mpl_toolkits.mplot3d import Axes3D
import matplotlib
import matplotlib.pyplot as plt
import seaborn as sbn

from sklearn.preprocessing import PolynomialFeatures
# regression libraries
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import Ridge
from sklearn.linear_model import Lasso
from sklearn.metrics import mean_squared_error
# to split data for testing and training - KFold cross validation implementation
from sklearn.model_selection import train_test_split
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score

# Scikitlearn imports to check results
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import GridSearchCV, train_test_split
from sklearn.metrics import confusion_matrix, accuracy_score, roc_auc_score,
    ↪ classification_report, f1_score
from sklearn.preprocessing import LabelEncoder, OneHotEncoder, StandardScaler
from numpy import argmax

# We'll need some metrics to evaluate our models
from sklearn.neural_network import MLPClassifier, MLPRegressor


import keras
# stochastic gradient descent
from keras.optimizers import SGD
from keras.models import Sequential
from keras.layers import Dense
```

```
import funclib
import data_processing

'''
Class, which handles both Logistic and Linear Regressions
'''
class RegressionPipeline:
    # constructor
    def __init__(self, *args):
        # Variables common to both of them
        pass

    def DoLinearRegression(self, *args):
        #======================================================================#
        # Liner Regression variables
        #======================================================================#
        # symbolic variables
        #x_symb = args[0]
        # array of values for each variable/feature
        #x_vals = args[1]
        # grid values
        #x = args[2]
        #y = args[3]
        #z = args[4]
        # 1.96 to calculate stuff with 95% confidence
        #confidence = args[5]
        # noise variance - for confidence intervals estimation
        #sigma = args[6]
        # k-value for Cross validation
        #kfold = args[7]
        # hyper parameter
        #lambda_par = args[8]
        # directory where to store plots
        #output_dir = args[9]
        #prefix = args[10]
        # degree of polynomial to fit
        #poly_degree = args[11]
        #======================================================================#
        funcNormal = funclib.NormalFuncs()
        funcError = funclib.ErrorFuncs()
        funcPlot  = funclib.PlotFuncs()
        X = args[0]
        x = args[1]
        y = args[2]
        z = args[3]
        x_rav = args[4]
        y_rav = args[5]
        z_rav = args[6]
        zshape = args[7]
        poly_degree = args[8]
```

```
        lambda_par = args[9]
        sigma = args[10]
        outputPath = [11]
        # getting the design matrix
        #X = func.ConstructDesignMatrix(x_symb, x_vals, poly_degree)
        # getting the Ridge/OLS Regression via direct multiplication
        ztilde_ridge = funcNormal.CallNormal(X, z_rav, lambda_par, sigma)
        ztilde_ridge = ztilde_ridge.reshape(zshape)

        ''' Scikit Learn '''
        poly_features = PolynomialFeatures(degree = poly_degree)
        X_scikit = np.swapaxes(np.array([x_rav, y_rav]), 0, 1)
        X_poly = poly_features.fit_transform(X_scikit)
        ridge_reg = Ridge(alpha = lambda_par, fit_intercept=True).fit(X_poly,
            ↪ z_rav)
        ztilde_sk = ridge_reg.predict(X_poly).reshape(zshape)
        zarray_ridge = [z, ztilde_ridge, ztilde_sk]
        print('\n')
        print("Ridge MSE (no CV) - " + str(funcError.CallMSE(zarray_ridge[0],
            ↪ zarray_ridge[1])) + "; sklearn - " + str(mean_squared_error(
            ↪ zarray_ridge[0], zarray_ridge[2])))
        print("Ridge R^2 (no CV) - " + str(funcError.CallR2(zarray_ridge[0],
            ↪ zarray_ridge[1])) + "; sklearn - " + str(ridge_reg.score(X_poly,
            ↪ z_rav)))
        print('\n')
        ''' Plotting Surfaces '''
        filename = 'ridge_p' + str(poly_degree).zfill(2) + '_n.png'
        funcPlot.PlotSurface(x, y, zarray_ridge, outputPath, filename)
        #filename = self.prefix + '_ridge_p' + str(self.poly_degree).zfill(2) + '
            ↪ _n' + npoints_name + '.png'
        #lib.plotSurface(self.x, self.y, zarray_ridge, self.output_dir, filename)

        #PlotSurface.PlotFuncs()
        return ztilde_ridge



    def DoLogisticRegression(self, *args):
        #========================================================================#
        # Logistic Regression variables
        #========================================================================#
        X_train = args[0]
        Y_train_onehot = args[1]
        epochs   = args[2]
        lmbd = args[3]
        alpha = args[4]
        '''
        Part 1: Implementing Logistic Regression via gradient Descent.
        Batch gradient descent and usual GD are implemented for Part 2:
        NN Logistic Regression.
        '''
```

```
        activeFuncs = funclib.ActivationFuncs()
        costFuncs = funclib.CostFuncs()
        theta = np.zeros((X_train.shape[1], 1))
        epochs1 = range(epochs)
        #if BatchSize == 0:
        m = len(Y_train_onehot)
        costs = []
        for epoch in epochs1:
            Y_pred = np.dot(X_train, theta)
            A = activeFuncs.CallSigmoid(Y_pred)#CallSigmoid(Y_pred)
            # cost function
            J, dJ = costFuncs.CallLogistic(X_train, Y_train_onehot, A)
            # Adding regularisation
            J = J + lmbd / (2*m) * np.sum(theta**2)
            if np.isnan(J):
                print("J is ", J)
            dJ = dJ + lmbd * theta / m
            # updating weights
            theta = theta - alpha * dJ #+ lmbd * theta/m
            # updating cost func history
            costs.append(J)
            # getting values of cost function at each epoch
            #if(epoch % 100 == 0):
            #    print('Cost after iteration# {:d}: {:f}'.format(epoch, J))

        #print("Old accuracy on training data: " + str(accuracy_score(predict(
            ↪ X_train), Y_train_onehot)))

        return costs, theta

    def PredictLogisticRegression(self,  *args):
        X = args[0]
        theta = args[1]
        #def predict(self,X,betas=[]):                                  # Calculates
            ↪ probabilities and onehots for y
        #if (len(betas)>0):
        #    beta=betas.copy()
        #else:
        #    beta=self.beta.copy()
        #print("Predicting y using logreg")
        # Returns probabilities
        activeFuncs = funclib.ActivationFuncs()
        A = activeFuncs.CallSigmoid(X @ theta)
        #print('A is ', A)
        Y_pred = (A > 0.5).astype(int)
        #Y_pred_onehot = self.initdata.onehotencoder.fit_transform(Y_pred.reshape
            ↪ (-1,1)) # Converts to onehot
        return Y_pred
```

Listing 5: "Functions' Library"

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Wed Nov 13 15:31:57 2019

@author: maksymb
"""

'''
Library Module
'''


# library imports
import os
import sys
import numpy as np
import math as mt
# for polynimial manipulation
import sympy as sp
import itertools as it
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import seaborn as sbn

from collections import Counter

# trying another loss calculation
from keras import backend as K

import time

'''
Class which contains all activation functions
(functions are taken from lecture slides)
'''
class ActivationFuncs:
    # class constructor
    def __init__(self, *args):
        pass

    # Sigmoid Function
    def CallSigmoid(self, *args):
        z = args[0]
        return 1 / (1 + np.exp(-z))

    # Derivative of sigmoid
    def CallDSigmoid(self, *args):
        z = args[0]
        p = self.CallSigmoid(z)
```

```
        return p * (1 - p)

    # tanh Function
    def CallTanh(self, *args):
        z = args[0]
        return np.tanh(z)

    # tanh'
    def CallDTanh(self, *args):
        z = args[0]
        return 1 - self.CallTanh(z)**2


    # Rectified Linear Unit Function <= need to check this one
    def CallReLU(self, *args):
        z = args[0]
        #z[z < 0] = 0
        return np.maximum(z, 0)

    # ReLU's derivative
    def CallDReLU(self, *args):
        z = args[0]
        #if z < 0:
        #    return 0
        #elif z >= 0:
        #    return (z>0)
        return 1.0 * (z > 0)

    # Softmax function
    def CallSoftmax(self, *args):
        # We need to normalize this function, otherwise we will
        # get nan in the output
        z = args[0]

        # We can choose an arbitrary value for log(C) term,
        # but generally log(C)= max (a) is chosen, as it shifts
        # all of elements in the vector to negative to zero,
        # and negatives with large
        #p = np.exp(z - np.max(z))#, axis=1, keepdims = True))
        #return p / np.sum(p, axis=0)#np.sum(np.exp(z), axis=1, keepdims=True)
        p = np.exp(z - np.max(z))#np.exp(z)
        return p / np.sum(p, axis=1, keepdims=True)

    # Softmax gradient (in Vectorized form,
    # also possible to write in element wise)
    def CallDSoftmax(self, *args):
        z = args[0]
        #print('Softmax, z shape is', z.shape)
        m = z.shape[0]
        p = self.CallSoftmax(z)
        #p = p.reshape(-1,1)
```

```
        #jacobian_m = np.diag(p)
        #for i in range(len(jacobian_m)):
        #    for j in range(len(jacobian_m)):
        #        if i == j:
        #            jacobian_m[i][j] = p[i] * (1-p[i])
        #        else:
        #            jacobian_m[i][j] = -p[i]*p[j]
        #p = p.reshape(-1,1)
        #trying something else
        return p * (1 - p) #np.diagflat(p)#jacobian_m#p * (1 - p)#np.diagflat(p) -
            ↪  np.matmul(p, p.T)

    # elu
    def CalleLU(self, *args):
        z = args[0]
        return np.choose(z < 0, [z, (np.exp(z)-1)])

    def CallDeLU(self, *args):
        z = args[0]
        return np.choose(z > 0, [1,  np.exp(z)])

        # identity
    def CallIdentity(self, *args):
        z = args[0]
        return z

    def CallDIdentity(self, *args):
        return 1

'''
Class which contains all Gradient Methods:
Gradient Descent Method, Stochastic gradient Descent,
Batch Gradient etc.
'''
class OptimizationFuncs:
    # class constructor
    def __init__(self, *args):
        self.activeFunc = ActivationFuncs()

    # Gradient Descent Method
    # rewrite it so it will be possible to use it for both regressions and NN
    # (just need to pass somehow J and dJ as they are the only difference)
    def SimpleGD(self, *args):
        # getting inputs
        X = args[0]
        y = args[1]
        theta = args[3]
        alpha = args[4]
        # total number of iterations
        epochs = args[5]
```

```
        # number of features
        m = len(y)
        # saving cost history (to make a plots out of it)
        costs = []
        # applying gradient descent algorithm
        for epoch in epochs:
            # our model
            y_pred = np.dot(X, theta)
            # applying sigmoid
            h = self.activeFunc.CallSigmoid(y_pred)
            # calculating cost
            #J = -np.sum(y*np.log(h) +(1-y)*np.log(1-h)) / m
            # calculating gradient of the cost function
            #dJ = np.dot(X.T, h - y) / m
            J, dJ = CostFuncs().CallLogistic(X, y, h)
            # updating weights
            theta = theta - alpha * dJ
            # saving current cost function for future reference
            costs.append(J)

        return theta#, costs

    # Stochastic Gradient Descent Method
    def StochasticGD(self, *args):
        return

    # Stochastic Gradient Descent Method with Batches
    def BatchedSGD(self, *args):
        return

        # Stochastic gradient descent method with batches


'''
Class which contsins all Cost functions
and their respective gradients (used in
optimization methods)
'''
class CostFuncs:
    # contsructor
    def __init__(self, *args):
        pass

    # Linear Regression
    def CallLinear(self, *args):
        X = args[0]
        y = args[1]
        h = args[2]
        m = np.size(y)
        # cost function
        J = 0
```

```
        # its gradient
        dJ = 0
        return J, dJ

    # logistic Regression
    def CallLogistic(self, *args):
        X = args[0]
        y = args[1]
        h = args[2]
        m = np.size(y)
        # cost function
        J = -np.sum(y * np.log(h+1e-10) +(1-y) * np.log(1-h+1e-10)) / m
        J = np.squeeze(J)
        # its gradient
        dJ = np.dot(X.T, h - y) / m
        return J, dJ

    # Feed Forward Neural Network
    def CallNNLogistic(self, *args):
        Y = args[0]
        AL = args[1]
        modelParams = args[2]
        nLayers = args[3]
        m = args[4]
        lambd = args[5]
        #print(m)
        #print(AL)
        #AL[AL == 1] = 0.999 # if AL=0 we get an error, alternatively, I could set
            ↪  J=0 in this case
        #AL[AL==0] = AL+1e-07
        #AL = np.ravel(AL)
        #print("Y is",np.shape(Y))
        #print('AL is',np.shape(AL))
        J = -np.sum(np.multiply(Y, np.log(AL+1e-10)) +  np.multiply(1-Y, np.log(1-
            ↪ AL+1e-10)))/m
        # sum of all weights (for all layers, except input one)
        Wtot = 0
        # Computing Regularisation Term for n layer NN
        for l in range(1, nLayers, 1):
            Wtot += np.sum(np.square(modelParams['W' + str(l)]))
        Wtot = Wtot * lambd / (2*m)
        J = J + Wtot
        #L2_regularization_cost = (np.sum(np.square(W1)) + np.sum(np.square(W2)))
            ↪ *(lambd/(2*m))
        #print(np.multiply(Y, np.log(AL+1e-07)))
        J = np.squeeze(J)

        return J

    # Cost Function For Linear Regression Problems
    def CallNNMSE(self, *args):          # Mean Squared error
```

```
        Y = args[0]
        AL = args[1]
        modelParams=args[2]
        nLayers = args[3]
        m =args[4]
        lambd = args[5]

        #J =  0.5 * np.mean(np.square(AL.ravel() - Y.ravel()))
        J =  0.5 * np.sum(np.square(AL.ravel() - Y.ravel())) / m

        #J = K.sqrt(K.mean(K.square(AL - Y), axis=None))
        #print(J)
        if np.isnan(J):
            sys.exit()
            print("J", J)
        # sum of all weights (for all layers, except input one)
        Wtot = 0
        # Computing Regularisation Term for n layer NN
        for l in range(1, nLayers, 1):
            Wtot += np.sum(np.square(modelParams['W' + str(l)]))
        Wtot = Wtot * lambd / (2*m)
        #print("Wtot", Wtot)
        if np.isnan(Wtot):
            print("Wtot", Wtot)

        J = J + Wtot
        np.squeeze(J)

        #print("Wtot is", Wtot)

        #print("J is ", J)

        return  J

    def CallDMSE(self, tar, y,  lmbd = 0) :
        return (tar-y)

'''
Class which contains all testing errors (MSNE, R^2, Accuracy etc.)
'''
class ErrorFuncs:
    def __init__(self, *args):
        pass

    # MSNE
    def CallMSE(self, *args):
        z_data = args[0]
        z_model = args[1]
        n = np.size(z_model)
        return np.sum((z_data - z_model)**2) / n
```

```
    # R^2 test
    def CallR2(self, *args):
        z_data = args[0]
        z_model = args[1]
        return 1 - np.sum((z_data - z_model)**2) / np.sum((z_data - np.mean(z_data
            ↪ ))**2)


'''
Class which holds all Normal equations,
i.e. simple OLS, Ridge and LASSO used in
project 1
'''
class NormalFuncs:
    # constructor
    def __init__(self, *args):
        pass

    '''
    Generating polynomials for given number of variables for a given degree
    using Newton's Binomial formula, and when returning the design matrix,
    computed from the list of all variables
    '''
    def ConstructDesignMatrix(self, *args):
        # the degree of polynomial to be generated
        poly_degree = args[2]
        # getting inputs
        #x_vals = self.x_vals
        x_symb = args[0]
        x_vals = args[1]
        # using itertools for generating all possible combinations
        # of multiplications between our variables and 1, i.e.:
        # x_0*x_1*1, x_0*x_0*x_1*1 etc. => will get polynomial
        # coefficients
        variables = list(x_symb.copy())
        variables.append(1)
        terms = [sp.Mul(*i) for i in it.combinations_with_replacement(variables,
            ↪ poly_degree)]
        # creating desing matrix
        points = len(x_vals[0]) * len(x_vals[1])
        # creating desing matrix composed of ones
        X1 = np.ones((points, len(terms)))
        # populating design matrix with values
        for k in range(len(terms)):
            f = sp.lambdify([x_symb[0], x_symb[1]], terms[k], "numpy")
            X1[:, k] = [f(i, j) for i in x_vals[1] for j in x_vals[0]]
        # returning constructed design matrix (for 2 approaches if needed)
        return X1
    '''
    Normal Equation with lambda, i.e. it is a Ridge Regression
    (set lambda = 0 to get OLS)
```

```
    '''
    def CallNormal(self, *args):
        # getting design matrix
        X = args[0]
        # getting z values
        z = np.ravel(args[1])
        # hyper parameter
        lambda_par = args[2]
        # constructing the identity matrix
        XTX = X.T.dot(X)
        I = np.identity(len(XTX), dtype=float)
        # calculating parameters
        # if we set lambda =0, we get usual OLS,
        # but we need to account for singularity,
        # so are using SVD
        if (lambda_par == 0):
            invA = self.CallSVD(X)
        else:
            invA = np.linalg.inv(XTX + lambda_par * I)
        beta = invA.dot(X.T).dot(z)
        # and making predictions
        ztilde = X @ beta

        # calculating beta confidence
        #confidence = args[3]  # 1.96
        # calculating variance
        #sigma = args[4]#np.var(z)  # args[4] #1
        #SE = sigma * np.sqrt(np.diag(invA)) * confidence
        #beta_min = beta - SE
        #beta_max = beta + SE

        return ztilde#, beta, beta_min, beta_max

    '''
    Singular Value Decomposition for Linear Regression
    '''
    def CallSVD(self, *args):
        # getting matrix
        X = args[0]
        # Applying SVD
        A = np.transpose(X) @ X
        U, s, VT = np.linalg.svd(A)
        D = np.zeros((len(U), len(VT)))
        for i in range(0, len(VT)):
            D[i, i] = s[i]
        UT = np.transpose(U)
        V = np.transpose(VT)
        invD = np.linalg.inv(D)
        invA = np.matmul(V, np.matmul(invD, UT))

        return invA
```

```
'''
Class to generate Data,
for now only contains
Franke function
'''
class DataFuncs:
    # constructor
    def __init__(self, *args):
        pass

    # Franke Function to generate Data Set
    def CallFranke(self, *args):
        x = args[0]
        y = args[1]
        term1 = 0.75 * np.exp(-(0.25 * (9 * x - 2) ** 2) - 0.25 * ((9 * y - 2) **
            ↪ 2))
        term2 = 0.75 * np.exp(-((9 * x + 1) ** 2) / 49.0 - 0.1 * (9 * y + 1))
        term3 = 0.5 * np.exp(-(9 * x - 7) ** 2 / 4.0 - 0.25 * ((9 * y - 3) ** 2))
        term4 = -0.2 * np.exp(-(9 * x - 4) ** 2 - (9 * y - 7) ** 2)
        return term1 + term2 + term3 + term4

    # Beale's function
    def CallBeale(self, *args):
        x = args[0]
        y = args[1]
        return (1.5 - x + x*y)**2 + (2.25 - x + x*y**2)**2 + (2.625 - x + x*y**3)
            ↪ **2

    # Paraboloid
    def CallParaboloid(self, *args):
        x = args[0]
        y = args[1]
        return (x**2 - y**2)

    # Sine
    def CallSinus(self, *args):
        x = args[0]
        y = args[1]
        return np.sin(x*y)

'''
Class which contains ll plotting functions
'''
class PlotFuncs:
    # constructorf
    def __init__(self, *args):
        pass

    # Plotting Surface Just to see that we Succeeded
    def PlotSurface(self, *args):
```

```
        # passing coordinates
        x = args[0]
        y = args[1]
        # takes an array of z values
        zarray = args[2]
        # output dir
        output_dir = args[3]
        # filename
        filename = args[4]
        print(filename)
        # Turning interactive mode on
        #plt.ion()
        fig = plt.figure(figsize=(10, 3))
        axes = [fig.add_subplot(1, 3, i, projection='3d') for i in range(1, len(
            ↪ zarray) + 1)]
        #axes[0].view_init(5,50)
        #axes[1].view_init(5,50)
        #axes[2].view_init(5,50)
        surf = [axes[i].plot_surface(x, y, zarray[i], alpha = 0.5,
                                     cmap = 'brg_r', label="Franke␣function",
                                         ↪ linewidth = 0, antialiased = False)
                                         ↪ for i in range(len(zarray))]
        # saving figure with corresponding filename
        #fig.savefig(output_dir + filename)
        # close the figure window
        #plt.close(fig)
        plt.show()
        # turning the interactive mode off
        #plt.ioff()
```