EEE 485/585-01 Statistical Learning and Data Analytics
Phase 3 Report

Syed Hasan Raza – 21503652
Burak Mandira - 21301474

Project Description

After comprehensive research on several ideas and datasets, we proceed with "Phishing Website Detection" as our term project for EEE 485/585. In this project, we try to predict whether a certain website is spoofing/phishing or not. A phishing website is a certain site which tries to steal user's information without his/her consent by deception. This website tricks the user into believing that it indeed is a legitimate website. The stolen information is generally account identity and password which is supposed to be confidential. It can be related to your email account, bank account or other social media accounts etc. So, this is indeed a serious problem in the field of cyber security which needs to be addressed. Since it is not obvious for users to differentiate between a real and phishing website by themselves, we implement an intelligent system based on machine learning algorithms, which can automatically detect/predict whether a certain website is phishing or not.

In technical terms, the problem we try to solve is essentially a binary classification problem i.e. whether a website is phishing (1) or not (0). We use logistic regression as one of the methods since the problem is a binary classification. Apart from this, we also try to solve it using k-nearest neighbours (kNN) and feedforward neural network as our second and third algorithms.

Dataset

During our research, one of the challenges we faced was finding a comprehensive and reliable training dataset. In fact, this challenge is faced by every researcher in this field as there is no solid agreement in literature on the definitive features that characterize phishing webpages. However, we finally found Phishing Websites Data Set [1, 2] which covers important features proved to be sound and effective in the prediction of phishing websites. The dataset consists of 11055 instances/samples with a total of 30 categorical features/attributes for each instance. We analyzed the dataset using built-in libraries and found the dataset to be almost balanced in terms of class distributions (56% phishing, 44% not phishing). The features of phishing webpages are categorized into address bar based features, domain based features, HTTPS and certification based features, HTML and JavaScript based features. See the features file in the dataset [2] for more details.

Methods

1) Logistic Regression

In Logistic Regression, we fit the data into linear regression model and then we apply a logistic function on the data to predict the target categorical dependent variable which is "phishing website" in our case. This also justifies the name of logistic regression. The logistic function we use is a sigmoid function, shown below in Figure 1 which can take any real-valued number and and maps it into a value between 0 and 1. However, the mapping is never exactly at those limits.

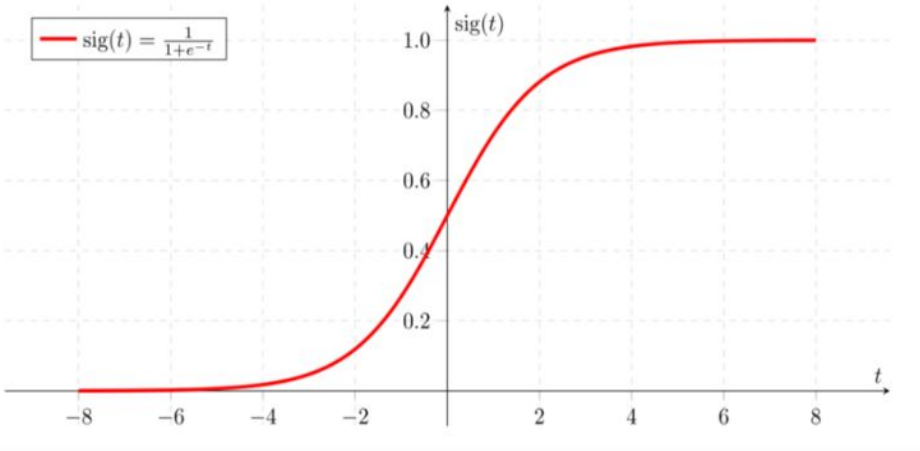

Figure 1: sigmoid function

Just like normal linear regression, we need to learn the coefficients of logistic regression algorithm, known as beta values, from our training data using maximum likelihood estimation (MLE). Good estimated beta values would be such that our model predicts a value very close to 1 for a class i.e. “website is phishing” and the value should be very close to 0 for the other class i.e. “website is not phishing”. To clearly implement this, we pass the output of the sigmoid activation function to a unit step function so that for probabilities greater than 0.5 we classify the website as phishing or else not phishing for probabilities less than 0.5. In very unlikely case of exactly 0.5 probability, we can flip a coin i.e randomly classify to one of the either class. To optimize the best beta values, we need to use a minimization algorithm which can be gradient descent algorithm or quasi-newton method. A generic description of logistic function according to our project can be depicted by diagram in Figure 2. Xs correspond to features, thetas correspond to beta values mentioned above and happy or sad is binary predicted class.

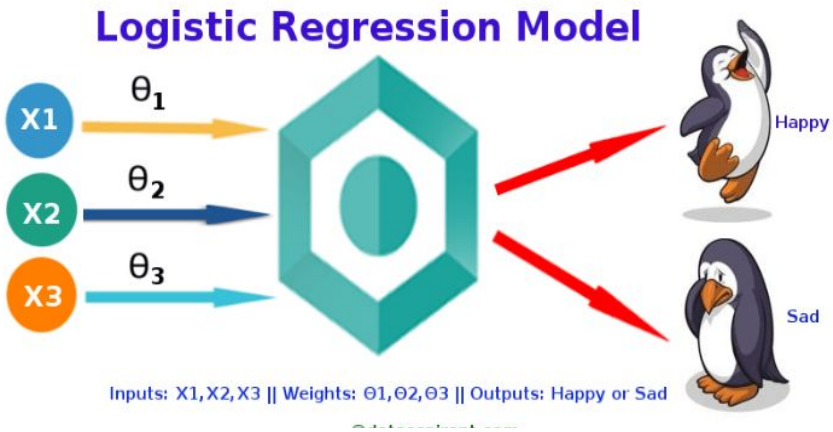

Figure 2: Logistic Regression Model

Mathematical derivation for logistic function is as follows. For each training data-point, we have a vector of features, xi , and an observed class, yi . The probability of that class was either p, if yi = 1, or 1 − p, if yi = 0 [5]. This is equivalent to bernoulli trials . With assumption of independence between the observations, we can write the conditional likelihood function as :

$$L(\beta_0, \beta) = \prod_{i=1}^{n} p(x_i)^{y_i}(1 - p(x_i)^{1-y_i}$$

where, p(x) is the sigmoid function shown in Figure 4 above with following equation.

$$p(x; b, w) = \frac{e^{\beta_0 + x \cdot \beta}}{1 + e^{\beta_0 + x \cdot \beta}} = \frac{1}{1 + e^{-(\beta_0 + x \cdot \beta)}}$$

We also define log odds as follows. we will later substitute this when deriving the likelihood/cost function

$$\log \frac{p(x)}{1 - p(x)} = \beta_0 + x \cdot \beta$$

To turn the multiplication into sums, we take the log of likelihood function and simplify.

$$\begin{aligned}
\ell(\beta_0, \beta) &= \sum_{i=1}^{n} y_i \log p(x_i) + (1 - y_i) \log 1 - p(x_i) \\
&= \sum_{i=1}^{n} \log 1 - p(x_i) + \sum_{i=1}^{n} y_i \log \frac{p(x_i)}{1 - p(x_i)} \\
&= \sum_{i=1}^{n} \log 1 - p(x_i) + \sum_{i=1}^{n} y_i (\beta_0 + x_i \cdot \beta) \\
&= \sum_{i=1}^{n} -\log 1 + e^{\beta_0 + x_i \cdot \beta} + \sum_{i=1}^{n} y_i (\beta_0 + x_i \cdot \beta)
\end{aligned}$$

Notice that we substituted the log odds equation in 2nd last step of above derivation. Generally, to find maximum likelihood estimates, we differentiate the likelihood function and equate it to zero to get the parameters. Since we have several beta, we start with one of them as follows.

$$\begin{aligned}
\frac{\partial \ell}{\partial \beta_j} &= -\sum_{i=1}^{n} \frac{1}{1 + e^{\beta_0 + x_i \cdot \beta}} e^{\beta_0 + x_i \cdot \beta} x_{ij} + \sum_{i=1}^{n} y_i x_{ij} \\
&= \sum_{i=1}^{n} \left(y_i - p(x_i; \beta_0, \beta)\right) x_{ij}
\end{aligned}$$

We will not set this to zero and solve exactly because that is a transcendental equation, and there is no closed-form solution. However, we can approximately solve it numerically using Gradient Ascent

method. Equations or derivations for that are not given here as the same method is also used in section 3 (feed-forward neural network), so it is useless to repeat at both places.

2) K- Nearest Neighbours (kNN)

In this algorithm, there is no learning for the model as most work happens when we make a prediction using raw training instances, which is why it is also known as instance based learning. For a new feature or instance x, classification can be made by searching in all training dataset for K similar or neighbouring features where output class can be calculated as the class with highest frequency from k-most similar instances [4].

In order to determine which K neighbors are similar to a new input feature vector, we will use a distance measure. Our initial choice is popular Euclidean distance, however, we can experiment with other distance measures like Manhattan distance while we tune our algorithm to get better results towards the end of project. Also, we will find optimized values of k by tuning the algorithm.

Although we just have 30 number of features in our dataset, we should still look out for"Curse of dimensionality" for our k-NN algorithm in case of poor results. This means that k-NN works fine with small number of features but not for very large datasets. In that case, we can use dimensionality reduction techniques like Principal Component Analysis (PCA) if needed.

3) Feed-Forward Neural Network (FFNN)

In FFNN, we fit the data into a non-linear model and then by using this model, we predict the target variables, e.g. whether a website is phishing or not. In FFNN, from neuron in the previous layer to each neuron in the next layer, there are connections (weights). So each layer is indeed a dense layer in this architecture. By using the power of backpropagation, the network will modify these weights and biases in order to perfectly classify the inputs into correct categories. A general model of a FFNN is depicted in Figure 3 below.

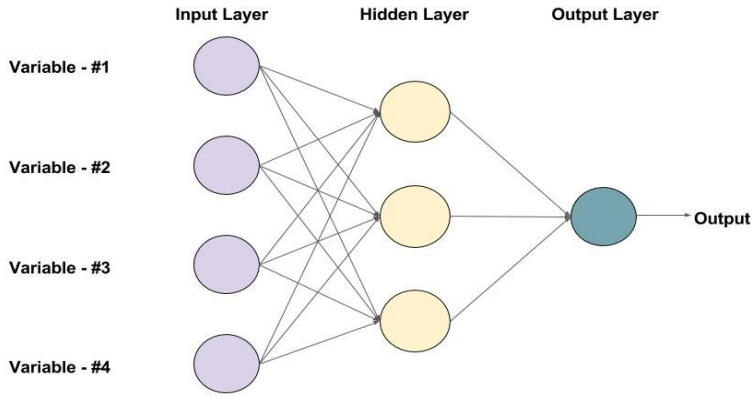

Figure 3: General model of a FFNN

Hyperparameters such as the number of layers and number of neurons in each layer, learning rate, batch size, etc. are determined via the cross-validation experiments in the Cross-Validation Experiments section. For your convenience, our best performing FFNN on validation set is found with the following hyperparameters (see the related section for details): one hidden layer of size 50, batch size is 32, learning rate is 0.001, hidden layer activation function is tanh. Since the problem is binary classification, softmax

activation function in the classification layer is used. We used cross-entropy loss as we think that it fits best for the classification task.

The output of the n+1'th layer can be calculated as:

$$O_{n+1} = \phi(O_n * W_{n\,;\,n+1})$$

where $W_{n\,;\,n+1}$ is the weight matrix between n'th and n+1'th layers, O_n is the output of the n'th layer, ϕ is the activation function, it's tanh in the hidden layer, softmax in the classification layer. Training starts with forward pass where we calculate the outputs of all layers including the classification layer. Then, we calculate the gradients for each weight matrices to be used in the gradient descent algorithm. We can calculate them using the chain rule:

$$O_1 = \phi(X_{11055x30} * W_{30x50})$$

$$\widehat{y} = \varphi(O_1 * W_{50x2})$$

$$D_2 = (\widehat{y} - y)$$

$$D_1 = (D_2 * W^T{}_{30x50}) \,.\, (1 - tanh^2(O_1))$$

where ϕ denotes tanh and φ denotes softmax. In the code, we initialize the weight matrices with uniform distribution in (-0.1, 0.1). Biases are initialized from a normal distribution with zero mean unit variance.

Validation Method

We use k-fold cross validation in order to separate the dataset into training, validation, and testings sets and to fine-tune the algorithms above using the validation set, specially feed forward neural network and logistic regression. Our choice for k is 10, i.e. 10-fold cross validation, since it's the most common used one and it provides a good trade-off between runtime and generality. The higher values k gets, the better approximates of Leave-One-Out cross-validation obtained at the end.

We analyzed the dataset and it is almost balanced in terms of class distributions (56% phishing, 44% not phishing). However, we applied data sampling in order not to take any risk while training FFNN and logistic regression models. Our choice of data sampling is oversampling since we did not want to throw away scarce data that we have. To oversample the minority class, we applied the most naive strategy is to generate new samples: randomly sampling with replacement the current available samples. After data sampling, we obtained equal number of samples for each class. Then, we separated 10% of the data as test set. We take into account the class distributions in order to preserve the percentage of samples for each class. This means equal amount of samples from each class since we applied oversampling beforehand. Test set will be used once at the end of the fine-tuning process to see the performance of the models in unseen data. The remaining data is split into training and test set using stratified 10-fold cross-validation. Stratified cross-validation takes into account the class distributions during fold separations. We did not normalize the features as all features are categorical, either binary or ternary like 0, 1 or -1, 0, 1. Since they do not quantify any real measurements and they are categorical, normalization does not make sense and even if it's applied it won't change the results. The overall performance of the models are calculated using the predictions and ground truth data that are collected from every fold.

Cross-Validation Experiments

1. **FFNN**

We conduct cross-validation experiments in order to fine-tune the hyperparameters of FFNN, i.e. number of layers, neurons, activation functions in each layer, learning rate, batch size, dropout rate, etc. First, we experiment to find the best performing hidden layer size. Since 10-fold cross-validation (CV) takes quite a lot of time even in a relatively small dataset, we applied 3-fold CV to find the number of layers with 500 training epochs in each fold. To do that, we trained one hidden layer model with various hidden layer sizes and pick the best performing one in the validation set. Table 1 displays overall accuracy as well as class-based accuracies below.

Hidden Layer Size	**Train**			**Validation**			**Test**		
	(+) Acc.	**(-) Acc.**	**Overall Acc.**	**(+) Acc.**	**(-) Acc.**	**Overall Acc.**	**(+) Acc.**	**(-) Acc.**	**Overall Acc.**
10	0.96	0.97	0.9647	0.95	0.95	0.9504	0.93	0.95	0.9369
50	0.98	0.99	0.9842	0.95	0.97	**0.9620**	0.95	0.97	0.9619
100	0.98	0.99	0.9839	0.95	0.97	0.9616	0.95	0.97	0.9610

Table 1: Hidden Layer Size Cross-Validation Experiment Results

From the results in Table 1, it can be seen that hidden layer with 50 neurons performed better in the validation with 0.9620 overall accuracy in 3-fold CV. Therefore, size of the hidden layer is chosen as 50. Although test set results are also provided in the tables, they are not used in the selection of hyperparameters. They are just given for the sake of completeness.

Hidden Layer Size	**Train**			**Validation**			**Test**		
	(+) Acc.	**(-) Acc.**	**Overall Acc.**	**(+) Acc.**	**(-) Acc.**	**Overall Acc.**	**(+) Acc.**	**(-) Acc.**	**Overall Acc.**
50	0.98	0.99	0.9842	0.95	0.97	**0.9620**	0.95	0.97	0.9619
50 / 20	0.97	0.99	0.9829	0.95	0.97	0.9598	0.96	0.96	0.9619

Table 2: Number of Hidden Layers & Size Cross-Validation Experiment Results

Then, we conduct another experiment to choose the number of hidden layers. We compared two models: the first one is with single hidden layer of size 50 and the second one is with two hidden layers of size 50 and 20 respectively. Since training FFNN with 500 epochs in 3-fold CV takes significant amount of time, we did not try more hidden layers. Table 2 shows the results above. According the results, single layer model performs better in the validation set with 0.9620 overall accuracy. Therefore, we proceed our hyperparameter selection experiments with single hidden layer of size 50.

After we finalize the number of hidden layers and size of the hidden layers, we conduct grid-search in the remaining hyperparameters e.g. batch size, learning rate, activation function in the hidden layer. Since our problem is a classification problems, we used softmax activation function in the

classification layer along with cross-entropy loss function while training FFNN. Although our implementation supports dropout mechanism, it jeopardize the performance of the model on the validation set significantly. It's mostly because we do not have large dataset and we have shallow network with only single hidden layer. Therefore, we did not apply dropout in our experiments. We tries 32, 64, 128 as batch size; 0.01 and 0.001 as learning rate; sigmoid, tanh, and relu as hidden layer activation function. Since we applied grid search techniques, that makes 18 experiments in total. We sorted the experiment results according to the overall validation accuracy and pick the best performing setup's hyper parameters as final hyperparameters.

Rank	**Batch Size**	**Learning Rate**	**Activation Function**	**Train Acc.**	**Validation Acc.**	**Test Acc.**
#1	32	0.001	tanh	0.9862	**0.9686**	0.9675
#2	64	0.01	ReLU	0.9853	0.9677	0.9651
#3	32	0.01	ReLU	0.9845	0.9674	0.9716
#4	64	0.001	tanh	0.9858	0.9670	0.9659
#5	32	0.01	sigmoid	0.9862	0.9665	0.9619

Table 2: Hyperparameter Grid-Search Cross-Validation Experiment Results

As it can be seen from Table 2, best performing hyperparameters are found to be 32, 0.001, tanh as batch size, learning rate and hidden layer activation function respectively.

2. kNN

In order to choose k value for kNN, we tried several values as k and pick the best performing one in the validation set. Table 3 shows the results of these experiments.

k	**Validation**			**Test**		
	(+) Acc.	**(-) Acc.**	**Overall Acc.**	**(+) Acc.**	**(-) Acc.**	**Overall Acc.**
3	0.95	0.95	**0.9503**	0.95	0.95	0.9513
5	0.94	0.95	0.9450	0.94	0.96	0.9505
9	0.93	0.94	0.9367	0.91	0.94	0.9294
11	0.93	0.94	0.9364	0.91	0.94	0.9261

Table 3: k value for kNN Cross-Validation Experiment Results

As Table 3 shows, best performing k value is 3 on the validation set. Therefore we choose k = 3 in our kNN implementation.

3. Logistic Regression

We also conduct cross-validation (CV) experiments in order to fine-tune the learning rate and batch size. We also compared the performance of three different update rules: batch learning, mini-batch learning, and online (stochastic) learning. We conduct our batch size experiments only for the case of mini-batch gradient descent. We tried two different learning rates: 0.01 and 0.001. For the case of mini-batch gradient descent, we tried three different batch sizes: 32, 64, and 128. For all experiments, we set epoch number to 100. At the end, we sorted the experiment results according to the overall validation accuracy and pick the best performing setup's hyperparameters as the final hyperparameters. Table 4, 5, and 6 show the best performing -according to validation set accuracy- three results (if any) for batch learning, mini-batch learning and online learning, respectively.

Rank	Learning Rate	Train Acc.	Validation Acc.	Test Acc.
#1	0.01	0.9049	**0.9043**	0.9010
#2	0.001	0.8987	0.8981	0.8945

Table 4: Hyperparameter CV Experiment Results for Batch Learning

Rank	Batch Size	Learning Rate	Train Acc.	Validation Acc.	Test Acc.
#1	32	0.01	0.9267	**0.9264**	0.9107
#2	64	0.01	0.9262	0.9255	0.9115
#3	128	0.01	0.9259	0.9252	0.9140

Table 5: Hyperparameter CV Experiment Results for Mini-batch Learning

Rank	Learning Rate	Train Acc.	Validation Acc.	Test Acc.
#1	0.001	0.9272	**0.9265**	0.9091
#2	0.01	0.9270	0.9257	0.9115

Table 6: Hyperparameter CV Experiment Results for Online Learning

As it can be seen from Table 4, 5, and 6; best performing learning rate is 0.01 for batch learning and mini-batch learning. Learning rate 0.001 performed better in terms of validation se accuracy for online learning. This result is quite expected since online learning oscillates (jumps) in the loss function space very frequently and making learning rate low decreases these oscillations (jumps), which results in a better convergence. Furthermore, best performing batch size is found to be 32 with learning rate 0.01 in case of mini-batch learning (see Table 5). To illustrate the comparison of different learning algorithms clearly, Table 7 given below combines the best performing results of each learning algorithm in a single table along with their 10-fold training time.

Learning Algorithm	Train			Validation			Test			Training Time (sec.)
	(+) Acc.	(-) Acc.	Overall Acc.	(+) Acc.	(-) Acc.	Overall Acc.	(+) Acc.	(-) Acc.	Overall Acc.	
Batch	0.92	0.89	0.9049	0.92	0.89	0.9043	0.92	0.88	0.9010	12
Mini-batch	0.93	0.92	0.9267	0.93	0.91	0.9264	0.91	0.91	0.9107	22
Online	0.93	0.92	0.9272	0.93	0.92	**0.9265**	0.91	0.91	0.9091	327

Table 7: Comparison of Best Performing Results of the Learning Algorithms

From the results in Table 7, it can be seen that online learning performed better in the validation set compared to others. However, its difference with mini-batch learning is negligibly small (1e-4). When we look at their training time, we see that it takes almost 5.5 minutes to train with online learning whereas it's only 22 seconds for mini-batch learning: mini-batch learning is almost 15 times faster. Hence, we can conclude that mini-batch learning is our winner if 0.0001 accuracy difference is not vital for our use case. Therefore, we give the result of mini-batch learning in the results and discussion section below.

Final Results & Discussion

In this section, we provide the final results of the chosen models: k-nearest neighbours (kNN), Feed-Forward Neural Network (FFNN), and logistic regression. We also comment on the results.

Model	Train			Validation			Test			Training Time (sec.)
	(+) Acc.	(-) Acc.	Overall Acc.	(+) Acc.	(-) Acc.	Overall Acc.	(+) Acc.	(-) Acc.	Overall Acc.	
kNN	NA	NA	NA	0.95	0.95	**0.9503**	0.95	0.95	0.9513	330*
FFNN	0.98	0.99	0.9862	0.97	0.97	**0.9686**	0.97	0.97	0.9675	145
Logistic Regression	0.93	0.92	0.9267	0.93	0.91	0.9264	0.91	0.91	0.9107	22

Table 8: Comparison of the Chosen Models

*: only for one fold; the rest are for 10-folds

As it can be seen from Table 8, all of our algorithms are working fine with very good accuracies: 95%, 97%, and 91% for kNN, FFNN, and logistic regression, respectively. This is the award of meticulous fine-tuning work. We can see that our winner is FFNN in terms of validation set accuracy (it's also the winner for the test case accuracy, luckily). It beats kNN and logistic regression. This results is not surprising since logistic regression is a subset of neural networks which are capable of drawing more complex boundaries between the classes compared to logistic regression models. Therefore, anything that can be done with logistic regression can also be done with neural networks with no worse performance.

When it comes to the training time comparison, kNN performs the worst as expected since it's quite computationally expensive to check neighbours and calculate distances from the query sample to training samples. Note that training time measures the completion time of a single fold for kNN differently than others in which completion time of 10-folds are measured. This is because normally there's no training phase for kNN, we thought that it would be unfair to measure it's completion time of 10-folds which is around 55 minutes. As for FFNN and logistic regression, FFNN is 6.5 times slower than logistic regression; however, it performs 4% - 6% (according to validation and test sets, respectively) better compared to logistic regression. Also, during testing there will not be such difference between FFNN and logistic regression since all the weights are learned during training and testing will take only a few seconds for both methods. Considering this and the performance enhancement of FFNN that cannot be denied, FFNN is the winner of all proposed algorithms.

References

[1] D. Dua and E. Karra Taniskidou, "UCI Machine Learning Repository", *University of California, School of Information and Computer Science*, 2017. [Online]. Available: http://archive.ics.uci.edu/ml. [Accessed: 19- Feb- 2019].

[2] R. Mohammad, L. McCluskey and F. Thabtah, "Phishing Websites Data Set", *UCI Machine Learning Repository*, 2015. [Online]. Available: https://archive.ics.uci.edu/ml/datasets/Phishing+Websites. [Accessed: 19- Feb- 2019].

[3] https://towardsdatascience.com/logistic-regression-detailed-overview-46c4da4303bc

[4] https://machinelearningmastery.com/k-nearest-neighbors-for-machine-learning/

[5] https://www.stat.cmu.edu/~cshalizi/uADA/12/lectures/ch12.pdf

Appendix 1: FFNN Implementation

```
#!/usr/bin/env python
# coding: utf-8

# In[1]:


import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import sys

from sklearn.model_selection import StratifiedKFold
from sklearn.utils import shuffle
from sklearn.metrics import classification_report, accuracy_score, precision_recall_fscore_support

# from sklearn.decomposition import PCA
from sklearn.preprocessing import minmax_scale

from imblearn.over_sampling import RandomOverSampler
from imblearn.under_sampling import RandomUnderSampler


# # Dataset class

# In[2]:


class Dataset:
    def __init__(self, datapath):
        self.datapath = datapath
        self.data = []
        self.labels = []

    def read_data(self):
        df = pd.read_csv(self.datapath, header=None)
        num_features = df.shape[1] - 1
        y = df.iloc[:, -1].values
        y[y==-1] = 0
        self.labels = y
        self.data = df.iloc[:, :-1].values

        print(self.labels.shape)
        print(self.data.shape)
        print(np.count_nonzero(self.labels==1),             np.count_nonzero(self.labels==0))

    def shuffle_data(self):
```

```
        self.data, self.labels = shuffle(self.data, self.labels, random_state=550)

    def normalize_data(self, min_range, max_range):
        self.data = minmax_scale(self.data, feature_range=(min_range, max_range))

    def prepare_nn(self, n_splits=10, normalize=True, shuffle_data=True, oversample=True,
undersample=False):
        self.read_data()
        if oversample:
            ros = RandomOverSampler(random_state=55)
            self.data, self.labels = ros.fit_resample(self.data, self.labels)
        elif undersample:
            rus = RandomUnderSampler(random_state=55)
            self.data, self.labels = rus.fit_resample(self.data, self.labels)
        if shuffle_data:
            self.shuffle_data()
        if normalize:
            self.normalize_data(0, 1)

        skf = StratifiedKFold(n_splits=n_splits, shuffle=shuffle_data, random_state=43)
        return skf


# # ANN

# In[3]:


class ANN:
        def __init__(self, input_size, output_size, hidden_size, hidden_size2=None, batch_size=None,
lr=0.001, activation='sigmoid'):
        # parameters
        np.random.seed(42)
        self.input_size = input_size
        self.output_size = output_size
        self.hidden_size = hidden_size
        self.hidden_size2 = hidden_size2
        self.lr = lr
        self.batch_size = batch_size
        self.activation = activation

        # weight
        self.w1 = np.random.uniform(low=-0.1, high=0.1, size=(self.input_size, self.hidden_size))
            self.w2 = np.random.uniform(low=-0.1, high=0.1, size=(self.hidden_size, self.output_size))  #
weight output
        if hidden_size2:
            self.w2 = np.random.uniform(low=-0.1, high=0.1, size=(self.hidden_size, self.hidden_size2))
```

```
            self.w3 = np.random.uniform(low=-0.1, high=0.1, size=(self.hidden_size2, self.output_size))  # weight output

        # bias
        self.b1 = np.random.rand() # from a uniform distribution over [0, 1)
        self.b2 = np.random.rand()
        self.b3 = np.random.rand()

        #history
        self.accuracy = []
        self.loss = []
        self.w1s = []
        self.w2s = []
        self.b1s = []
        self.b2s = []
        self.w3s = []
        self.b3s = []

    def to_onehot(self, y):
        targets = np.array(y).reshape(-1)
        n_classes = np.unique(y).size
        return np.eye(n_classes)[targets]

    def softmax(self, x):
        exps = np.exp(x - x.max())  # more stable softmax
        return exps / np.sum(exps, axis=1, keepdims=True)

    def softmax_derivative(self, predictions, y_onehot):
        return predictions - y_onehot

    def sigmoid(self, x):
        return 1 / (1 + np.exp(-x))

    def tanh(self, x):
        return np.tanh(x)

    def relu(self, x):
        return np.maximum(0, x)

    def relu_derivative(self, x):
        x[x <= 0] = 0
        x[x > 0] = 1
        return x

    def get_derivative(self, z):
        if self.activation == 'sigmoid':
            return self.sigmoid(z) * (1 - self.sigmoid(z))
        elif self.activation == 'tanh':
```

```
            return 1 - np.power(self.tanh(z), 2)
        elif self.activation == 'relu':
            return self.relu_derivative(z)
        else: # no softmax in the hidden layers
            print(self.activation, "is not supported in the hidden layer!")
            sys.exit(-1)

    def get_activation(self, z):
        if self.activation == 'sigmoid':
            return self.sigmoid(z)
        elif self.activation == 'tanh':
            return self.tanh(z)
        elif self.activation == 'relu':
            return self.relu(z)
        else: # no softmax in the hidden layers
            print(self.activation, "is not supported in the hidden layer!")
            sys.exit(-1)

    def cross_entropy(self, Y, Y_hat):
        m = Y.shape[0]
        x = Y_hat[range(m), Y]
        correct_logprobs = -np.log(x)
        data_loss = np.sum(correct_logprobs)
        return 1./m * data_loss

    def calc_accuracy(self, y, predictions):
        preds = np.argmax(predictions, axis=1)
        p, r, f1, sup = precision_recall_fscore_support(y, preds)
        total_acc = accuracy_score(y, preds)
        class_based_accuracies = r

        return total_acc, class_based_accuracies

    def print_accuracy(self, total_acc, class_based_accuracies):
        num_of_classes = class_based_accuracies.shape[0]
        print("Total accuracy is {0:.2f}".format(total_acc))
        for i in range(num_of_classes):
            print("Class {} Accuracy: {:.2f}".format(2*i - 1, class_based_accuracies[i]))

    def dropout_forward(self, A, p):
        """
            randomly drop out/shut down neurons of activation layer
            with probability of p
        :param A: Activation Layer
        :param p: dropout keep probability
        :return: Activation layer after dropping neurons and dropout mask
        """
        d = np.random.rand(A.shape[0], A.shape[1])
```

```
        d = d < p
        A = np.multiply(A, d)  # shut down some neurons of activation layer
#         A /= p
        return A, d

    def dropout_backward(self, dA, p, d):
        """
            shut down the same neurons as during the forward propagation
        :param dA: derivative of activation layer
        :param p: dropout factor
        :param d: dropout mask
        :return:
        """
        dA = np.multiply(d, dA)
        dA /= p
        return dA

    def forward_propagation(self, data, dropout_keep=1, is_test=False):
        z1 = data.dot(self.w1) + self.b1
        a1 = self.get_activation(z1)
        if not is_test:
            a1, d = self.dropout_forward(a1, dropout_keep)
        z2 = a1.dot(self.w2) + self.b2
        if self.hidden_size2:
            a2 = self.get_activation(z2)
            last_layer = a2.dot(self.w3) + self.b3
        else:
            last_layer = z2
        predictions = self.softmax(last_layer)
        if is_test:
            return predictions
        else:
            return predictions, z1, z2, d

    def backward_propagation(self, data, label_matrix, predictions, z1, z2, d_mask, dropout_keep=1):
#         predictions, z1, z2, d_mask = self.forward_propagation(data, dropout_keep)
        if self.hidden_size2:
            a2 = self.get_activation(z2)
            dZ3 = self.softmax_derivative(predictions, label_matrix)
            dW3 = a2.T.dot(dZ3)
            dB3 = np.sum(dZ3, axis=0, keepdims=True)
            dA2 = dZ3.dot(self.w3.T)
            activation_der = self.get_derivative(z2)
            dZ2 = dA2 * activation_der
            self.w3 -= self.lr * dW3  # update weights
            self.b3 -= self.lr * dB3  # update bias
        else:
            dZ2 = self.softmax_derivative(predictions, label_matrix)
```

```
        a1 = self.get_activation(z1)
        dW2 = a1.T.dot(dZ2)
        dB2 = np.sum(dZ2, axis=0, keepdims=True)
        dA1 = dZ2.dot(self.w2.T)
        dA1 = self.dropout_backward(dA1, dropout_keep, d_mask)
        activation_der = self.get_derivative(z1)
        dZ1 = dA1 * activation_der  # sigmoid derivative
        dW1 = np.dot(data.T, dZ1)
        dB1 = np.sum(dZ1, axis=0)

        # update weights and bias
        self.w2 -= self.lr * dW2
        self.b2 -= self.lr * dB2
        self.w1 -= self.lr * dW1
        self.b1 -= self.lr * dB1

#       return predictions

    def train(self, X, y, epochs, dropout_keep=1):
        y_onehot = self.to_onehot(y)
        for epoch in range(epochs):
            if self.batch_size:
                for i in range(int(len(X) / self.batch_size)):
                    lo = i * self.batch_size
                    hi = (i + 1) * self.batch_size
                    if (hi > len(X)):
                        break
                    batch_data = X[lo:hi]
                    batch_label_matrix = y_onehot[lo:hi]
                    predictions, z1, z2, d_mask = self.forward_propagation(batch_data, dropout_keep)
                                        self.backward_propagation(batch_data,  batch_label_matrix,  predictions,
z1, z2, d_mask, dropout_keep)
            else:
                predictions, z1, z2, d_mask = self.forward_propagation(X, dropout_keep)
                    self.backward_propagation(X, y_onehot, predictions,                                    z1, z2,
d_mask, dropout_keep)

            # one more forward with the updated weights
            predictions, _, _, _ = self.forward_propagation(X, dropout_keep)
            loss = self.cross_entropy(y, predictions)
            total_acc, class_based_accuracies = self.calc_accuracy(y, predictions)

            self.loss.append(loss)
            self.accuracy.append(total_acc)
            self.w1s.append(self.w1)
            self.w2s.append(self.w2)
            self.b1s.append(self.b1)
            self.b2s.append(self.b2)
```

```
            if self.hidden_size2:
                self.w3s.append(self.w3)
                self.b3s.append(self.b3)

            if epoch % 100 == 0:
                print("Epoch #{} with loss {}".format(epoch, loss))
                self.print_accuracy(total_acc, class_based_accuracies)
        print("Epoch #{} with loss {}".format(epoch, loss))
        self.print_accuracy(total_acc, class_based_accuracies)

        update_stats('train', y, predictions)
        plt.plot(self.accuracy)
        plt.ylabel('accuracy')
        plt.show()

    def test(self, X, y, dropout_keep=1, isTest=False):
        index_min = np.argmax(np.array(self.accuracy))  # pick the best model with highest accuracy
        if self.hidden_size2:
            self.w3 = self.w3s[index_min] * dropout_keep
            self.b3 = self.b3s[index_min]
        self.w1 = self.w1s[index_min] * dropout_keep
        self.b1 = self.b1s[index_min]
        self.w2 = self.w2s[index_min] * dropout_keep
        self.b2 = self.b2s[index_min]

        predictions = self.forward_propagation(X, is_test=True)

        loss = self.cross_entropy(y, predictions)
        print("Testing with the highest accuracy model:\nLoss of {}".format(loss))
        total_acc, class_based_accuracies = self.calc_accuracy(y, predictions)
        self.print_accuracy(total_acc, class_based_accuracies)

        if not isTest:
            update_stats('valid', y, predictions)
        else:
            return update_stats('test', y, predictions)


# # Data structures for overall results

# In[4]:


cumm_train_pred = np.array([])
cumm_train_y = np.array([])
cumm_valid_pred = np.array([])
cumm_valid_y = np.array([])
```

In[5]:

```
def update_stats(dataset, y, predictions):
    global cumm_train_pred, cumm_train_y
    global cumm_valid_pred, cumm_valid_y

    preds = np.argmax(predictions, axis=1)
    if dataset == 'train':
        cumm_train_y = np.hstack( (cumm_train_y, y))
        cumm_train_pred = np.hstack( (cumm_train_pred, preds))
    elif dataset == 'valid':
        cumm_valid_y = np.hstack( (cumm_valid_y, y))
        cumm_valid_pred = np.hstack( (cumm_valid_pred, preds))
    else: # test time
        print('==================')
        print('Test Report:')
        print(classification_report(y, preds, target_names=['Class -1', 'Class 1']))
        print('Test Accuracy:', accuracy_score(y, preds))

        return preds
```

In[6]:

```
def display_report(X_test, y_test, ann, d):
    global test_accuracies, test_models

    print('\n==================')
    print('Overall Train Report:')
    print(classification_report(cumm_train_y, cumm_train_pred, target_names=['Class -1', 'Class 1']))
    print('Overall Train Accuracy:', accuracy_score(cumm_train_y, cumm_train_pred))
    print('==================')
    print('Overall Valid Report:')
    print(classification_report(cumm_valid_y, cumm_valid_pred, target_names=['Class -1', 'Class 1']))
    print('Overall Valid Accuracy:', accuracy_score(cumm_valid_y, cumm_valid_pred))
    print('==================\n')

    print("Test Size:",len(X_test))
    a = np.count_nonzero(y_test)
    print("Class Ratios(+1/-1):\t{}/{}".format(a, len(y_test)-a))

    return ann.test(X_test, y_test, dropout_keep=d, isTest=True)

#     test_accuracies = np.hstack( (test_accuracies, test_acc))
#     test_models = np.hstack( (test_models, ann_instance))
```

Wrapper training class

In[7]:

```
# activation is one of relu, tanh, sigmoid (no softmax support for the hidden layers)
def train(epochs=1000, k=10, hidden_size=10, hidden_size2=None, activation='relu', batch_size=128, lr=0.001, d=1):

    # read & split data
    dataset = Dataset('data.csv')
                    skf = dataset.prepare_nn(n_splits=k, normalize=False, shuffle_data=True, oversample=True, undersample=False)

    # separate test set (%10) using one of the folds
    X_test = np.array([])
    y_test = np.array([])
    test_skf = StratifiedKFold(n_splits=10, shuffle=True, random_state=43)
    for train, test in test_skf.split(dataset.data, dataset.labels):
        dataset.data, X_test = dataset.data[train], dataset.data[test]
        dataset.labels, y_test = dataset.labels[train], dataset.labels[test]
        break

    input_size = dataset.data.shape[1]
    output_size = 2

    i = 1
    for train_index, test_index in skf.split(dataset.data, dataset.labels):
        X_train, X_valid = dataset.data[train_index], dataset.data[test_index]
        y_train, y_valid = dataset.labels[train_index], dataset.labels[test_index]

        print("\nFold {}:\ttrain_size:{} valid_size:{}".format(i, len(X_train), len(X_valid)))
        a = np.count_nonzero(y_train)
        b = np.count_nonzero(y_valid)
        print("Class Ratios(+1/-1):\ttrain:{}/{} valid:{}/{}\n".format(a, len(y_train)-a, b, len(y_valid)-b))

            ann = ANN(input_size, output_size, hidden_size=hidden_size, hidden_size2=hidden_size2, activation=activation, batch_size=batch_size, lr=lr)
        ann.train(X_train, y_train, epochs, dropout_keep=d)
        ann.test(X_valid, y_valid, dropout_keep=d)
        i += 1

        print('----------------------')

    # print overall classification report (test score is based on the last trained ann)
    test_preds = display_report(X_test, y_test, ann, d)
```

```
    return ann, y_test, test_preds


# In[8]:


# activation is one of relu, tanh, sigmoid (no softmax support for the hidden layers)
# _ = train(epochs=500, k=10, hidden_size=50, hidden_size2=None, activation='relu', batch_size=128, lr=0.001, d=1)


# # Cross-validation

# In[9]:


# cross-validate the hyperparameters using validation set
batch_sizes = [32, 64, 128]
learning_rates = [0.01, 0.001]
activations = ['sigmoid', 'tanh', 'relu']

val_accuracies = np.array([])
val_models = np.array([])
test_accuracies = np.array([])

for bs in batch_sizes:
    for lr in learning_rates:
        for act in activations:
            cumm_train_pred = np.array([])
            cumm_train_y = np.array([])
            cumm_valid_pred = np.array([])
            cumm_valid_y = np.array([])

            print('\n\n============================')
            print('Batch Size:{}, Learning Rate:{}, Activation:{}'.format(bs, lr, act))
            print('============================\n')

                    ann, y_test, test_preds = train(epochs=500, k=10, hidden_size=50, hidden_size2=None, activation=act, batch_size=bs, lr=lr, d=1)

            val_accuracies = np.hstack( (val_accuracies, accuracy_score(cumm_valid_y, cumm_valid_pred)))
            val_models = np.hstack( (val_models, ann))
            test_accuracies = np.hstack( (test_accuracies, accuracy_score(y_test, test_preds)))

print('\n\n---------RESULTS---------')
t = zip(val_accuracies, val_models, test_accuracies)
sorted_models = sorted(t, key=lambda tup: tup[0], reverse=True)
```

```
for m in sorted_models:
    print('Best validation model accuracy:', m[0])
            print('Batch Size:{}, Learning Rate:{}, Activation:{}'.format(m[1].batch_size, m[1].lr,
m[1].activation))
    print('Test accuracy of the best model:', m[2])
    break


print('\nOther models:')
i = 0
for m in sorted_models:
    if i == 0:
        i = 1
        continue
    print('Validation model accuracy:', m[0])
            print('Batch Size:{}, Learning Rate:{}, Activation:{}'.format(m[1].batch_size, m[1].lr,
m[1].activation))
    print('Test accuracy of the model:', m[2])
    print()


# In[10]:


import datetime
datetime.datetime.now()
```

Appendix 2: kNN Implementation

```
def euclideanDistance(x1, x2):
    return (math.sqrt(sum((x1 - x2)**2)))

#This function returns the indexes of rows in trainingset for k Nearest neighbours of  test instance
def getNeighbors(xTrainSet, xTestInstance, k):
    dists = np.zeros(len(xTrainSet))
    n = np.zeros(k)
    for j in range(len(xTrainSet)):
        d = euclideanDistance(xTrainSet[j], xTestInstance)
        dists[j] = d
    sortedIndexes = np.argsort(dists)
    for i in range(k):
        n[i] = sortedIndexes[i]
    return n

#This function predicts the class of test instance according to majority rule voting by k nearest neighbours
class label
def predict(neighbors, y_train):
```

```
    count= 0
    for z in range(len(neighbors)):
        if(y_train[np.int(neighbors[z])] == 1):
            count += 1
        else:
            count -= 1
    if(count > 0):
        return 1
    elif(count < 0):
        return 0


# read & split data
dataset = Dataset('data.csv')
skf = dataset.prepare_nn(n_splits=10, normalize=False, shuffle_data=True,\
                         oversample=True, undersample=False)

# separate test set (%10) using one of the folds
X_test = np.array([])
y_test = np.array([])
test_skf = StratifiedKFold(n_splits=10, shuffle=True, random_state=43)
for train, test in test_skf.split(dataset.data, dataset.labels):
    dataset.data, X_test = dataset.data[train], dataset.data[test]
    dataset.labels, y_test = dataset.labels[train], dataset.labels[test]
    break

start = time.time()
X_train = dataset.data
y_train = dataset.labels

k = 3
fold = 1
cumm_valid_pred = np.array([])
cumm_valid_y = np.array([])

for train_index, test_index in skf.split(dataset.data, dataset.labels):

    X_train, X_valid = dataset.data[train_index], dataset.data[test_index]
    y_train, y_valid = dataset.labels[train_index], dataset.labels[test_index]
    val_predictions = np.zeros(len(X_valid))

    print("\nFold {}:\ttrain_size:{} valid_size:{}".format(fold, len(X_train), len(X_valid)))
    a = np.count_nonzero(y_train)
    b = np.count_nonzero(y_valid)
    print("Class Ratios(+1/-1):\ttrain:{}/{} valid:{}/{}\n".format(a, len(y_train)-a, b, len(y_valid)-b))

    for i in range(len(X_valid)):
        neighbors = getNeighbors(X_train, X_valid[i], k)
```

```
        val_predictions[i] = predict(neighbors, y_train)

    print("Validation Size:",len(X_valid))
    a = np.count_nonzero(y_valid)
    print("Class Ratios(+1/-1):\t{}/{}".format(a, len(y_valid)-a))

    p, r, f1, sup = precision_recall_fscore_support(y_valid, val_predictions)
    total_acc = accuracy_score(y_valid, val_predictions)

    cumm_valid_pred = np.hstack((cumm_valid_pred, val_predictions))
    cumm_valid_y = np.hstack((cumm_valid_y, y_valid))

    print("Total accuracy is {0:.4f}".format(total_acc))
    for i in range(2):
        print("Class {} Accuracy: {:.2f}".format(2*i - 1, r[i]))

    fold += 1

print('\n================')
print('Overall Valid Report:')
print(classification_report(cumm_valid_y, cumm_valid_pred, target_names=['Class -1', 'Class 1']))
print('Overall Valid Accuracy:', accuracy_score(cumm_valid_y, cumm_valid_pred))
print('================\n')


# test time
test_predictions = np.zeros(len(X_test))
for i in range(len(X_test)):
    neighbors = getNeighbors(X_train, X_test[i], k)
    test_predictions[i] = predict(neighbors, y_train)

print("Test Size:",len(X_test))
a = np.count_nonzero(y_valid)
print("Class Ratios(+1/-1):\t{}/{}".format(a, len(y_test)-a))

p, r, f1, sup = precision_recall_fscore_support(y_test, test_predictions)
total_acc = accuracy_score(y_test, test_predictions)
print("Test accuracy is {0:.4f}".format(total_acc))
for i in range(2):
    print("Class {} Accuracy: {:.2f}".format(2*i - 1, r[i]))
```

Appendix 3: Logistic Regression Implementation

```
import pandas as pd

#import matplotlib.pyplot as plot

import numpy as np
```

```
import time
from sklearn.model_selection import StratifiedKFold
from sklearn.utils import shuffle
from sklearn.metrics import classification_report, accuracy_score, precision_recall_fscore_support
# from sklearn.decomposition import PCA
from sklearn.preprocessing import minmax_scale
from imblearn.over_sampling import RandomOverSampler
from imblearn.under_sampling import RandomUnderSampler


def calc_accuracy(y, predictions):
    y = y.squeeze()
    preds = predictions.squeeze()
    p, r, f1, sup = precision_recall_fscore_support(y, preds)
    total_acc = accuracy_score(y, preds)
    class_based_accuracies = r
    return total_acc, class_based_accuracies


def print_accuracy(total_acc, class_based_accuracies):
    num_of_classes = class_based_accuracies.shape[0]
    print("Total accuracy is {0:.4f}".format(total_acc))
    for i in range(num_of_classes):
        print("Class {} Accuracy: {:.4f}".format(2*i - 1, class_based_accuracies[i]))
class Dataset:
    def __init__(self, datapath):
        self.datapath = datapath
        self.data = []
        self.labels = []
```

```
def read_data(self):
    df = pd.read_csv(self.datapath, header=None)
    num_features = df.shape[1] - 1
    y = df.iloc[:, -1].values
    y[y==-1] = 0
    self.labels = y
    self.data = df.iloc[:, :-1].values

    print(self.labels.shape)
    print(self.data.shape)
    print(np.count_nonzero(self.labels==1),\
          np.count_nonzero(self.labels==0))

def shuffle_data(self):
    self.data, self.labels = shuffle(self.data, self.labels, random_state=550)

def normalize_data(self, min_range, max_range):
    self.data = minmax_scale(self.data, feature_range=(min_range, max_range))

def prepare_nn(self, n_splits=10, normalize=True, shuffle_data=True,\
                              oversample=True, undersample=False):
    self.read_data()
    if oversample:
        ros = RandomOverSampler(random_state=55)
        self.data, self.labels = ros.fit_resample(self.data, self.labels)
    elif undersample:
```

```
            rus = RandomUnderSampler(random_state=55)
            self.data, self.labels = rus.fit_resample(self.data, self.labels)
        if shuffle_data:
            self.shuffle_data()
        if normalize:
            self.normalize_data(0, 1)

        skf = StratifiedKFold(n_splits=n_splits, shuffle=shuffle_data, random_state=43)
        return skf

def logistic_func(X, weights):
    z = np.dot(X,weights)
    return 1 / (1 + np.exp(-z))

def g_ascent(X, h, y, weight, rate):
    return weight + rate * (1/len(X))*np.dot(X.T, y - h)

def full_batch(x_train, y_train, weights, rate, max_iters, cumm_train_y, cumm_train_pred):
    for i in range(max_iters):
#         print('FB Iter:', i+1)
        h = logistic_func(x_train, weights)
        weights = g_ascent(x_train, h, y_train, weights, rate)

        # display train accuracy
        _, preds = predict(x_train,y_train, weights)
#         total_acc, class_based_accuracies = calc_accuracy(y_train, preds)
#         print_accuracy(total_acc, class_based_accuracies)
```

```
#       print()
    cumm_train_y['FB'] = np.hstack( (cumm_train_y['FB'], y_train.squeeze()))
    cumm_train_pred['FB'] = np.hstack( (cumm_train_pred['FB'], preds.squeeze()))
    return weights

def mini_batch(x_train, y_train, weights, rate, batch_size, max_iters, cumm_train_y, cumm_train_pred):
    for k in range(max_iters):
#       print('MB Iter:', k+1)
        for i in range(int(len(x_train)/(batch_size))):
            x_train_MB = x_train[i*(batch_size):(i+1)*(batch_size), :]
            y_train_MB = y_train[i*(batch_size):(i+1)*(batch_size), :]
            h = logistic_func(x_train_MB, weights)
            weights = weights + rate * (1/(batch_size)*np.dot(x_train_MB.T, y_train_MB - h))

        if len(x_train) % batch_size != 0:
            i = int(len(x_train)/(batch_size))
            x_train_MB = x_train[i*(batch_size):, :]
            y_train_MB = y_train[i*(batch_size):, :]
            h = logistic_func(x_train_MB, weights)
            weights = weights + rate * (1/(x_train_MB.shape[0])*np.dot(x_train_MB.T, y_train_MB - h))

         # display train accuracy
        _, preds = predict(x_train,y_train, weights)
#       total_acc, class_based_accuracies = calc_accuracy(y_train, preds)
#       print_accuracy(total_acc, class_based_accuracies)
#       print()
    cumm_train_y['MB'] = np.hstack( (cumm_train_y['MB'], y_train.squeeze()))
```

```
    cumm_train_pred['MB'] = np.hstack( (cumm_train_pred['MB'], preds.squeeze()))
    return weights

def stochastic(x_train, y_train, weights, rate, max_iters, cumm_train_y, cumm_train_pred):
    for k in range(max_iters):
#       print('ST Iter:', k+1)
        for i in range(len(x_train)):
            x_train_ST = x_train[i,:].reshape(len(x_train[0]),1)
            y_train_ST = y_train[i,:].reshape(1,1)
            z = x_train_ST.T@weights
            h = 1 / (1 + np.exp(-z))
            weights = weights + rate*x_train_ST*(y_train_ST-h)

        # display train accuracy
        _, preds = predict(x_train,y_train, weights)
#       total_acc, class_based_accuracies = calc_accuracy(y_train, preds)
#       print_accuracy(total_acc, class_based_accuracies)
#       print()
    cumm_train_y['ST'] = np.hstack( (cumm_train_y['ST'], y_train.squeeze()))
    cumm_train_pred['ST'] = np.hstack( (cumm_train_pred['ST'], preds.squeeze()))
    return weights

def predict(x_test,y_test, trained_weights):
    y_predicted = logistic_func(x_test, trained_weights)
    y_predicted[y_predicted < 0.5] = 0
    y_predicted[y_predicted >= 0.5] = 1
    true_predictions_count = sum(1*(y_predicted == y_test))
```

```
    accuracy = (true_predictions_count/x_test.shape[0])*100
    return accuracy,y_predicted

X_test = np.array([])
y_test = np.array([])

def train(k, lr, batch_size, max_iters, onlyMB=False):
    global X_test, y_test

    cumm_train_pred = {'FB': np.array([]), 'MB': np.array([]), 'ST': np.array([])}
    cumm_train_y = {'FB': np.array([]), 'MB': np.array([]), 'ST': np.array([])}
    cumm_valid_pred = {'FB': np.array([]), 'MB': np.array([]), 'ST': np.array([])}
    cumm_valid_y = {'FB': np.array([]), 'MB': np.array([]), 'ST': np.array([])}

    # read & split data
    dataset = Dataset('data.csv')
    skf = dataset.prepare_nn(n_splits=k, normalize=False, shuffle_data=True,\
                             oversample=True, undersample=False)

    # separate test set (%10) using one of the folds
    X_test = np.array([])
    y_test = np.array([])
    test_skf = StratifiedKFold(n_splits=10, shuffle=True, random_state=43)
    for train, test in test_skf.split(dataset.data, dataset.labels):
        dataset.data, X_test = dataset.data[train], dataset.data[test]
        dataset.labels, y_test = dataset.labels[train], dataset.labels[test]
        break
```

```
    y_test = y_test.reshape(-1,1)


    tot_FB_time = 0.
    tot_MB_time = 0.
    tot_ST_time = 0.


    i = 1
    for train_index, test_index in skf.split(dataset.data, dataset.labels):
        X_train, X_valid = dataset.data[train_index], dataset.data[test_index]
        y_train, y_valid = dataset.labels[train_index], dataset.labels[test_index]


        y_train, y_valid = y_train.reshape(-1,1), y_valid.reshape(-1,1)


        print("\nFold {}:\ttrain_size:{} valid_size:{}".format(i, len(X_train), len(X_valid)))
        a = np.count_nonzero(y_train)
        b = np.count_nonzero(y_valid)
        print("Class Ratios(+1/-1):\ttrain:{}/{} valid:{}/{}\n".format(a, len(y_train)-a, b, len(y_valid)-b))


        FB_train_weights = -1
        if not onlyMB:
            # Full Batch
            weights = np.zeros(len(X_valid[0])).reshape(len(X_train[0]),1)
            start = time.time()
                    FB_train_weights = full_batch(X_train, y_train, weights, lr, max_iters, cumm_train_y,
cumm_train_pred)
            end = time.time()
            total_time = end - start
            tot_FB_time += total_time
```

```
        FB_accuracy = predict(X_valid,y_valid, FB_train_weights)
        predictions = FB_accuracy[1]
        total_acc, class_based_accuracies = calc_accuracy(y_valid, predictions)
        print_accuracy(total_acc, class_based_accuracies)
                print("The accuracy for Full Batch GD is %f with training time of %f seconds." %
(FB_accuracy[0],total_time))
        cumm_valid_pred['FB'] = np.hstack( (cumm_valid_pred['FB'], predictions.squeeze()))
        cumm_valid_y['FB'] = np.hstack( (cumm_valid_y['FB'], y_valid.squeeze()))


    # Mini Batch
    weights = np.zeros(len(X_valid[0])).reshape(len(X_train[0]),1)
    start = time.time()
     MB_train_weights = mini_batch(X_train, y_train, weights, lr, batch_size, max_iters, cumm_train_y,
cumm_train_pred)
    end = time.time()
    total_time = end - start
    tot_MB_time += total_time


    MB_accuracy = predict(X_valid,y_valid, MB_train_weights)
    predictions = MB_accuracy[1]
    total_acc, class_based_accuracies = calc_accuracy(y_valid, predictions)
    print_accuracy(total_acc, class_based_accuracies)
            print("The accuracy for Mini Batch GD is %f with training time of %f seconds." %
(MB_accuracy[0],total_time))
    cumm_valid_pred['MB'] = np.hstack( (cumm_valid_pred['MB'], predictions.squeeze()))
    cumm_valid_y['MB'] = np.hstack( (cumm_valid_y['MB'], y_valid.squeeze()))


    ST_train_weights = -1
```

```
        if not onlyMB:
            # Stochastic
            weights = np.zeros(len(X_valid[0])).reshape(len(X_train[0]),1)
            start = time.time()
                    ST_train_weights = stochastic(X_train, y_train, weights, lr, max_iters, cumm_train_y, cumm_train_pred)
            end = time.time()
            total_time = end - start
            tot_ST_time += total_time

            ST_accuracy = predict(X_valid,y_valid, ST_train_weights)
            predictions = ST_accuracy[1]
            total_acc, class_based_accuracies = calc_accuracy(y_valid, predictions)
            print_accuracy(total_acc, class_based_accuracies)
                    print("The accuracy for Stochastic GD is %f with training time of %f seconds." % (ST_accuracy[0],total_time))
            cumm_valid_pred['ST'] = np.hstack( (cumm_valid_pred['ST'], predictions.squeeze()))
            cumm_valid_y['ST'] = np.hstack( (cumm_valid_y['ST'], y_valid.squeeze()))
        i += 1

    cumm_dicts = (cumm_train_pred, cumm_train_y, cumm_valid_pred, cumm_valid_y)
    trained_weights = (FB_train_weights, MB_train_weights, ST_train_weights)
    times = (tot_FB_time, tot_MB_time, tot_ST_time)

    return cumm_dicts, trained_weights, times

# test time
def print_test_results(GD, trained_weights, times):
```

```
    FB_train_weights = trained_weights[0]
    MB_train_weights = trained_weights[1]
    ST_train_weights = trained_weights[2]
    tot_FB_time = times[0]
    tot_MB_time = times[1]
    tot_ST_time = times[2]

    if GD == 'FB':
        FB_accuracy = predict(X_test,y_test, FB_train_weights)
        predictions = FB_accuracy[1]
        total_acc, class_based_accuracies = calc_accuracy(y_test, predictions)
        print_accuracy(total_acc, class_based_accuracies)
            print("Test accuracy for Full Batch GD is %f with total training time of %f seconds." %
(FB_accuracy[0],tot_FB_time))
    elif GD == 'MB':
        MB_accuracy = predict(X_test,y_test, MB_train_weights)
        predictions = MB_accuracy[1]
        total_acc, class_based_accuracies = calc_accuracy(y_test, predictions)
        print_accuracy(total_acc, class_based_accuracies)
            print("Test accuracy for Mini Batch GD is %f with total training time of %f seconds." %
(MB_accuracy[0],tot_MB_time))
    elif GD == 'ST':
        ST_accuracy = predict(X_test,y_test, ST_train_weights)
        predictions = ST_accuracy[1]
        total_acc, class_based_accuracies = calc_accuracy(y_test, predictions)
        print_accuracy(total_acc, class_based_accuracies)
            print("Test accuracy for Stochastic GD is %f with total training time of %f seconds." %
(ST_accuracy[0],tot_ST_time))
    else:
```

```
        print("Unrecognized GD option ", GD)


    return total_acc


## train once (not CV)
#cumm_dicts, trained_weights, times = train(k=10, lr=0.01, batch_size=64, max_iters=100)


#cumm_train_pred = cumm_dicts[0]
#cumm_train_y = cumm_dicts[1]
#cumm_valid_pred = cumm_dicts[2]
#cumm_valid_y = cumm_dicts[3]


## display reports
#for GD in cumm_train_pred:
#    print('\n==================')
#    print('Overall Train[{}] Report:'.format(GD))
#        print(classification_report(cumm_train_y[GD], cumm_train_pred[GD], target_names=['Class -1', 'Class 1']))
#                print('Overall Train[{}] Accuracy:'.format(GD), accuracy_score(cumm_train_y[GD], cumm_train_pred[GD]))
#    print('==================')
#    print('Overall Valid[{}] Report:'.format(GD))
#        print(classification_report(cumm_valid_y[GD], cumm_valid_pred[GD], target_names=['Class -1', 'Class 1']))
#                print('Overall Valid[{}] Accuracy:'.format(GD), accuracy_score(cumm_valid_y[GD], cumm_valid_pred[GD]))
#    print('==================\n')


#    print("Test Size:",len(X_test))
```

```
#    a = np.count_nonzero(y_test)
#    print("Class Ratios(+1/-1):\t{}/{}".format(a, len(y_test)-a))

#    print_test_results(GD, trained_weights, times)



# CV: cross-validate lr and batch_size (only for MB) using validation set
batch_sizes = [32, 64, 128]
learning_rates = [0.01, 0.001]

val_accuracies = {'FB': np.array([]), 'MB': np.array([]), 'ST': np.array([])}
val_models = {'FB': np.array([]), 'MB': np.array([]), 'ST': np.array([])}
test_accuracies = {'FB': np.array([]), 'MB': np.array([]), 'ST': np.array([])}

k = 10
max_iters = 100
for bs in batch_sizes:
    for lr in learning_rates:
        print('\n\n============================')
        print('Batch Size:{}, Learning Rate:{}'.format(bs, lr))
        print('============================\n')

        onlyMB=(bs!=batch_sizes[0])
        if not onlyMB:
                cumm_dicts, trained_weights, times = train(k=k, lr=lr, batch_size=bs, max_iters=max_iters,
onlyMB=onlyMB)

            cumm_train_pred = cumm_dicts[0]
```

```
        cumm_train_y = cumm_dicts[1]
        cumm_valid_pred = cumm_dicts[2]
        cumm_valid_y = cumm_dicts[3]

        for GD in ['FB', 'MB', 'ST']:
            print('\n==================')
            print('Overall Train[{}] Report:'.format(GD))
              print(classification_report(cumm_train_y[GD], cumm_train_pred[GD], target_names=['Class -1', 'Class 1']))
                  print('Overall Train[{}] Accuracy:'.format(GD), accuracy_score(cumm_train_y[GD], cumm_train_pred[GD]))
            print('==================')
            print('Overall Valid[{}] Report:'.format(GD))
             print(classification_report(cumm_valid_y[GD], cumm_valid_pred[GD], target_names=['Class -1', 'Class 1']))
                  print('Overall Valid[{}] Accuracy:'.format(GD), accuracy_score(cumm_valid_y[GD], cumm_valid_pred[GD]))
            print('==================\n')

              val_accuracies[GD] = np.hstack( (val_accuracies[GD], accuracy_score(cumm_valid_y[GD], cumm_valid_pred[GD])))
            val_models[GD] = np.hstack( (val_models[GD], np.array({'batch_size': bs, 'lr': lr})))
            test_accuracy = print_test_results(GD, trained_weights, times)
            test_accuracies[GD] = np.hstack( (test_accuracies[GD], test_accuracy))


    else:   # only MB
            cumm_dicts, trained_weights, times = train(k=k, lr=lr, batch_size=bs, max_iters=max_iters, onlyMB=onlyMB)
```

```
        cumm_train_pred = cumm_dicts[0]

        cumm_train_y = cumm_dicts[1]

        cumm_valid_pred = cumm_dicts[2]

        cumm_valid_y = cumm_dicts[3]


        print('\n==================')

        print('Overall Train[{}] Report:'.format('MB'))

          print(classification_report(cumm_train_y['MB'], cumm_train_pred['MB'], target_names=['Class -1', 'Class 1']))

                print('Overall Train[{}] Accuracy:'.format('MB'), accuracy_score(cumm_train_y['MB'], cumm_train_pred['MB']))

        print('==================')

        print('Overall Valid[{}] Report:'.format('MB'))

          print(classification_report(cumm_valid_y['MB'], cumm_valid_pred['MB'], target_names=['Class -1', 'Class 1']))

                print('Overall Valid[{}] Accuracy:'.format('MB'), accuracy_score(cumm_valid_y['MB'], cumm_valid_pred['MB']))

        print('==================\n')


          val_accuracies['MB'] = np.hstack( (val_accuracies['MB'], accuracy_score(cumm_valid_y['MB'], cumm_valid_pred['MB'])))

        val_models['MB'] = np.hstack( (val_models['MB'], np.array({'batch_size': bs, 'lr': lr})))

        MB_test_accuracy = print_test_results('MB', trained_weights, times)

        test_accuracies['MB'] = np.hstack( (test_accuracies['MB'], MB_test_accuracy))


print('\n\n---------RESULTS---------')

for GD in ['FB', 'MB', 'ST']:

    t = zip(val_accuracies[GD], val_models[GD], test_accuracies[GD])

    sorted_models = sorted(t, key=lambda tup: tup[0], reverse=True)
```

```
    for m in sorted_models:
        print('[{}] Best validation model accuracy:'.format(GD), m[0])
        print('[{}] Batch Size:{}, Learning Rate:{}'.format(GD, m[1]['batch_size'], m[1]['lr']))
        print('[{}] Test accuracy of the best model:'.format(GD), m[2])
        break
    print('---')


print('\nOther models:')
for GD in ['FB', 'MB', 'ST']:
    t = zip(val_accuracies[GD], val_models[GD], test_accuracies[GD])
    sorted_models = sorted(t, key=lambda tup: tup[0], reverse=True)
    i = 0
    for m in sorted_models:
        if i == 0:
            i = 1
            continue
        print('[{}] validation model accuracy:'.format(GD), m[0])
        print('[{}] Batch Size:{}, Learning Rate:{}'.format(GD, m[1]['batch_size'], m[1]['lr']))
        print('[{}] Test accuracy of the model:'.format(GD), m[2])
        print()
    print('---')
```