Mehmet Bayık
21802166
20.12.2023

EEE485 – Final Report
Alzheimer MRI Classification

Introduction

Alzheimer becoming more common disease in the world. It is a progressive disease that destroys memory and other important mental functions. Alzheimer's disease is the most common cause of dementia — a continuous decline in thinking, behavioral and social skills that disrupts a person's ability to function independently. The early signs of the disease may be forgetting recent events or conversations. As the disease progresses, a person with Alzheimer's disease will develop severe memory impairment and lose the ability to carry out everyday tasks. Alzheimer's disease is the sixth leading cause of death in the United States. [1] Those with Alzheimer's live an average of eight years after their symptoms become noticeable to others, but survival can range from four to 20 years, depending on detection phase, age, and other health conditions.

Problem Definition

Detecting Alzheimer might be hard for doctors who are at the start of their career, and in some extreme cases even experienced doctors can be indecisive about choosing the phase of patient's Alzheimer. Goal of this project is helping doctors as a secondary tool to detect Alzheimer cases with higher accuracy. To achieve this goal, three machine learning algorithms will be implemented. Since we know that detecting Alzheimer early is important to slow down disease, one of my performance metrics will be False Positive Rate for Non-Demented class since it is most problematic class to detect wrongly.

The first algorithm that I implemented is logistic regression for this classification task. Logistic regression is a basic but powerful way to classify. Although it is not the best way, it will be still viable. The second algorithm will be Support Vector Machine. Which is separating data points using hyperplanes. It may take higher epochs to converge but it will give very good result. The third algorithm will be Neural Networks. Which is actually advanced version of logistic regression using generalized linear models. I implemented these algorithms from scratch without using any ML libraries. I will be using Python for this project.

Dataset Description

In this project, Alzheimer MRI dataset used from Kaggle. [2] In this Kaggle dataset, there are total of 6400 images collected from several websites, hospitals, public repositories. The dataset includes 4 classes of 128x128 preprocessed images. These classes are: Non-Demented, Very Mild Demented, Mild Demented, Moderate Demented and there are 3200, 2240, 896, 64 images in these classes respectively. There is no csv file for this dataset. I will be using the folder structure to extract data from images to NumPy arrays. I will be using 80% of the data for training, 10% of the data for validation and 10% of the data for testing. Validation set will be used for hyperparameter tuning for same models. Test set will be used for final evaluation of the 3 models with 3 different algorithms.

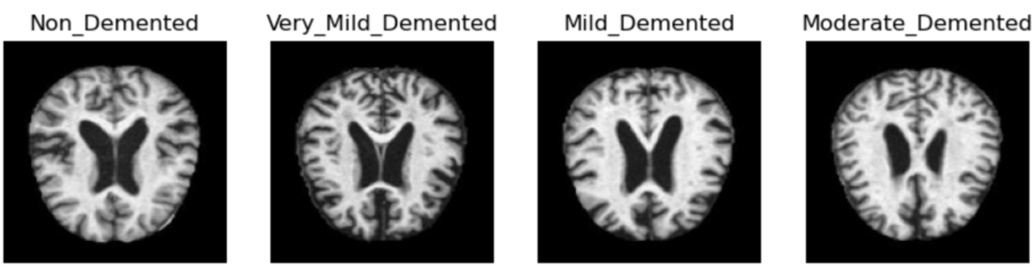

Fig.1: Sample Image from Each Class

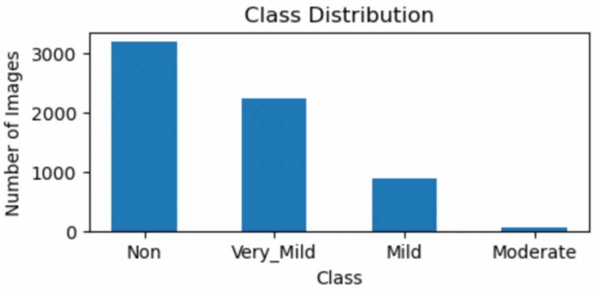

Fig.2: Class Distribution

There are some different angle - different process method images exist in Kaggle dataset. Trying to train model with different kind of data must be ambiguous. Despite my doubts, dataset performed really well with my selected algorithms. So, mixing different angle and processed images might not be wrong to do.

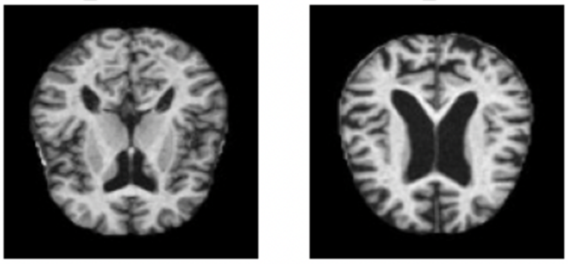

Fig.3: Two (different angle - different processed) image.

Principal Component Analysis (PCA)

In the context of our MRI image analysis, the Principal Component Analysis (PCA) algorithm proves to be an invaluable tool for handling the considerable dataset comprising 6400 images for each class. Initially, the pixel values of each image are organized into a matrix and mean-centering is applied to eliminate bias and establish a standardized reference point. The subsequent computation of the covariance matrix unveils the inter-feature relationships within the data. Through eigen decomposition, PCA identifies the principal components—eigenvectors that represent the directions of maximum variance. By selecting the top eigenvectors based on their corresponding eigenvalues, a projection matrix is formed, allowing for dimensionality reduction while retaining essential information. This reduction not only aids in managing computational complexity but also holds potential for noise reduction and insightful data visualization. Employing PCA on our extensive MRI dataset provides a means to extract meaningful features, facilitating more efficient analysis and interpretation across the multitude of images in each class.

To make things faster in trainings and also in PCA —which take around 20 minutes normally first I flattened image data to 5200x16864 matrix where each row represents one sample. Then I deleted columns with zero variance, then I got 10869 columns total. This decreased time for PC analysis. Then, since first 2879 principal components explain more than 99% of the variance in data, I choose my k value as 2789 then projected to matrix for first 2780 PCs using PCA algorithm written from scratch. Python implementation of this algorithm from scratch can be seen at Appendix A.

```
First 1030 PCs explain more than 90% of the data
First 1581 PCs explain more than 95% of the data
First 2789 PCs explain more than 99% of the data
```

Fig.4: Principal Component Analysis

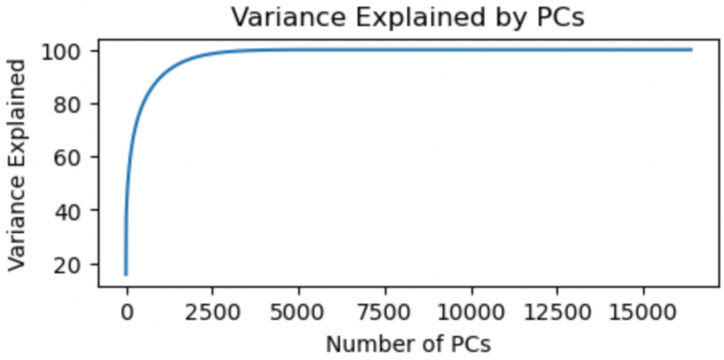

Fig.5: Variance explained for first k PCs

Description of Selected Machine Learning Algorithms

In order to classify Alzheimer disease, 3 different algorithms are chosen. These algorithms are Logistic Regression, Support Vector Machine and Neural Networks. These methods will be explained in detail.

1. Logistic Regression

Logistic regression is a fundamental and widely used statistical technique in machine learning for classification tasks like the Alzheimer MRI Classification. In the context of our project, which involves distinguishing between different classes of MRI images associated with Alzheimer's disease, logistic regression serves as a powerful tool. Unlike linear regression, which predicts continuous outcomes, logistic regression is specifically designed for predicting the probability of an instance belonging to a particular class. In our case, it models the probability that an MRI image falls into the corresponding categories. The logistic regression algorithm employs the logistic function to constrain the output to the range [0, 1], mapping the linear combination of input features to a probability score. During training, the model adjusts its parameters to optimize the likelihood of the observed class labels. The decision boundary generated by logistic regression aids in effectively separating the two classes, enabling accurate classification of Alzheimer and non-Alzheimer MRI images within our extensive dataset of 6400 pictures obtained from Kaggle. Since we have 4 classes, we will use one vs rest method to basically convert multiclass logistic regression to 3 different binary logistic regression for 4 classes.

Logistic regression uses sigmoid function to calculate probability of class by putting values between 0 and 1. So, if probability higher than threshold, let's say 0.5, it will predict "1", otherwise it will predict "0". In multiclass case, it will be little different. Assume input matrix is $X \in R_{nxp}$ where n is the number of samples and p is the number of features for each sample. The labels are $Y \in R_{nx1}$. After one hot encoding, $Y_{ONEHOT} \in R_{nx4}$. In this case, a weight vector of $W \in R_{px4}$ will be used. Sigmoid function can be seen as:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

$$Z = X \cdot W \qquad P = \sigma(Z)$$

$$P = \frac{1}{1 + e^{-X \cdot W}}$$

So, to calculate each class probability in multiclass logistic regression, multinomial logistic regression using softmax function is used like below:

$$P_1 = \frac{e^{W_1 X}}{1 + \sum_{k=1}^{3} e^{W_k X}}$$

$$P_2 = \frac{e^{W_2 X}}{1 + \sum_{k=1}^{3} e^{W_k X}}$$
$$P_3 = \frac{e^{W_3 X}}{1 + \sum_{k=1}^{3} e^{W_k X}}$$

$$P_4 = 1 - (P_1 + P_2 + P_3)$$

Since we calculate for K-1 classes, we automatically know probability of K^{th} class, so we don't need to calculate it again. Hence, our Loss function is similar to binary cross entropy function, but not same. It is called sparse cross entropy function.

$$l(W, X) = \frac{1}{n} \sum_{i=1}^{n} log \left(\frac{e^{W_{k=Y_i} X_i}}{1 + \sum_{k=0}^{class\_num} e^{W_k X_i}} \right)$$

To get optimum model, we should minimize this loss function. Derivative of loss function multinomial logistic regression can be seen below in vectorized manner. [3]

$$\nabla f(W) = \frac{1}{n} (X^T (Y_{ONEHOT} - P) + regularization$$

Also, I added L2, Ridge regularization parameter for gradient descent.

$$L2(regularization) = 2\lambda W$$

Then update parameters for each step. This way our model will converge.

$$W(t+1) = W(t) - \alpha \cdot \nabla f(W)$$

Python implementation of this algorithm from scratch can be seen at Appendix B.

2. Support Vector Machine (SVM)

The Support Vector Machine (SVM) algorithm emerges as a robust methodology for our multi-class classification task such as the Alzheimer MRI classification, encompassing four distinct classes. SVMs are renowned for their efficacy in discriminating between multiple classes by identifying optimal hyperplanes in the feature space. In our project, SVM works by finding the decision boundaries that maximize the margin between different classes, aiming to achieve the greatest separation between Alzheimer-related categories within the dataset. SVMs are particularly adept at handling high-dimensional data, making them well-suited for the complexity of MRI images. During training, the algorithm seeks to identify support vectors that are instances crucial for defining the decision boundaries ensuring a robust and accurate classification. The flexibility of SVMs allows them to adapt to intricate patterns within the 6400 MRI images obtained from Kaggle, contributing to the nuanced analysis of Alzheimer-related variations across the multi-class spectrum.

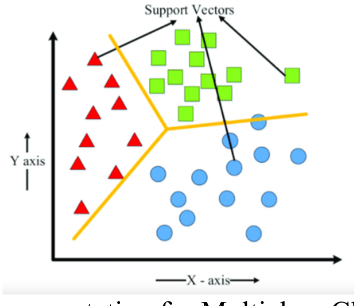

Fig.6: SVM Representation for Multiclass Classification [4]

Support vector machine works very similar to logistic regression except the activation function. After multiplying W with X, instead of applying sigmoid function, it will be compared with 1 or -1. If the result is higher than 1, prediction will be considered as 1, if its less than -1, prediction will be considered as -1. Using this comparison, we have a loss function called Hinge loss function. Assume d is number of features, W_j is weight vector of class j, λ is regularization parameter, N is number of instances, X_i is feature vector for instance i, y_i is label for instance i and W_{yi} is weight vector of correct class of instance i; our Hinge loss function to optimize will become:

$$J(W) = \frac{1}{2}\sum_{j=1}^{d} W^T \cdot W_j + \lambda \sum_{i=1}^{N} \max\,(0{,}1 - y_i.\,(W_{yi}^T \cdot X_i))$$

By taking derivative of this, we got our gradient:

$$\nabla_{\mathrm{wj}} J(W) = W_j - \lambda \sum_{i=1}^{N} \delta_{\mathrm{yi,j}} \cdot 1\big(y_i \cdot \big(W_j^T \cdot X_i\big) < 1\big) \cdot X_i$$

Where $\delta_{yi,j}$ is Kronecker delta equals to 1 if y_j=j, and 0 otherwise. [5] Update rule of gradient descent can be seen below:

$$W_{ij} = W_{ij} - \eta(\,\nabla_{\mathrm{wij}} J(W)\,)$$

In this project, since using for loops are inefficient for image tasks, one vs rest approach of support vector machine implemented in vectorized manner. Predictions can be made by calculating scores for each class and taking highest value, just like in logistic regression SoftMax output.

$$scores = X \cdot W$$

$$prediction = argmax(scores)\; for\; each\; row$$

Python implementation of this algorithm from scratch can be seen at Appendix C.

3. Neural Networks

The application of a neural network, specifically a multi-layer perceptron (MLP), proves instrumental in our endeavor to classify the Alzheimer MRI dataset into its four distinct classes. Neural networks are a class of machine learning algorithms inspired by the structure and function of the human brain. In the context of our project, the perceptron consists of layers of interconnected nodes that collectively learn intricate patterns from the input features. The neural network's architecture allows it to capture complex relationships and hierarchical representations within the high-dimensional MRI data. During training, the network adjusts its weights and biases to minimize the difference between predicted and actual class labels, refining its ability to discern subtle variations among the four classes. The non-linear activation functions employed in each node enable the model to learn and represent intricate features, essential for the nuanced classification of Alzheimer-related patterns within the extensive dataset of 6400 MRI images sourced from Kaggle. The adaptability and capacity for feature extraction make neural networks a powerful tool for uncovering intricate patterns in multi-class classification tasks.

Neural networks include multiple layers and each layer have some number of neurons. At each layer some activation function, ∅ which may be sigmoid function like logistic regression or may be "ReLU" or "leaky ReLU" or "Hyperbolic Tangent" etc. Assume that weight matrix between L^{th} layer and $(L+1)^{th}$ layer is W^L matrix with n(L) x n(L+1) dimensions while n is number of neurons, β is bias. So forward propagation algorithm can be defined as below.

$$X^{l+1} = \emptyset\big(V^{l+1}\big) = \emptyset\big(X^l \cdot W^l + \beta^l\big)$$

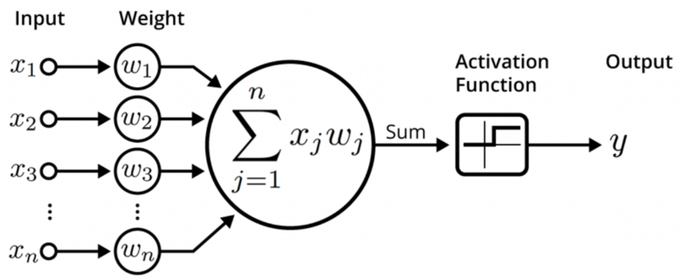

Fig.7: Illustration of Basic Neural Network [6]

Back propagation is more complicated. If we define Loss function L as δ^l. We can find derivatives using chain rule. [7]

$$\delta^l \triangleq \frac{\partial L}{\partial V^l}$$

$$\delta^{l-1} = \emptyset'\left(V^{l-1}\right)\left(\delta^l W^{l^T}\right)$$

$$\frac{\partial L}{\partial W^l} = \frac{1}{n}\left(X^{l^T}\delta^l\right)$$

$$\frac{\partial L}{\partial \beta^l} = \frac{1}{n}\sum\left(\delta^l\right)$$

Using these derivatives, weights can be updated while going from last layer to first layer.

$$W^l(k+1) = W^l(k) - \alpha\frac{\partial L}{\partial W^l}$$

$$\beta^l(k+1) = \beta^l(k) - \alpha\frac{\partial L}{\partial W^l}$$

Since using for loops is inefficient for image tasks, NN algorithm implemented in vectorized manner in this project. Python implementation of this algorithm from scratch can be seen at Appendix D.

Simulation Setup

Image data given in different folders without extra csv file. In order to split my data easily, I created dataframe with filenames and labels. After creating dataframe, I splitted for training, validation and test sets using hand-written function. Then I converted images to NumPy arrays using matplotlib's imread function. Then, flattened every image and added as rows. So, I had array with 6400 rows and 16384 columns. 6400 represents number of images and 10859 is just flattened version of 128x128 image pixels after deleting columns with zero variance.

	filename	class_label
0	non.jpg	Non_Demented
1	non_10.jpg	Non_Demented
2	non_100.jpg	Non_Demented
3	non_1000.jpg	Non_Demented
4	non_1001.jpg	Non_Demented
...	...	...
6395	moderate_63.jpg	Moderate_Demented
6396	moderate_64.jpg	Moderate_Demented

Fig.8: Dataframe created by checking folders

After I created numpy arrays, I normalized all X feature columns using handwritten StandardScaler() class, with fitting training X values to get better and faster results. Because normalized features will converge better with gradient descent. In order to make training faster, I applied PCA for training data. Also, I applied one-hot-encoding using handwritten OneHotEncoder() class for Y labels for multi class classification task.

$$X_{normalized} = \frac{X - mean(X_{train})}{std(X_{train})}$$

$$Y = \begin{bmatrix} 0 \\ 1 \\ 2 \\ 3 \end{bmatrix} \dashrightarrow Y_{ONEHOT} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

```
(5120, 10859) (5120, 4)
(640, 10859) (640, 4)
(640, 10859) (640, 4)
```

Fig.9: X and Y arrays after train-val-test split

Simulation Results

Three different machine learning algorithm models trained for this project with using different hyperparameters such as learning rate, regularization type, regularization term lambda, batch size, weight initialization techniques and layer-neuron number for just neural network algorithm. Since we are comparing our best models for each algorithm, confusion matrix and classification report of test set for each algorithm will be shared. Validation set used for hyperparameter optimization only. For all models, grid search algorithm applied and more than 50 models with different hyperparameters trained for each algorithm. Batch size 512 is used since it is still fast compared to lower batch size but also can represent stochastic nature of gradient descent and converge faster and better.

Best Model for Logistic Regression:
Val. Acc. = 93.28% (at max epoch)
Test Acc. = 95.78%
Batch Size: 512
Weight Initialization: zero
Learning Rate: 0.01 with momentum 0.99
Regularization: L2 with Lambda 0.01
Max Epoch: 1000
Training Time: 07:46 minutes

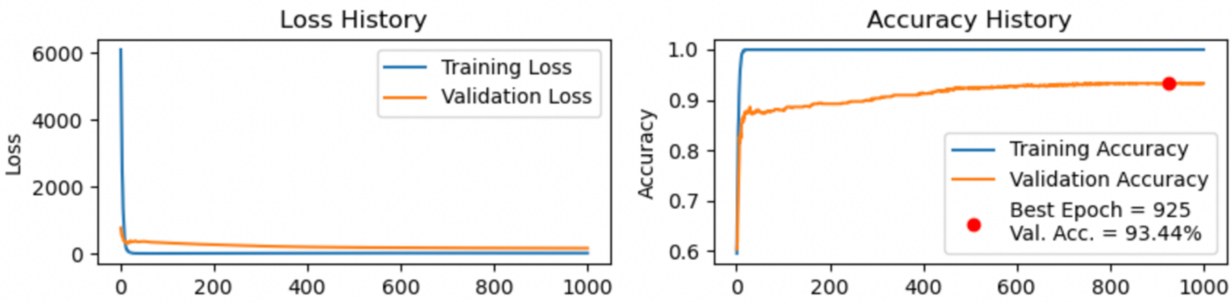

Fig.10: Loss and accuracy plots for best model of logistic regression

```
Accuracy is: 95.78 %
F1 Score is: 95.78 %
Precision of Class 0 is: 97.47 %

Classification Report:
              precision    recall  f1-score   support
0               97.47 %   96.25 %   96.86 %       320
1               93.51 %   96.43 %   94.95 %       224
2               95.45 %   93.33 %   94.38 %        90
3              100.00 %   83.33 %   90.91 %         6
accuracy                            95.78 %       640
macro avg       96.61 %   92.34 %   94.27 %       640
weighted avg    95.82 %   95.78 %   95.78 %       640
```

Fig.11: Classification Report for best model of logistic regression

```
Confusion Matrix:
      0    1   2  3
0   308   10   2  0
1     7  216   1  0
2     1    5  84  0
3     0    0   1  5
```

Fig.12: Confusion Matrix for best model of logistic regression

Best Model for Support Vector Machine:
Val. Acc. = 93.90% (at max epoch)
Test Acc. = 95.94%
Batch Size: 512
Weight Initialization: zero
Learning Rate: 0.001 static
Regularization: L2 with Lambda 0.1
Max Epoch: 2500
Training Time: 08:12 minutes

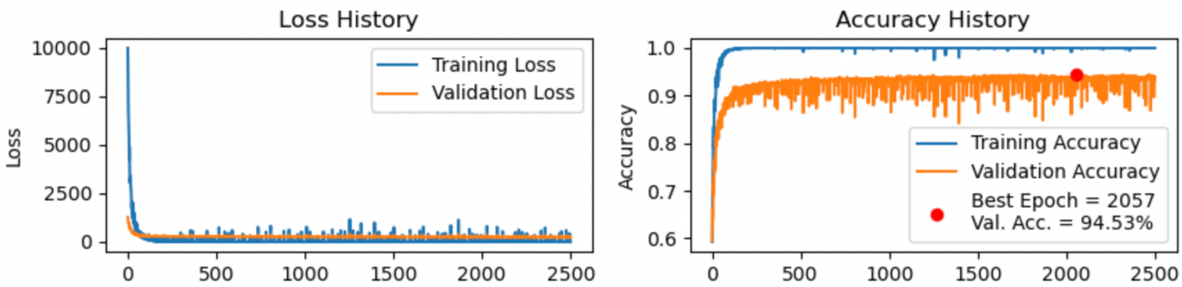

Fig.13: Loss and accuracy plots for best model of support vector machine

```
Accuracy is: 95.94 %
F1 Score is: 95.94 %
Precision of Class 0 is: 97.48 %
```

Classification Report:

	precision	recall	f1-score	support
0	97.48 %	96.56 %	97.02 %	320
1	93.91 %	96.43 %	95.15 %	224
2	95.45 %	93.33 %	94.38 %	90
3	100.00 %	83.33 %	90.91 %	6
accuracy			95.94 %	640
macro avg	96.71 %	92.41 %	94.37 %	640
weighted avg	95.97 %	95.94 %	95.94 %	640

Fig.14: Classification Report for best model of support vector machine

Confusion Matrix:

	0	1	2	3
0	309	9	2	0
1	7	216	1	0
2	1	5	84	0
3	0	0	1	5

Fig.15: Confusion Matrix for best model of support vector machine

Best Model for Neural Networks:
Val. Acc. = 96.09% (at max epoch)
Test Acc. = 98.44%
Hidden Layers: [(64, 'leaky-relu'), (32, 'leaky-relu')]
Batch Size: 512
Weight Initialization: he-normal
Learning Rate: 0.01 static
Regularization: L2 with Lambda 0.0001
Max Epoch: 2500
Training Time: 09:43 minutes

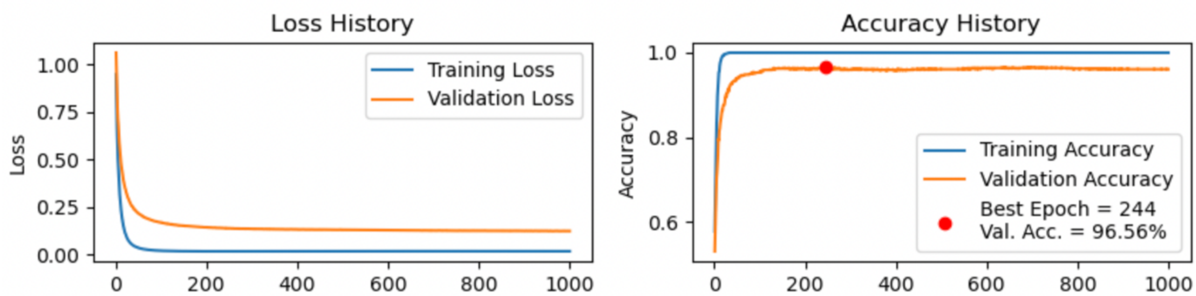

Fig.16: Loss and accuracy plots for best model of neural networks

Accuracy is: 98.44 %
F1 Score is: 98.43 %
Precision of Class 0 is: 99.06 %

Classification Report:

	precision	recall	f1-score	support
0	99.06 %	98.44 %	98.75 %	320
1	97.80 %	99.11 %	98.45 %	224
2	97.78 %	97.78 %	97.78 %	90
3	100.00 %	83.33 %	90.91 %	6
accuracy			98.44 %	640
macro avg	98.66 %	94.66 %	96.47 %	640
weighted avg	98.44 %	98.44 %	98.43 %	640

Fig.17: Classification Report for best model of neural networks

Confusion Matrix:

	0	1	2	3
0	315	4	1	0
1	2	222	0	0
2	1	1	88	0
3	0	0	1	5

Fig.18: Confusion Matrix for best model of neural networks

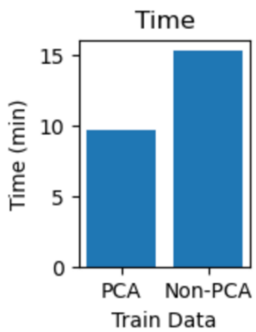

Fig.19: Time comparison for PCA and non-PCA data

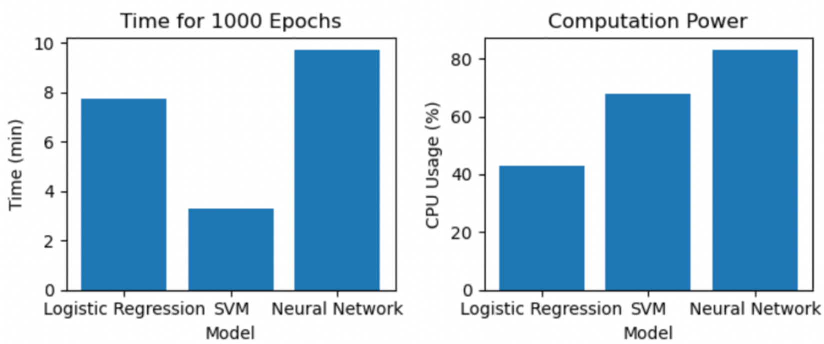

Fig.20: Time and computation power needed for different models (batch size 512)

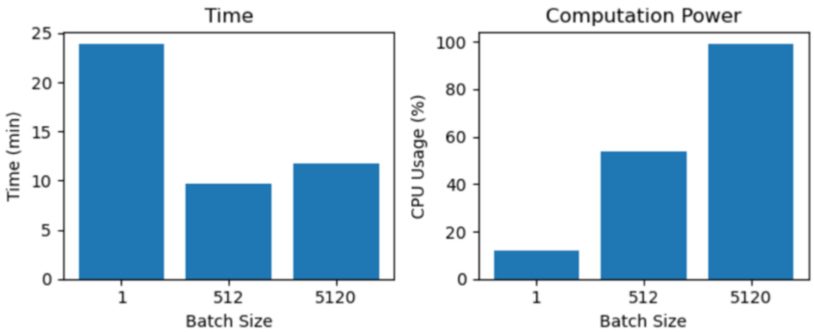

Fig.21: Time and CPU usage of training for different batch sizes

Discussion on Performance of the Algorithms

As it can be seen from results in table 1, neural network algorithm resulted best results. Since both logistic regression and support vector machine are algorithms based on linear separation of classes, they both performed similar results while support vector machine got better results than logistic regression by small percent. Support vector machine accuracy plot seems bouncy but due to stochastic nature, I was able to get best results using larger regularization parameter in this algorithm. Logistic Regression accuracy increased when we compared to first report after implementing batch gradient descent and using large grid search. Both logistic regression and SVM can be implemented with polynomial kernels but since I got good results using linear kernels, I didn't use polynomial kernels instead passed to implement deep learning algorithm: neural networks.

To compare times, support vector machine is fastest algorithm per epoch, but it converges slower than logistic regression so they both take similar time to obtain best results. Neural networks is slowest model but not too slow, when we compare efficiency, neural network is still best algorithm since its results are far better than other algorithms. It used more computation power when we look at CPU usage percentages, but its results are probably worth the time and computing power used by model. For neural networks, different number of layers and different neuron numbers are trained with grid search, best model found with 2 hidden layers with 64 and 32 neurons respectively. Maybe more complex models with higher number of neurons can result better but since its time to train was around 10 minutes, more complex models are not trained.

Different batch sizes are used for this project. When batch size 5120 is used which is batch gradient descent, model converges slower due to lack of stochastic nature, and model trains slower than batch size 512 which is unexpected. The reason is when we train model with full batches, there will be heavy matrix computations because images are 128x128. Because of this, CPU heats a lot and slows itself to protect. While using batch size 1 or in other name stochastic gradient descent should converge in lower number of epochs, its training takes too much time. For sweet point of time and stochastic nature, batch size 512 is used for all models when grid searching best hyperparameters.

One of important performance metric was precision of class 0 which is non demented class, and it represents not missing the people with dementia. With best model, 99.06% precision obtained for class 0 on test set. It can be said that it is very good result for 10-minute image classification training.

Algorithm	**F1-Score**	**Precision of 0**	**Power Usage**	**Time**	**Epochs**
Logistic Regression	95.78 %	97.47 %	43% CPU	07:46	1000
Support Vector Machine	95.94 %	97.48 %	68% CPU	08:12	2500
Neural Networks	98.43 %	99.06 %	83% CPU	09:43	1000

Table 1: Model Comparison on Test Set

Conclusion

The project is completed with implementing three different machine learning algorithms from scratch and training on chosen Alzheimer MRI dataset. For testing and saving result purposes, 1 class and 6 functions written from scratch to print and save results such as accuracy, confusion matrices, classification reports etc. In data preprocessing part, 4 functions written from scratch to apply some preprocessing such as standard scaling, one hot encoding, pca and finding variances of data. Alzheimer MRI classification probably can be used successfully in real life, after taking MRI images and processing it since the dataset used in this project is already processed with free surfer which is open source neuroimage data analysis and processing package. After optimizing all three algorithms using validation dataset and found best hyperparameters, test set results is reported in this report. Writing logistic regression, support vector machine and neural network algorithms were challenging but educative. While implementing these algorithms using for loops is intuitive, implementing same algorithms in vectorized manner was more challenging but needed for the purpose of this project, which is image classification. Otherwise, training take 10-100 times more time, depending on dataset since while python kernel using only one cpu core, numpy library can use all cpu cores to make parallel computing. Finally, this project was helpful and educative to learn principles behind machine learning and deep learning algorithms and to practice python with writing these algorithms using python and numpy from scratch instead of blindly using classes from machine learning specific libraries.

References

[1] “2023 Alzheimer’s Disease Facts and Figures,” Alzheimer’s Dement, vol. 19, no. 4, pp. 1598–1695, Feb. 2023. doi:10.1002/alz.13016

[2] S. Kumar, “Alzheimer MRI preprocessed dataset,” Kaggle, https://www.kaggle.com/datasets/sachinkumar413/alzheimer-mri-dataset (accessed Oct. 16, 2023).

[3] S. Yang, “Multiclass logistic regression from scratch,” Medium, https://towardsdatascience.com/multiclass-logistic-regression-from-scratch-9cc0007da372 (accessed Nov. 20, 2023).

[4] R. Muzzammel and A. Raza, “A support vector machine learning-based protection technique for MT-HVDC systems,” Energies, vol. 13, no. 24, p. 6668, 2020. doi:10.3390/en13246668

[5] Sidharth, “Implementing SVM from scratch using Python,” PyCodeMates, https://www.pycodemates.com/2022/10/implementing-SVM-from-scratch-in-python.html (accessed Dec. 20, 2023).

[6] N. McCullum, “Deep Learning Neural Networks explained in plain English,” freeCodeCamp.org, https://www.freecodecamp.org/news/deep-learning-neural-networks-explained-in-plain-english/ (accessed Nov. 20, 2023).

[7] O. Aflak, “Neural network from scratch in Python,” Medium, https://towardsdatascience.com/math-neural-network-from-scratch-in-python-d6da9f29ce65 (accessed Dec. 20, 2023).

Mehmet Bayık
21802166
20.12.2023

Appendix A – Data Analysis and Preprocessing

```
# %%
"""
dependencies:
  - python=3.8.17
  - numpy=1.24.0
  - matplotlib=3.7.1
  - pandas=2.0.2
"""

# %%
import os
import random
import datetime
from itertools import product

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

# import random
# random.seed(42)
# np.random.seed(42)
# np.random.RandomState(42)
# os.environ['TF_DETERMINISTIC_OPS'] = '1'

# %%
finish_sound = "afplay /Users/mehmet/Documents/vs-code/winsquare.mp3"
# play sound when finished
# os.system(finish_sound)

# %%
classes = ['Non_Demented','Very_Mild_Demented','Mild_Demented','Moderate_Demented']
folder_path = '/Users/mehmet/Documents/vs-code/EEE485-Statistical-Learning-and-
Data-Analytics/dataset/'
datafile = ['','','','']
dataframe = pd.DataFrame()
for i in range(len(classes)):
    datafile[i] = sorted(os.listdir(folder_path + classes[i]))
    filenames = pd.DataFrame(datafile[i], columns=['filename'])
    class_labels = pd.DataFrame(np.full((len(datafile[i]),1), classes[i]),
columns=['class_label'])
    dataframe = pd.concat([dataframe, pd.concat([filenames, class_labels],
axis=1)], axis=0)
dataframe = dataframe.reset_index(drop=True)
dataframe['class_label'] = dataframe['class_label'].str[:]
classes = dataframe['class_label'].unique()
dataframe

# %%
```

```
class_counts = []
for i in datafile:
    class_counts.append(len(i))
    print('There are',len(i),'images belonging to',classes[datafile.index(i)],
'class')
print('Total number of images:', sum(class_counts))

# %%
# Plot class distribution

fig, ax = plt.subplots(1, 1)
fig.set_size_inches(5, 2)
bins = np.linspace(0 - .25, 3 + .25, 8)
ax.hist(dataframe['class_label'].str[:-9].values,bins=bins)
ax.set_title('Class Distribution')
ax.set_xlabel('Class')
ax.set_ylabel('Number of Images')
plt.show()

# %%
# Display 1 random images from each class

fig, ax = plt.subplots(1, 4)
fig.set_size_inches(10, 5)
for i in range(len(classes)):
    for j in range(1):
        # get random image dataframe
        start = dataframe[dataframe['class_label']==classes[i]].first_valid_index()
        end = dataframe[dataframe['class_label']==classes[i]].last_valid_index()
        sample = np.random.randint(start, end)-start
        dataframe[dataframe['class_label']==classes[i]].iloc[sample,0]
        random_image =
dataframe[dataframe['class_label']==classes[i]].iloc[sample,0]
        filename = folder_path + classes[i] + '/' + random_image
        ax[i].imshow(plt.imread(filename), cmap='gray')
        ax[i].set_title(classes[i])
        ax[i].axis('off')
plt.show()

# %%
# Convert all images to numpy array and flatten them

folderpath ='/Users/mehmet/Documents/vs-code/EEE485-Statistical-Learning-and-Data-
Analytics/dataset/'
image_data = []
for instance in dataframe['filename']:
    # find class label
    folder_name =
dataframe[dataframe['filename']==instance]['class_label'].values[0] + '/'
    image2 = plt.imread(folderpath+folder_name+instance)
    image2_flatten = image2.flatten().T
```

```
    image_data.append(image2_flatten)
image_arr = np.array(image_data)
output_labels = np.array(dataframe['class_label'].values)
image_arr.shape, output_labels.shape

# %%
# Create dataframe from image array

image_df = pd.DataFrame(image_arr)
# Rescale pixel values
#image_df = image_df/255
image_df.columns = image_df.columns.astype(str)
image_df['filename'] = dataframe['filename']
image_df['class_label'] = output_labels
image_df

# %%
def train_test_split(dataframe, test_size, validation_size=0, random_state=42):
    # Function to split pandas dataframe into train, test and validation sets
    """ Split data into train and test sets.
    Args:
        dataframe (pandas dataframe): Input Pandas Dataframe
        test_size (float): float between 0 and 1
        validation_size (float): float between 0 and 1
        random_state (int): random seed
    """
    class_labels = dataframe['class_label'].unique()
    dataframe = dataframe.sample(frac=1,
random_state=random_state).reset_index(drop=True)
    train_size = 1 - test_size - validation_size
    # train
    train_df = pd.DataFrame()
    for i in range(len(classes)):
        train_df = pd.concat([train_df,
dataframe[dataframe['class_label']==classes[i]].iloc[:round(class_counts[i]*train_s
ize),:]], axis=0)
    train_df = train_df.sample(frac=1,
random_state=random_state).reset_index(drop=True)

    # test
    test_df = pd.DataFrame()
    for i in range(len(classes)):
        test_df = pd.concat([test_df,
dataframe[dataframe['class_label']==classes[i]].iloc[round(class_counts[i]*(train_s
ize+validation_size)):,:]], axis=0)
    test_df = test_df.sample(frac=1,
random_state=random_state).reset_index(drop=True)

    if validation_size > 0:
        # validation
        val_df = pd.DataFrame()
```

```
        for i in range(len(classes)):
            val_df = pd.concat([val_df,
dataframe[dataframe['class_label']==classes[i]].iloc[round(class_counts[i]*train_si
ze):round(class_counts[i]*(1-validation_size)),:]], axis=0)
        val_df = val_df.sample(frac=1,
random_state=random_state).reset_index(drop=True)

        return train_df, val_df, test_df
    return train_df, test_df

# %%
def delete_zero_columns(dataframe):
    # X: dataframe
    k = dataframe['filename']
    y = dataframe['class_label']
    X = dataframe.drop(['filename','class_label'], axis=1).values
    mean_ = np.mean(X, axis=0)
    scale_ = np.std(X - mean_, axis=0)
    if np.any(scale_ == 0):
        mask = np.where(scale_ == 0)
    X_new = np.delete(X, mask, axis=1)
    out_df = pd.concat([pd.DataFrame(X_new), k, y], axis=1)
    return out_df

# %%
image_df_clean = delete_zero_columns(image_df)
train_df, val_df, test_df = train_test_split(image_df_clean, test_size=0.1,
validation_size=0.1, random_state=42)

train_df.shape,val_df.shape, test_df.shape

# %%
class StandardScaler():
    # StandardScaler Class written from scratch similar to
sklearn.preprocessing.StandardScaler
    def __init__(self):
        pass

    def fit(self, X):
        self.mean_ = np.mean(X, axis=0)
        self.scale_ = np.std(X - self.mean_, axis=0)
        if np.any(self.scale_ == 0):
            self.scale_ = np.where(self.scale_ == 0, 1, self.scale_)
        return self

    def transform(self, X):
        return (X - self.mean_) / self.scale_

    def fit_transform(self, X):
        return self.fit(X).transform(X)
```

```
# %%
class OneHotEncoder():
    def __init__(self):
        pass

    def fit(self, classes_encode=None, y=None):
        """ Which class is encoded as which number
        Args:
            classes_encode (dict): Dictionary of classes and their encoded values
        """
        if classes_encode is None:
            self.classes_encode = {class_:i for i, class_ in
enumerate(np.unique(y))}
        if y is None:
            self.classes_encode = classes_encode
        return self

    def transform(self, y):
        """ One hot encoder
        Args:
            y (pandas dataframe): Output labels
        """

        for i in y:
            y = y.replace(i, self.classes_encode[i])
        # One-hot encoding
        y_onehot = np.zeros((len(y.values), 4))
        for i in range(len(y)):
            y_onehot[i][y[i]] = 1

        return y_onehot

# %%
X_train = train_df.drop(['filename','class_label'], axis=1).values
y_train = train_df['class_label']
X_val = val_df.drop(['filename','class_label'], axis=1).values
y_val = val_df['class_label']
X_test = test_df.drop(['filename','class_label'], axis=1).values
y_test = test_df['class_label']

# Scale data using StandardScaler
scaler = StandardScaler()
scaler.fit(X_train)
X_train = scaler.transform(X_train)
X_val = scaler.transform(X_val)
X_test = scaler.transform(X_test)

# Encode Labels with given dictionary
classes_encode = {'Non_Demented':0, 'Very_Mild_Demented':1, 'Mild_Demented':2,
'Moderate_Demented':3}
encoder = OneHotEncoder()
```

```
encoder.fit(classes_encode)
y_train = encoder.transform(y_train)
y_val = encoder.transform(y_val)
y_test = encoder.transform(y_test)

print(X_train.shape, y_train.shape,'\n', X_val.shape, y_val.shape,'\n',
X_test.shape, y_test.shape)

# %%
# # Save data to numpy arrays
# np.save('dataset/original-numpy/X_train.npy', X_train)
# np.save('dataset/original-numpy/y_train.npy', y_train)
# np.save('dataset/original-numpy/X_val.npy', X_val)
# np.save('dataset/original-numpy/y_val.npy', y_val)
# np.save('dataset/original-numpy/X_test.npy', X_test)
# np.save('dataset/original-numpy/y_test.npy', y_test)

# %%
# PCA on training data
images = X_train
mean = np.mean(images, axis=0)
data_mn = images - mean
cov_mat = np.matmul(data_mn.T,data_mn)
eigenvalues, eigenvectors = np.linalg.eig(cov_mat)

#sort the eigenvalues in descending order
idx = eigenvalues.argsort()[::-1]
eigenvalues = eigenvalues[idx]
eigenvectors = eigenvectors[:,idx]
total_variance = np.sum(eigenvalues)

#calculate the variance explained by each eigenvalue for first 10 eigenvalues
variance_explained = [(i/total_variance)*100 for i in eigenvalues]

# %%
percs=[90,95,99]
for perc in percs:
    printed = 0
    for i in range(len(variance_explained)):
        variance_until_i = np.sum(variance_explained[:i+1])
        if i < 10:
            pass
            #print(str(i+1)+'. principal component
|','PVE:',variance_explained[i].round(3),'| Cumulative
PVE:',variance_until_i.round(3))

        if variance_until_i >= perc:
            if printed == 0:
                printed = 1
                print('First',str(i+1),'PCs explain more than {}% of the
data:'.format(perc), variance_until_i.round(3))
```

```
# %%
variance_until_i_list = []
for i in range(len(variance_explained)):
    variance_until_i = np.sum(variance_explained[:i+1])
    variance_until_i_list.append(variance_until_i)

# %%
# Plot variance until ith list
fig, ax = plt.subplots(1, 1)
fig.set_size_inches(5, 2)
ax.plot(variance_until_i_list)
ax.set_title('Variance Explained by PCs')
ax.set_xlabel('Number of PCs')
ax.set_ylabel('Variance Explained')
plt.show()

# %%
# Reconstruct images using first k principal components
k=2789
first_k_eigen = eigenvectors[:,:k].T
#projection = np.matmul(images-mean,first_k_eigen.T)
#projection = np.matmul(projection,first_k_eigen) + mean
# Use float64
projection = np.matmul(images-mean,first_k_eigen.T).astype(np.float64)
projection = np.matmul(projection,first_k_eigen).astype(np.float64) +
mean.astype(np.float64)
X_train_pca = projection
X_train_pca.shape

# %%
# Save pca applied train data to numpy array

# np.save('dataset/original-numpy/X_train_pca_2.npy', X_train_pca)
```

Mehmet Bayık
21802166
20.12.2023

Appendix B – Logistic Regression

```
# %%
"""
dependencies:
  - python=3.8.17
  - numpy=1.24.0
  - matplotlib=3.7.1
  - pandas=2.0.2
"""
import os
import random
import datetime
from itertools import product

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

finish_sound = "afplay /Users/mehmet/Documents/vs-code/winsquare.mp3"
# play sound when finished
# os.system(finish_sound)

# %%
# Read data from npy file ( already preprocessed )
filename = 'original-numpy'
X_train = np.load(f'dataset/{filename}/X_train_pca_2.npy')
X_val = np.load(f'dataset/{filename}/X_val.npy')
X_test = np.load(f'dataset/{filename}/X_test.npy')
y_train = np.load(f'dataset/{filename}/y_train.npy')
y_val = np.load(f'dataset/{filename}/y_val.npy')
y_test = np.load(f'dataset/{filename}/y_test.npy')
print(X_train.shape, y_train.shape,'\n', X_val.shape, y_val.shape,'\n',
X_test.shape, y_test.shape)

# %%
class LogisticRegression():
    # Logistic Regression Model written from scratch without Bias w0
    def __init__(self, seed=42):
        np.random.seed(seed)
        self.W = None
        self.now = None
        self.print_result = True
        self.history_steps1 = None
        self.history = None
        self.validation_accuracy = None

    def validation_accuracy(self):
        return self.validation_accuracy

    def history(self):
```

```
        return self.history

    def load_history(self):
        pd_hist = pd.read_csv(f'model-comparison/{self.now}/history.csv')
        self.history = np.array(pd_hist.iloc[:,1:])

    def plot(self, save = True):
        # Save history as csv file
        history_local = self.history
        if type(history_local) is not pd.DataFrame:
            history_df = pd.DataFrame(history_local)
        if save == True:
            hist_csv_file = f'model-comparison/{self.now}/history.csv'
            with open(hist_csv_file, mode='w') as f:
                history_df.to_csv(f)
        # Plot Loss and Accuracy History as Subplots
        fig, ax = plt.subplots(1, 2)
        fig.set_size_inches(10, 2)
        index = np.arange(1,self.history.shape[1]+1)*self.history_steps1

        ax[0].plot(index, self.history[0], label='Training Loss')
        ax[0].plot(index, self.history[2], label='Validation Loss')
        ax[0].set_title('Loss History')
        ax[0].set_xlabel('Epoch')
        ax[0].set_ylabel('Loss')
        ax[0].legend()
        # find best validation accuracy and its epoch
        best_val_acc = np.max(self.history[3])
        best_val_acc_epoch = (np.argmax(self.history[3]) + 1)*self.history_steps1
        label='Best Epoch = '+str(best_val_acc_epoch)+'\nVal. Acc. =
'+str((best_val_acc*100).round(2))+ '%'
        ax[1].plot(index, self.history[1], label='Training Accuracy')
        ax[1].plot(index, self.history[3], label='Validation Accuracy')
        ax[1].plot(best_val_acc_epoch, best_val_acc, 'ro', label=label)
        ax[1].set_title('Accuracy History')
        ax[1].set_xlabel('Epoch')
        ax[1].set_ylabel('Accuracy')
        ax[1].legend()
        if save is True and self.now is not None:
            plt.savefig(f'model-comparison/{self.now}/plot.png')
        if self.print_result == True:
            plt.show()
        else:
            plt.close(fig)


    def validation(self, X_nonbiased, y, W, lmbda):
        # add bias
        ones=np.ones(X_nonbiased.shape[0])
        X=np.c_[ones,X_nonbiased]
        # Find loss and accuracy on validation set
```

```
        y_onehot = y # y is already one-hot encoded
        Z = - X @ W
        P = np.exp(Z) / np.sum(np.exp(Z), axis=1, keepdims=True)
        loss = - np.sum(y_onehot * np.log(P)) + lmbda * np.sum(W**2)
        y_pred = self.predict(X_nonbiased)
        accuracy = np.mean(y_pred == np.argmax(y, axis=1))
        return loss, accuracy

    def fit(self, X_nonbiased, y, X_val, y_val, now = None, print_result =
True,max_epoch=400,
            batch_size=5120, weight_init='zero', lr=0.01, lr_type = 'static',
regularization='l2: 0.01',
            history_steps = 50, print_step = 100):
        start_time = datetime.datetime.now()
        # if there isn't model-comparison folder, create it
        if not os.path.exists('model-comparison'):
            os.mkdir('model-comparison')
        self.print_result = print_result
        if now is not None:
            self.now = now
        # Create folder for current model
            if not os.path.exists('model-comparison/'+now):
                os.mkdir('model-comparison/'+now)

        self.history_steps1 = history_steps
        self.history = np.zeros((4,max_epoch//history_steps))
        y_onehot = y # y is already one-hot encoded
        lr_print = str(lr) + ' ' + lr_type
        model_specs = 'LR | Batch Size: {} | Weight Init. {} | lr: {} |
Regularization: {} | Max Epoch: {} |'.format(batch_size, weight_init, lr_print,
regularization, max_epoch)

        # add bias
        ones=np.ones(X_nonbiased.shape[0])
        X=np.c_[ones,X_nonbiased]
        N = X.shape[0]

        # Initialize weights ( shape = features x classes matrix )
        if weight_init == 'zero':
            self.W = np.zeros((X.shape[1], y_onehot.shape[1]))
        elif weight_init == 'uniform':
            self.W = np.random.uniform(0, 1, (X.shape[1], y_onehot.shape[1]))
        elif weight_init == 'normal':
            self.W = np.random.normal(0, 1, (X.shape[1], y_onehot.shape[1]))

        # Print loss and accuracy every 100 iterations or every max_iter//10
iterations if max_iter >= 1000
        if max_epoch >= 1000:
            print_step = max_epoch // 10

        # Gradient Descent
```

```
        for epoch in range(1, max_epoch+1):
            # Shuffle all data X and y in the same order every epoch
            shuffle_index = np.arange(X.shape[0])
            np.random.shuffle(shuffle_index)
            X = X[shuffle_index]
            y_onehot = y_onehot[shuffle_index]

            for iteration in range(X.shape[0]//batch_size):
                X_batch = X[batch_size*iteration:batch_size*(iteration+1)]
                y_batch = y_onehot[batch_size*iteration:batch_size*(iteration+1)]
                Z_batch = - X_batch @ self.W
                # For numerical stability
                ### Z_batch = Z_batch - np.max(Z_batch, axis=1, keepdims=True)
                # Logistic function to find probabilities
                P_batch = np.exp(Z_batch) / (np.sum(np.exp(Z_batch), axis=1,
keepdims=True))
                N_batch = batch_size
                # Derivative of Residual ( log-loss )
                # P_batch = Softmax(- X_batch @ self.W)
                dRSS = (2/N_batch)*(X_batch.T @ (y_batch - P_batch))
                # Choose regularization
                if regularization[0:2] == 'l2':
                    # L2 regularization
                    lmbda = float(regularization[4:])
                    dRegTerm = 2 * (lmbda) * (N_batch/N) * self.W
                    # Bias term is not regularized
                    dRegTerm[0] *= 0

                elif regularization[0:2] == 'l1':
                    # L1 regularization
                    lmbda = float(regularization[4:])
                    dRegTerm = lmbda * np.sign(self.W)
                    # Bias term is not regularized
                    dRegTerm[0] *= 0
                else:
                    # No regularization
                    lmbda = 0
                    dRegTerm = 0
                # Calculate gradient
                gradient = dRSS + dRegTerm

                if lr_type[0:8] == 'momentum':
                    if epoch == 1:
                        last_gradient = gradient
                    else:
                        momentum = float(lr_type[10:])
                        gradient = gradient + momentum * last_gradient
                        last_gradient = gradient

                # Update weights
                # ( W already has bias term, so we don't need seperate update,
```

```
                # W is (features+1) x classes matrix: bias is in the first row
                # and bias is not regularized ) W = (16864+1) x 10

                self.W = self.W - lr * (N_batch/N) * gradient

                # Change learning rate if lr_type is adaptive
                if lr_type == 'adaptive':
                    if epoch % 300 == 0:
                        lr = lr * 0.5
                        if print_result == True:
                            print('Learning rate changed to', lr)

            # Calculate loss and accuracy every 50 epochs:
            if epoch % history_steps == 0:
                # After each x epoch, calculate loss and accuracy on validation set
                Z = - X @ self.W
                # Numerical stability
                ### Z = Z - np.max(Z, axis=1, keepdims=True)
                P = np.exp(Z) / np.sum(np.exp(Z), axis=1, keepdims=True)

                loss = - np.sum(y_onehot * np.log(P)) + lmbda * np.sum(self.W**2)
                accuracy = np.mean(self.predict(X_nonbiased) == np.argmax(y,
axis=1))
                val_loss = self.validation(X_val, y_val, self.W, lmbda)[0]
                val_acc = self.validation(X_val, y_val, self.W, lmbda)[1]
                self.validation_accuracy = val_acc
                self.history[:,(epoch//history_steps)-1] = np.array([loss,
accuracy, val_loss, val_acc])

                # Print loss and accuracy every 100 epochs
                if epoch % print_step == 0:
                    line1 = 'Epoch: ' + str(epoch)
                    line2 = ' | Loss: ' + str(loss) + ' | Accuracy: ' +
str(accuracy)[0:5]
                    line3 = ' | Val. Loss: ' + str(val_loss) + ' | Val. Acc: ' +
str(val_acc)[0:5]
                    # line2 = ' | Loss: ' + str(round(loss)) + ' | Accuracy: ' +
str(accuracy)[0:5]
                    # line3 = ' | Val. Loss: ' + str(round(val_loss)) + ' | Val.
Acc: ' + str(val_acc)[0:5]
                    if print_result == True:
                        print(line1 + line2 + line3)
                    if now is not None:
                        with open('model-comparison/{}/log.txt'.format(now), 'a')
as f:
                            f.write(line1 + line2 + line3 + '\n')
            if epoch == max_epoch:
                end_time = datetime.datetime.now()
                if print_result == True:
                    print('Training finished. Time elapsed:', end_time -
start_time, '\n')
```

```
                    print('Accuracy: ', str(accuracy)[0:5], 'Val. Accuracy: ',
str(val_acc)[0:5])
                val_acc_print = str(val_acc*100)+ '00'
                if now is not None:
                    with open('model-comparison/{}/log.txt'.format(now), 'a') as f:
                        write_line = 'Training finished. Time elapsed: ' +
str(end_time - start_time) + '\n'
                        f.write(write_line)
                    with open('model-comparison/{}/{}-val-
acc.txt'.format(now,val_acc_print[0:5]), 'w') as f:
                        f.write(model_specs)
                    with open('model-comparison/last.txt', 'w') as f:
                        f.write(str(now))

    def predict(self, X_nonbiased):
        # add bias
        ones=np.ones(X_nonbiased.shape[0])
        X=np.c_[ones,X_nonbiased]
        Z = - X @ self.W
        # Logistic function to find probabilities
        P = np.exp(Z) / np.sum(np.exp(Z), axis=1, keepdims=True)
        # Predict class
        y = np.argmax(P, axis=1)
        return y
    def save_weights(self):
        # save history steps
        with open('model-comparison/{}/history_steps.txt'.format(self.now), 'w') as
f:
            f.write(str(self.history_steps1))
        # save weights (bias included in W)
        filename = 'model-comparison/{}/weights.npy'.format(self.now)
        np.save(filename, self.W)
    def load_weights(self, now):
        # load history steps
        with open('model-comparison/{}/history_steps.txt'.format(now), 'r') as f:
            self.history_steps1 = int(f.read())
        # load weights (bias included in W)
        filename = 'model-comparison/{}/weights.npy'.format(now)
        self.W = np.load(filename)
        self.now = now

# %%
class EvaluateModel():
    # Class to evaluate model performance, similar to sklearn.metrics
ClassificationReport and ConfusionMatrix
    def __init__(self, y_true, y_pred, str1, now, save=True, print_result=True):
        self.y_true = np.argmax(y_true, axis=1)
        self.y_pred = y_pred
        if save == True:
            os.mkdir('model-comparison/'+now+'/'+str1)
```

```
            np.savetxt('model-comparison/{}/{}/pred.csv'.format(now,str1), y_pred,
delimiter=',', fmt='%d')

        result = self.classification_report()
        fpr0 = 100 - float(result['precision'][0][0:4])
        line1 = 'Accuracy is: ' + str(result['f1-score']['accuracy'])
        line2 = 'F1 Score is: ' + str(result['f1-score']['weighted avg'])
        line3 = 'Precision of Class 0 is: ' + '{0:.2f}'.format(100-fpr0)+ ' %'
        line4 = '\nClassification Report:'
        line5 = '\nConfusion Matrix:'
        cm = self.confusion_matrix()
        line6 = '\n'
        res_total = line1 + '\n' + line2 + '\n' + line3 + '\n' + line4 + '\n' +
str(result) + '\n' + line5 + '\n' + str(cm) + '\n' + line6
        # write to file
        if save == True:
            with open('model-comparison/{}/{}/report.txt'.format(now,str1), 'w') as
f:
                f.write(res_total)
        if print_result == True:
            print(res_total)

    def accuracy_score(self, y_t, y_p):
        correct = sum(y_t == y_p)
        return correct / len(y_t)

    def scores(self, y_t, y_p, class_label= 1):
        true = y_t == class_label
        pred = y_p == class_label
        tp = sum(true & pred)
        fp = sum(~true & pred)
        fn = sum(true & ~pred)
        tn = sum(~true & ~pred)
        precision = tp / (tp + fp)
        recall = tp / (tp + fn)
        f1 = 2 * (precision * recall) / (precision + recall)
        return precision, recall, f1

    def confusion_matrix(self,labels=None):
        labels = labels if labels else sorted(set(self.y_true) | set(self.y_pred))
        indexes = {v:i for i, v in enumerate(labels)}
        matrix = np.zeros((len(indexes),len(indexes))).astype(int)
        for t, p in zip(self.y_true, self.y_pred):
            matrix[indexes[t], indexes[p]] += 1
        # print('Confusion Matrix: ')
        # print(pd.DataFrame(matrix, index=labels, columns=labels))
        return pd.DataFrame(matrix, index=labels, columns=labels)

    def classification_report(self):
        output_dict = {}
        support_list = []
```

```
        precision_list = []
        recall_list = []
        f1_list = []
        for i in np.unique(self.y_true):
            support = sum(self.y_true == i)
            precision, recall, f1 = self.scores(self.y_true, self.y_pred,
class_label=i)
            output_dict[i] = {'precision':precision, 'recall':recall, 'f1-
score':f1, 'support':support}
            precision_list.append(precision)
            recall_list.append(recall)
            f1_list.append(f1)
            support_list.append(support)
        support = np.sum(support_list)
        output_dict['accuracy'] = {'precision':0, 'recall':0, 'f1-
score':self.accuracy_score(self.y_true, self.y_pred), 'support':support}
        # macro avg
        macro_precision = np.mean(precision_list)
        macro_recall = np.mean(recall_list)
        macro_f1 = np.mean(f1_list)
        output_dict['macro avg'] = {'precision':macro_precision,
'recall':macro_recall, 'f1-score':macro_f1, 'support':support}
        # weighted avg
        weighted_precision = np.average(precision_list, weights=support_list)
        weighted_recall = np.average(recall_list, weights=support_list)
        weighted_f1 = np.average(f1_list, weights=support_list)
        output_dict['weighted avg'] = {'precision':weighted_precision,
'recall':weighted_recall, 'f1-score':weighted_f1, 'support':support}
        # convert to dataframe and format
        report_d = pd.DataFrame(output_dict).T
        annot = report_d.copy()
        annot.iloc[:, 0:3] = (annot.iloc[:, 0:3]*100).applymap('{:.2f}'.format) + '
%'
        annot['support'] = annot['support'].astype(int)
        annot.loc['accuracy','precision'] = ''
        annot.loc['accuracy','recall'] = ''
        return annot

# %%
def GridSearch(model_options, X_train, y_train, X_val, y_val, X_test, y_test,
print_result=False, seed=42, history_steps=1):
    # Grid Search Function
    best_metric = 0
    for i in range(len(model_options)):
        models = model_options[i]
        model_number = i + 1
        now = datetime.datetime.now().strftime("%d-%m-%H-%M")
        # Create folder for current model
        if not os.path.exists('model-comparison/'+now):
            os.mkdir('model-comparison/'+now)
        else:
```

```
            now = now + str('--1')
            os.mkdir('model-comparison/'+now)
        model = LogisticRegression(seed=seed)
        start_time = datetime.datetime.now()
        model.fit(X_train, y_train, X_val, y_val, now, print_result=print_result,
max_epoch=models[0], history_steps=history_steps,
                  weight_init= models[1], batch_size=models[2], lr=models[3],
lr_type=models[4], regularization=models[5])
        end_time = datetime.datetime.now()
        time_elapsed = str(end_time - start_time)[2:7]
        metric = model.validation_accuracy
        model.save_weights()
        model.plot()
        y_pred = model.predict(X_val)
        results = EvaluateModel(y_val, y_pred, 'val', now,
print_result=print_result)
        y_pred = model.predict(X_test)
        results = EvaluateModel(y_test, y_pred, 'test', now,
print_result=print_result)
        if metric > best_metric:
            best_metric = metric
            best_model = now
        print('Model ', str(model_number), ' saved with name: ', now)
        print(models, 'Val-Accuracy:', metric)

        # append to txt file
        lr_print = str(models[3]) + ' ' + models[4]
        model_specs = 'LR | Batch Size: {} | Weight Init: {} | lr: {} |
Regularization: {} | Max Epoch: {}'.format(models[2], models[1], lr_print,
models[5], models[0])
        with open('model-comparison/best-models.txt', 'a') as f:
            f.write(now + ' | ' + model_specs + ' | ' + str(metric) + ' | Time
Elapsed: '+ time_elapsed +'\n')
        print(len(model_options)-model_number, 'models left to train.')
    best_metric = str(best_metric*100)[:5]
    print('Best Model is:', best_model, 'with validation accuracy:', best_metric,
'%')

# %%
def TrainModel(max_epoch, batch_size, weight_init, lr, lr_type, regularization):
    model_options = [[max_epoch, weight_init, batch_size, lr, lr_type,
regularization]]
    return model_options

# %%
# Train New Model
model_parameters = TrainModel(
    max_epoch=1000, batch_size=512, weight_init='zero', lr=0.01, lr_type='momentum:
0.99', regularization='l2: 0.01')
```

```
GridSearch(model_parameters, X_train, y_train, X_val, y_val, X_test, y_test,
print_result=True, seed=42, history_steps=1)

os.system(finish_sound)

# %%
# Grid Search Combinations

max_epoch = [1000]
weight_init = ['zero', 'uniform', 'normal']
batch_size = [512, 5120, 1]
lr = [0.01, 0.005, 0.001]
lr_type = ['momentum: 0.99', 'static', 'adaptive']
regularization = ['l2: 0.01', 'l2: 0.001', 'l2: 0.0001', 'l1: 0.01', 'l1: 0.001',
'l1: 0.0001']
params = [max_epoch, weight_init, batch_size, lr, lr_type, regularization]
model_options = list(product(*params))
print('Number of combinations:', len(model_options))
print('Combination 1:', model_options[0])

# %%
GridSearch(model_options[0:1], X_train, y_train, X_val, y_val, X_test, y_test,
seed=42, history_steps=100)
os.system(finish_sound)

# %%
# # Train new model
# now = datetime.datetime.now().strftime("%d-%m-%H-%M")

# # Fit model
# model = LogisticRegression()
# model.fit(X_train, y_train, X_val, y_val, now, max_epoch=1000,
#           batch_size=5120, weight_init='zero', lr=0.01, lr_type='momentum: 0.99',
regularization='l2: 0.01')
# model.save_weights()
# model.plot()

# # Validation Set Results
# y_pred = model.predict(X_val)
# results = EvaluateModel(y_val, y_pred, 'val', now)

# # Test Set Results
# y_pred = model.predict(X_test)
# results = EvaluateModel(y_test, y_pred, 'test', now)

# # play sound when finished
# os.system(finish_sound)

# %%
# # Load Trained Model and Evaluate
```

```
# #now = open('model-comparison/last.txt', 'r').read()
# now = '20-12-07-12'
# model = LogisticRegression()
# model.load_weights(now)
# model.load_history()

# # Validation Set Results
# model.plot(save=False)
# y_pred = model.predict(X_val)
# results = EvaluateModel(y_val, y_pred, 'val', now, save=False)

# # Test Set Results
# y_pred = model.predict(X_test)
# results = EvaluateModel(y_test, y_pred, 'test', now, save=False)
```

Mehmet Bayık
21802166
20.12.2023

Appendix C – Support Vector Machine

```
# %%
"""
dependencies:
  - python=3.8.17
  - numpy=1.24.0
  - matplotlib=3.7.1
  - pandas=2.0.2
"""
import os
import random
import datetime
from itertools import product

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

finish_sound = "afplay /Users/mehmet/Documents/vs-code/winsquare.mp3"
# play sound when finished
# os.system(finish_sound)

# %%
# Read data from npy file ( already preprocessed )
filename = 'original-numpy'
X_train = np.load(f'dataset/{filename}/X_train_pca_2.npy')
X_val = np.load(f'dataset/{filename}/X_val.npy')
X_test = np.load(f'dataset/{filename}/X_test.npy')
y_train = np.load(f'dataset/{filename}/y_train.npy')
y_val = np.load(f'dataset/{filename}/y_val.npy')
y_test = np.load(f'dataset/{filename}/y_test.npy')

print(X_train.shape, y_train.shape,'\n', X_val.shape, y_val.shape,'\n',
X_test.shape, y_test.shape)

# %%
class SVM:
    def __init__(self, seed=42):
        np.random.seed(seed)
        self.W = None
        self.now = None
        self.print_result = True
        self.history_steps1 = None
        self.history = None
        self.validation_accuracy = None
        self.degree = None
        self.alpha = None
        self.constant = None

    def validation_accuracy(self):
```

```
        return self.validation_accuracy

    def history(self):
        return self.history

    def load_history(self):
        pd_hist = pd.read_csv(f'model-comparison/{self.now}/history.csv')
        self.history = np.array(pd_hist.iloc[:,1:])

    def plot(self, save = True):
        # Save history as csv file
        history_local = self.history
        if type(history_local) is not pd.DataFrame:
            history_df = pd.DataFrame(history_local)
        if save == True:
            hist_csv_file = f'model-comparison/{self.now}/history.csv'
            with open(hist_csv_file, mode='w') as f:
                history_df.to_csv(f)
        # Plot Loss and Accuracy History as Subplots
        fig, ax = plt.subplots(1, 2)
        fig.set_size_inches(10, 2)
        index = np.arange(1,self.history.shape[1]+1)*self.history_steps1

        ax[0].plot(index, self.history[0], label='Training Loss')
        ax[0].plot(index, self.history[2], label='Validation Loss')
        ax[0].set_title('Loss History')
        ax[0].set_xlabel('Epoch')
        ax[0].set_ylabel('Loss')
        ax[0].legend()
        # find best validation accuracy and its epoch
        best_val_acc = np.max(self.history[3])
        best_val_acc_epoch = (np.argmax(self.history[3]) + 1)*self.history_steps1
        label='Best Epoch = '+str(best_val_acc_epoch)+'\nVal. Acc. = 
'+str((best_val_acc*100).round(2))+ '%'
        ax[1].plot(index, self.history[1], label='Training Accuracy')
        ax[1].plot(index, self.history[3], label='Validation Accuracy')
        ax[1].plot(best_val_acc_epoch, best_val_acc, 'ro', label=label)
        ax[1].set_title('Accuracy History')
        ax[1].set_xlabel('Epoch')
        ax[1].set_ylabel('Accuracy')
        ax[1].legend()
        if save is True and self.now is not None:
            plt.savefig(f'model-comparison/{self.now}/plot.png')
        if self.print_result == True:
            plt.show()
        else:
            plt.close(fig)


    def loss(self, X_nonbiased, y, W):
        # Hinge loss
```

```
        if X_nonbiased.shape[1] != W.shape[0]:
            ones=np.ones(X_nonbiased.shape[0])
            X=np.c_[ones,X_nonbiased]
        else:
            X = X_nonbiased
        scores = X.dot(W)
        num_samples = X.shape[0]
        correct_class_mask = (np.arange(num_samples), y)
        margins = np.maximum(0, scores - scores[correct_class_mask][:, np.newaxis]
+ 1)
        margins[correct_class_mask] = 0
        loss = np.sum(margins)

        return loss

    def gradient(self, X, y, W):

        # Linear kernel
        scores = X.dot(W)
        num_samples = X.shape[0]
        correct_class_mask = (np.arange(num_samples), y)
        margins = np.maximum(0, scores - scores[correct_class_mask][:, np.newaxis]
+ 1)
        margins[correct_class_mask] = 0
        grad_mask = (margins > 0).astype(float)
        grad_mask[correct_class_mask] = -np.sum(grad_mask, axis=1)

        # # Polynomial kernel
        # scores = np.power((self.alpha * X.dot(X.T) + self.constant), self.degree)
        # num_samples = X.shape[0]
        # correct_class_mask = (np.arange(num_samples), y)
        # margins = np.maximum(0, scores - scores[correct_class_mask][:,
np.newaxis] + 1)
        # margins[correct_class_mask] = 0
        # grad_mask = (margins > 0).astype(float)
        # grad_mask[correct_class_mask] = -np.sum(grad_mask, axis=1)

        return X.T.dot(grad_mask)

    def fit(self, X_nonbiased, y, X_val, y_val, now=None, print_result = True,
            batch_size=5120, weight_init='zero', lr=0.01, lr_type = 'static',
lmbda=0.01, max_epoch=1000,
            degree = 3, alpha = 1, constant = 1,
            history_steps = 1, print_step = 100):
        self.degree = degree
        self.alpha = alpha
        self.constant = constant
        start_time = datetime.datetime.now()
        # if there isn't model-comparison folder, create it
        if not os.path.exists('model-comparison'):
            os.mkdir('model-comparison')
```

```
        self.print_result = print_result
        if now is not None:
            self.now = now
        # Create folder for current model
            if not os.path.exists('model-comparison/'+now):
                os.mkdir('model-comparison/'+now)

        self.history_steps1 = history_steps
        self.history = np.zeros((4,max_epoch//history_steps))
        y_onehot = y
        lr_print = str(lr) + ' ' + lr_type
        model_specs = 'SVM | Batch Size: {} | Weight Init. {} | lr: {} | Lambda: {} 
| Max Epoch: {} |'.format(batch_size, weight_init, lr_print, lmbda, max_epoch)

        # add bias
        ones=np.ones(X_nonbiased.shape[0])
        X=np.c_[ones,X_nonbiased]


        #self.W = np.random.rand(num_features, num_classes)

        # zero initialization
        # bias included in W

        self.W = np.zeros((X.shape[1], y.shape[1]))
        # For Poly Kernel
        # self.W = np.zeros((num_features, 512))

        # One hot encoded to not one hot encoded
        y = np.argmax(y, axis=1)
        y_val = np.argmax(y_val, axis=1)

        # Print loss and accuracy every 100 iterations or every max_iter//10 
iterations if max_iter >= 1000
        # if max_epoch >= 1000:
        #     print_step = max_epoch // 10

        # Gradient Descent
        for epoch in range(1,max_epoch+1):

            # Shuffle all data X and y in the same order every epoch
            shuffle_index = np.arange(X.shape[0])
            np.random.shuffle(shuffle_index)
            X = X[shuffle_index]
            y = y[shuffle_index]

            for iteration in range(X.shape[0]//batch_size):

                X_batch = X[batch_size*iteration:batch_size*(iteration+1)]
                y_batch = y[batch_size*iteration:batch_size*(iteration+1)]
```

```
                reg_term = lmbda * self.W
                reg_term[0] = 0


                dRSS = self.gradient(X_batch, y_batch, self.W)
                gradient = (dRSS/batch_size) + reg_term

                if lr_type[0:8] == 'momentum':
                    if epoch == 1:
                        last_gradient = gradient
                    else:
                        momentum = float(lr_type[10:])
                        gradient = gradient + momentum * last_gradient
                        last_gradient = gradient


                self.W -= lr * gradient

            if lr_type[0:8] == 'adaptive' and epoch % 3000 == 0:
                    k = float(lr_type[9:])
                    lr *= k
                    if print_result == True:
                        print('Learning rate changed to: ', lr)

            # For each 100 epochs print losses and accuracy
            if epoch % history_steps == 0:
                # how to calculate accuracy
                loss = self.loss(X, y, self.W)
                val_loss = self.loss(X_val, y_val, self.W)
                accuracy = np.mean(self.predict(X) == y)
                val_acc = np.mean(self.predict(X_val) == y_val)
                self.validation_accuracy = val_acc
                self.history[:,(epoch//history_steps)-1] = np.array([loss,
accuracy, val_loss, val_acc])

                if epoch % print_step == 0:
                    line1 = 'Epoch: ' + str(epoch)
                    line2 = ' | Loss: ' + str(loss)[:5] + ' | Accuracy: ' +
str(accuracy)[0:5]
                    line3 = ' | Val. Loss: ' + str(val_loss)[:5] + ' | Val. Acc: '
+ str(val_acc)[0:5]
                    # line2 = ' | Loss: ' + str(round(loss)) + ' | Accuracy: ' +
str(accuracy)[0:5]
                    # line3 = ' | Val. Loss: ' + str(round(val_loss)) + ' | Val.
Acc: ' + str(val_acc)[0:5]
                    if print_result == True:
                        print(line1 + line2 + line3)
                    if now is not None:
                        with open('model-comparison/{}/log.txt'.format(now), 'a')
as f:
                            f.write(line1 + line2 + line3 + '\n')
```

```
            if epoch == max_epoch:
                end_time = datetime.datetime.now()
                if print_result == True:
                    print('Training finished. Time elapsed:', end_time -
start_time, '\n')
                    print('Accuracy: ', str(accuracy)[0:5], 'Val. Accuracy: ',
str(val_acc)[0:5])
                val_acc_print = str(val_acc*100)+ '00'
                if now is not None:
                    with open('model-comparison/{}/log.txt'.format(now), 'a') as f:
                        write_line = 'Training finished. Time elapsed: ' +
str(end_time - start_time) + '\n'
                        f.write(write_line)
                    with open('model-comparison/{}/{}-val-
acc.txt'.format(now,val_acc_print[0:5]), 'w') as f:
                        f.write(model_specs)
                    with open('model-comparison/last.txt', 'w') as f:
                        f.write(str(now))

    def predict(self, X_nonbiased):
        if X_nonbiased.shape[1] != self.W.shape[0]:
            # add bias
            ones=np.ones(X_nonbiased.shape[0])
            X=np.c_[ones,X_nonbiased]
        else:
            X = X_nonbiased

        scores = X.dot(self.W)
        predictions = np.argmax(scores, axis=1)
        return predictions

    def save_weights(self):
        # save history steps
        with open('model-comparison/{}/history_steps.txt'.format(self.now), 'w') as
f:
            f.write(str(self.history_steps1))
        # save weights (bias included in W)
        filename = 'model-comparison/{}/weights.npy'.format(self.now)
        np.save(filename, self.W)
    def load_weights(self, now):
        # load history steps
        with open('model-comparison/{}/history_steps.txt'.format(now), 'r') as f:
            self.history_steps1 = int(f.read())
        # load weights (bias included in W)
        filename = 'model-comparison/{}/weights.npy'.format(now)
        self.W = np.load(filename)
        self.now = now

# %%
class EvaluateModel():
```

```
    # Class to evaluate model performance, similar to sklearn.metrics
ClassificationReport and ConfusionMatrix
    def __init__(self, y_true, y_pred, str1, now, save=True, print_result=True):
        self.y_true = np.argmax(y_true, axis=1)
        self.y_pred = y_pred
        if save == True:
            os.mkdir('model-comparison/'+now+'/'+str1)
            np.savetxt('model-comparison/{}/{}/pred.csv'.format(now,str1), y_pred,
delimiter=',', fmt='%d')

        result = self.classification_report()
        fpr0 = 100 - float(result['precision'][0][0:4])
        line1 = 'Accuracy is: ' + str(result['f1-score']['accuracy'])
        line2 = 'F1 Score is: ' + str(result['f1-score']['weighted avg'])
        line3 = 'Precision of Class 0 is: ' + '{0:.2f}'.format(100-fpr0)+ ' %'
        line4 = '\nClassification Report:'
        line5 = '\nConfusion Matrix:'
        cm = self.confusion_matrix()
        line6 = '\n'
        res_total = line1 + '\n' + line2 + '\n' + line3 + '\n' + line4 + '\n' +
str(result) + '\n' + line5 + '\n' + str(cm) + '\n' + line6
        # write to file
        if save == True:
            with open('model-comparison/{}/{}/report.txt'.format(now,str1), 'w') as
f:
                f.write(res_total)
        if print_result == True:
            print(res_total)

    def accuracy_score(self, y_t, y_p):
        correct = sum(y_t == y_p)
        return correct / len(y_t)

    def scores(self, y_t, y_p, class_label= 1):
        true = y_t == class_label
        pred = y_p == class_label
        tp = sum(true & pred)
        fp = sum(~true & pred)
        fn = sum(true & ~pred)
        tn = sum(~true & ~pred)
        precision = tp / (tp + fp)
        recall = tp / (tp + fn)
        f1 = 2 * (precision * recall) / (precision + recall)
        return precision, recall, f1

    def confusion_matrix(self,labels=None):
        labels = labels if labels else sorted(set(self.y_true) | set(self.y_pred))
        indexes = {v:i for i, v in enumerate(labels)}
        matrix = np.zeros((len(indexes),len(indexes))).astype(int)
        for t, p in zip(self.y_true, self.y_pred):
            matrix[indexes[t], indexes[p]] += 1
```

```
        # print('Confusion Matrix: ')
        # print(pd.DataFrame(matrix, index=labels, columns=labels))
        return pd.DataFrame(matrix, index=labels, columns=labels)

    def classification_report(self):
        output_dict = {}
        support_list = []
        precision_list = []
        recall_list = []
        f1_list = []
        for i in np.unique(self.y_true):
            support = sum(self.y_true == i)
            precision, recall, f1 = self.scores(self.y_true, self.y_pred,
class_label=i)
            output_dict[i] = {'precision':precision, 'recall':recall, 'f1-
score':f1, 'support':support}
            precision_list.append(precision)
            recall_list.append(recall)
            f1_list.append(f1)
            support_list.append(support)
        support = np.sum(support_list)
        output_dict['accuracy'] = {'precision':0, 'recall':0, 'f1-
score':self.accuracy_score(self.y_true, self.y_pred), 'support':support}
        # macro avg
        macro_precision = np.mean(precision_list)
        macro_recall = np.mean(recall_list)
        macro_f1 = np.mean(f1_list)
        output_dict['macro avg'] = {'precision':macro_precision,
'recall':macro_recall, 'f1-score':macro_f1, 'support':support}
        # weighted avg
        weighted_precision = np.average(precision_list, weights=support_list)
        weighted_recall = np.average(recall_list, weights=support_list)
        weighted_f1 = np.average(f1_list, weights=support_list)
        output_dict['weighted avg'] = {'precision':weighted_precision,
'recall':weighted_recall, 'f1-score':weighted_f1, 'support':support}
        # convert to dataframe and format
        report_d = pd.DataFrame(output_dict).T
        annot = report_d.copy()
        annot.iloc[:, 0:3] = (annot.iloc[:, 0:3]*100).applymap('{:.2f}'.format) + '
%'
        annot['support'] = annot['support'].astype(int)
        annot.loc['accuracy','precision'] = ''
        annot.loc['accuracy','recall'] = ''
        return annot

# %%
def GridSearch(model_options, X_train, y_train, X_val, y_val, X_test, y_test,
print_result=False, seed=42, history_steps=100):
    # Grid Search Function
    best_metric = 0
    for i in range(len(model_options)):
```

```
        models = model_options[i]
        model_number = i + 1
        now = datetime.datetime.now().strftime("%d-%m-%H-%M")
        # Create folder for current model
        if not os.path.exists('model-comparison/'+now):
            os.mkdir('model-comparison/'+now)
        else:
            now = now + str('--1')
            os.mkdir('model-comparison/'+now)
        model = SVM(seed=seed)
        start_time = datetime.datetime.now()
        model.fit(X_train, y_train, X_val, y_val, now, print_result=print_result,
max_epoch=models[0], history_steps=history_steps,
                 weight_init= models[1], batch_size=models[2], lr=models[3],
lr_type=models[4], lmbda=models[5])
        end_time = datetime.datetime.now()
        time_elapsed = str(end_time - start_time)[2:7]
        metric = model.validation_accuracy
        model.save_weights()
        model.plot()
        y_pred = model.predict(X_val)
        results = EvaluateModel(y_val, y_pred, 'val', now,
print_result=print_result)
        y_pred = model.predict(X_test)
        results = EvaluateModel(y_test, y_pred, 'test', now,
print_result=print_result)
        if metric > best_metric:
            best_metric = metric
            best_model = now
        print('Model ', str(model_number), ' saved with name: ', now)
        print(models, 'Val-Accuracy:', metric)

        # append to txt file
        lr_print = str(models[3]) + ' ' + models[4]
        model_specs = 'SVM | Batch Size: {} | Weight Init: {} | lr: {} | Lambda: {}
| Max Epoch: {}'.format(models[2], models[1], lr_print, models[5], models[0])
        with open('model-comparison/best-models.txt', 'a') as f:
            f.write(now + ' | ' + model_specs + ' | ' + str(metric) + ' | Time
Elapsed: '+ time_elapsed +'\n')
        print(len(model_options)-model_number, 'models left to train.')
    best_metric = str(best_metric*100)[:5]
    print('Best Model is:', best_model, 'with validation accuracy:', best_metric,
'%')

# %%
def TrainModel(max_epoch, batch_size, weight_init, lr, lr_type, lmbda):
    model_options = [[max_epoch, weight_init, batch_size, lr, lr_type, lmbda]]
    return model_options

# %%
# Train New Model
```

```
model_parameters = TrainModel(
    max_epoch=1000, batch_size=512, weight_init='zero', lr=0.001, lr_type='static',
lmbda=0.1)

GridSearch(model_parameters, X_train, y_train, X_val, y_val, X_test, y_test,
print_result=True, seed=42, history_steps=1)

os.system(finish_sound)

# %%
# Grid Search Combinations

max_epoch = [2500]
weight_init = ['zero']
#batch_size = [1, 512, 5120]
batch_size = [512, 5120]
lr = [0.1, 0.01, 0.001, 0.0001, 0.00001, 0.000001]
#lr = [0.01, 0.001, 0.0001]

lr_type = ['static']
regularization = [0.1, 0.01, 0.001, 0.0001, 0.00001, 0.000001, 0]
#regularization = [0.01, 0.005, 0.001, 0.0005, 0.0001]
params = [max_epoch, weight_init, batch_size, lr, lr_type, regularization]
model_options = list(product(*params))
print('Number of combinations:', len(model_options))
print('Combination 1:', model_options[0])

# %%
# Grid Search All Combinations

GridSearch(model_options[0:1], X_train, y_train, X_val, y_val, X_test, y_test,
seed=42, history_steps=100)
os.system(finish_sound)

# %%
""" # Train New Model
now = datetime.datetime.now().strftime("%d-%m-%H-%M")

model = SVM(seed=42)
model.fit(X_train, y_train, X_val, y_val, now=now, print_result=True,
        lr=0.001, lmbda=0.001, max_epoch=1000)
model.save_weights()
model.plot()

# Validation Set Results
y_pred = model.predict(X_val)
results = EvaluateModel(y_val, y_pred, 'val', now)
# Test Set Results
y_pred = model.predict(X_test)
results = EvaluateModel(y_test, y_pred, 'test', now)
"""
```

Mehmet Bayık
21802166
20.12.2023

Appendix D – Neural Networks

```
# %%
"""
dependencies:
  - python=3.8.17
  - numpy=1.24.0
  - matplotlib=3.7.1
  - pandas=2.0.2
"""
import os
import random
import datetime
from itertools import product

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

finish_sound = "afplay /Users/mehmet/Documents/vs-code/winsquare.mp3"
# play sound when finished
# os.system(finish_sound)

# %%
# Read data from npy file ( already preprocessed )
filename = 'original-numpy'
X_train = np.load(f'dataset/{filename}/X_train_pca_2.npy')
X_val = np.load(f'dataset/{filename}/X_val.npy')
X_test = np.load(f'dataset/{filename}/X_test.npy')
y_train = np.load(f'dataset/{filename}/y_train.npy')
y_val = np.load(f'dataset/{filename}/y_val.npy')
y_test = np.load(f'dataset/{filename}/y_test.npy')

# # Push all X to positive side
# X_train = X_train + np.abs(np.min(X_train))
# X_val = X_val + np.abs(np.min(X_val))
# X_test = X_test + np.abs(np.min(X_test))

# # Remove one hot encoding from y
# y_train = np.argmax(y_train, axis=1)
# y_val = np.argmax(y_val, axis=1)
# y_test = np.argmax(y_test, axis=1)

# X_train = X_train - np.min(X_train, axis=0) + 1e-3
# #X_train = X_train / np.max(X_train, axis=0)
# X_val = X_val - np.min(X_val, axis=0) + 1e-3
# #X_val = X_val / np.max(X_val, axis=0)
# X_test = X_test - np.min(X_test, axis=0) + 1e-3
# #X_test = X_test / np.max(X_test, axis=0)
```

```
print(X_train.shape, y_train.shape,'\n', X_val.shape, y_val.shape,'\n',
X_test.shape, y_test.shape)

# %%
class NN:
    def __init__(self, seed=42):
        np.random.seed(seed)
        self.n_features = None
        self.n_classes = None

        self.input_layer = None
        self.layers = []
        self.output_layer = None
        self.Weights = []
        self.Biases = []

        self.now = None
        self.print_result = True
        self.history_steps1 = None
        self.history = None
        self.validation_accuracy = None

    def validation_accuracy(self):
        return self.validation_accuracy

    def history(self):
        return self.history

    def load_history(self):
        pd_hist = pd.read_csv(f'model-comparison/{self.now}/history.csv')
        self.history = np.array(pd_hist.iloc[:,1:])

    def loss(self, X, y):
        # Cross Entropy Loss
        pred = self.Forward(X)[-1]
        return -np.mean(np.sum(y * np.log(pred), axis=1))

    def plot(self, save = True):
        # Save history as csv file
        history_local = self.history
        if type(history_local) is not pd.DataFrame:
            history_df = pd.DataFrame(history_local)
        if save == True:
            hist_csv_file = f'model-comparison/{self.now}/history.csv'
            with open(hist_csv_file, mode='w') as f:
                history_df.to_csv(f)
        # Plot Loss and Accuracy History as Subplots
        fig, ax = plt.subplots(1, 2)
        fig.set_size_inches(10, 2)
        index = np.arange(1,self.history.shape[1]+1)*self.history_steps1
        ax[0].plot(index, self.history[0], label='Training Loss')
```

```
        ax[0].plot(index, self.history[2], label='Validation Loss')
        ax[0].set_title('Loss History')
        ax[0].set_xlabel('Epoch')
        ax[0].set_ylabel('Loss')
        ax[0].legend()
        # find best validation accuracy and its epoch
        best_val_acc = np.max(self.history[3])
        best_val_acc_epoch = (np.argmax(self.history[3]) + 1)*self.history_steps1
        label='Best Epoch = '+str(best_val_acc_epoch)+'\nVal. Acc. =
'+str((best_val_acc*100).round(2))+ '%'
        ax[1].plot(index, self.history[1], label='Training Accuracy')
        ax[1].plot(index, self.history[3], label='Validation Accuracy')
        ax[1].plot(best_val_acc_epoch, best_val_acc, 'ro', label=label)
        ax[1].set_title('Accuracy History')
        ax[1].set_xlabel('Epoch')
        ax[1].set_ylabel('Accuracy')
        ax[1].legend()
        if save is True and self.now is not None:
            plt.savefig(f'model-comparison/{self.now}/plot.png')
        if self.print_result == True:
            plt.show()
        else:
            plt.close(fig)

    def add_input_layer(self):
        self.input_layer = True

    def add_hidden_layer(self, n_neurons, activation=None):
        self.layers.append((n_neurons, activation))

    def add_output_layer(self, activation='softmax'):
        self.layers.append((activation))

    def Initialize_weights(self, weight_init='zero'):
        for i in range(len(self.layers)):

            # All middle hidden layers
            if i != 0 and i != len(self.layers)-1:
                prev_n_neurons = self.layers[i-1][0]
                n_neurons = self.layers[i][0]
            # First Hidden layer
            elif i == 0:
                prev_n_neurons = self.n_features
                n_neurons = self.layers[i][0]
            # Output layer
            elif i == len(self.layers)-1:
                prev_n_neurons = self.layers[i-1][0]
                n_neurons = self.n_classes

            # Initialize weights and biases for each layer
            if weight_init == 'random':
```

```
                self.Weights.append(np.random.randn(prev_n_neurons, 
n_neurons)*0.01)
                self.Biases.append(np.random.randn(n_neurons)*0.01)
            elif weight_init == 'zero':
                self.Weights.append(np.zeros((prev_n_neurons, n_neurons)))
                self.Biases.append(np.zeros(n_neurons))
            elif weight_init == 'he-normal':
                # He Normal Initialization
                self.Weights.append(np.random.randn(prev_n_neurons, 
n_neurons)*np.sqrt(2/prev_n_neurons))

self.Biases.append(np.random.randn(n_neurons)*np.sqrt(2/prev_n_neurons))
            elif weight_init == 'xavier-normal':
                # Xavier Normal Initialization
                self.Weights.append(np.random.randn(prev_n_neurons, 
n_neurons)*np.sqrt(1/prev_n_neurons))

self.Biases.append(np.random.randn(n_neurons)*np.sqrt(1/prev_n_neurons))
            #print("Weights shape:", self.Weights[i].shape)
            #print ("Biases shape:", self.Biases[i].shape)

    def Activation(self, output, activation=None, derivative=False):
        if activation == 'relu':
            if derivative:
                return np.where(output > 0, 1, 0)
            else:
                return np.maximum(0, output)
        elif activation == 'leaky-relu':
            if derivative:
                return np.where(output > 0, 1, 0.01)
            else:
                return np.where(output > 0, output, 0.01 * output)

        elif activation == 'sigmoid':
            if derivative:
                sigmoid_output = self.Activation(output, activation='sigmoid')
                return sigmoid_output * (1 - sigmoid_output)
            else:
                return 1 / (1 + np.exp(-output))
        elif activation == 'softmax':
            if derivative:
                # The derivative of softmax is a bit involved and requires the 
Jacobian matrix
                # For simplicity, we can assume softmax is only used in the output 
layer
                # and compute its derivative accordingly
                softmax_output = self.Activation(output, activation='softmax')
                return softmax_output * (1 - softmax_output)
            else:
                exp_output = np.exp(output)
                return exp_output / np.sum(exp_output, axis=1, keepdims=True)
```

```
    def Forward(self, X):
        # Forward pass for each layer
        input = X
        outputs = []
        for layer_num in range(len(self.layers)):
                layer = self.layers[layer_num]
                if layer_num == len(self.layers) - 1:
                    activation = layer
                else:
                    activation = layer[1]
                W = self.Weights[layer_num]
                b = self.Biases[layer_num]
                output = self.Activation(np.dot(input, W) + b, activation)
                outputs.append(output)
                # Next layer's input is this layer's output
                input = output
        return outputs

    def Backward(self, X, y, outputs):
        m = X.shape[0]  # Number of training examples
        # Initialize gradients
        dW = [0] * len(self.Weights)
        db = [0] * len(self.Biases)

        # Backward pass for all layers
        for layer_num in reversed(range(len(self.layers))):
            if layer_num == 0:
                # First hidden layer
                input_layer_backward = X
            else:
                input_layer_backward = outputs[layer_num-1]
            output_layer_backward = outputs[layer_num]
            layer = self.layers[layer_num]
            if layer_num == len(self.layers) - 1:
                activation = layer[0]
                # Last layer Compute gradients
                dZ = output_layer_backward - y
                dW[layer_num] = np.dot(input_layer_backward.T, dZ) / m
                db[layer_num] = np.sum(dZ, axis=0) / m
            else:
                activation = layer[1]
                # Hidden layers Compute gradients
                dZ = np.dot(dZ, self.Weights[layer_num+1].T) *
self.Activation(output_layer_backward, activation, derivative=True)
                dW[layer_num] = np.dot(input_layer_backward.T, dZ) / m
                db[layer_num] = np.sum(dZ, axis=0) / m
            #print(f'Layer {layer_num} dW shape: {dW[layer_num].shape} db shape:
{db[layer_num].shape}')
```

```
        return dW, db


    def fit(self, X, y, X_val, y_val, now=None, max_epoch = 100, print_result=True,
save=False,
            batch_size=5120, weight_init='zero', lr=0.01, lr_type = 'static',
regularization='l2: 0.01',
            history_steps = 10, print_step = 50):

        start_time = datetime.datetime.now()
        # if there isn't model-comparison folder, create it
        if not os.path.exists('model-comparison'):
            os.mkdir('model-comparison')
        self.print_result = print_result
        if now is not None:
            self.now = now
        # Create folder for current model
            if not os.path.exists('model-comparison/'+now):
                os.mkdir('model-comparison/'+now)

        self.history_steps1 = history_steps
        self.history = np.zeros((4,max_epoch//history_steps))

        if regularization[0:2] == 'l2':
            # L2 regularization
            lmbda = float(regularization[4:])
        else:
            lmbda = 0

        lr_print = str(lr) + ' ' + lr_type
        model_specs = 'NN | Hidden Layers: {} |Batch Size: {} | Weight Init. {} |
lr: {} | Lambda: {} | Max Epoch: {} |'.format(str(self.layers), batch_size,
weight_init, lr_print, lmbda, max_epoch)

        self.n_features = X.shape[1]
        # y is one hot encoded
        self.n_classes = y.shape[1]

        self.Initialize_weights(weight_init=weight_init)

        old_val_acc = 0
        for epoch in range(1,max_epoch+1):

            # Shuffle all data X and y in the same order every epoch
            shuffle_index = np.arange(X.shape[0])
            np.random.shuffle(shuffle_index)
            X = X[shuffle_index]
            y = y[shuffle_index]

            for iteration in range(X.shape[0]//batch_size):
```

```
                X_batch = X[batch_size*iteration:batch_size*(iteration+1)]
                y_batch = y[batch_size*iteration:batch_size*(iteration+1)]

                # For all layers, Forward pass one time
                outputs = self.Forward(X_batch)
                # For all layers, Backward pass one time
                dW, db = self.Backward(X_batch, y_batch, outputs)

                # Update weights and biases
                for layer_num in range(len(self.layers)):
                    self.Weights[layer_num] -= lr * dW[layer_num] + 2 * lmbda *
self.Weights[layer_num]
                    self.Biases[layer_num] -= lr * db[layer_num]


            if epoch % history_steps == 0:
                # how to calculate accuracy
                t_loss = self.loss(X, y)
                val_loss = self.loss(X_val, y_val)
                # Compute accuracy
                pred = self.Forward(X)[-1]
                accuracy = np.mean(np.argmax(pred, axis=1) == np.argmax(y, axis=1))
                # Validation accuracy
                pred_val = self.predict(X_val)
                val_acc = np.mean(pred_val == np.argmax(y_val, axis=1))
                self.validation_accuracy = val_acc
                self.history[:,(epoch//history_steps)-1] = np.array([t_loss,
accuracy, val_loss, val_acc])

                if epoch % print_step == 0:
                    line1 = 'Epoch: ' + str(epoch)
                    line2 = ' | Loss: ' + str(t_loss)[:5] + ' | Accuracy: ' +
str(accuracy)[0:5]
                    line3 = ' | Val. Loss: ' + str(val_loss)[:5] + ' | Val. Acc: '
+ str(val_acc)[0:5]
                    #line2 = ' | Accuracy: ' + str(accuracy)[0:5]
                    #line3 = ' | Val. Acc: ' + str(val_acc)[0:5]
                    if print_result == True:
                        print(line1 + line2 + line3)
                    if now is not None:
                        with open('model-comparison/{}/log.txt'.format(now), 'a')
as f:
                            f.write(line1 + line2 + line3 + '\n')

                    if abs(old_val_acc-val_acc) < 0.005:
                        #lr = lr * 0.9
                        #print(f'Learning rate is updated to {lr}')
                        pass

                    old_val_acc = val_acc
```

```
            if epoch == max_epoch:
                end_time = datetime.datetime.now()
                if print_result == True:
                    print('Training finished. Time elapsed:', end_time -
start_time, '\n')
                    print('Accuracy: ', str(accuracy)[0:5], 'Val. Accuracy: ',
str(val_acc)[0:5])
                val_acc_print = str(val_acc*100)+ '00'
                if now is not None:
                    with open('model-comparison/{}/log.txt'.format(now), 'a') as f:
                        write_line = 'Training finished. Time elapsed: ' +
str(end_time - start_time) + '\n'
                        f.write(write_line)
                    with open('model-comparison/{}/{}-val-
acc.txt'.format(now,val_acc_print[0:5]), 'w') as f:
                        f.write(model_specs)
                    with open('model-comparison/last.txt', 'w') as f:
                        f.write(str(now))

    def predict(self, X):
        pred_l = self.Forward(X)
        pred = pred_l[-1]
        return np.argmax(pred, axis=1)

    def save_weights(self):
        # if there isn't model folder, create it
        if not os.path.exists('model-comparison/{}/model'.format(self.now)):
            os.mkdir('model-comparison/{}/model'.format(self.now))
        # save history steps
        with open('model-comparison/{}/model/history_steps.txt'.format(self.now),
'w') as f:
            f.write(str(self.history_steps1))
        # save layers first to txt file
        with open('model-comparison/{}/model/layers.txt'.format(self.now), 'w') as
f:
            f.write(str(self.layers))
        # save weights and biases
        for i in range(len(self.layers)):
            filename = 'model-comparison/{}/model/weights{}.npy'.format(self.now,
i+1)
            np.save(filename, self.Weights[i])
            filename2 = 'model-comparison/{}/model/biases{}.npy'.format(self.now,
i+1)
            np.save(filename2, self.Biases[i])

    def load_weights(self, now):
        # load history steps
        with open('model-comparison/{}/model/history_steps.txt'.format(now), 'r')
as f:
            self.history_steps1 = int(f.read())
```

```
        # load layers from txt file
        with open('model-comparison/{}/model/layers.txt'.format(now), 'r') as f:
            self.layers = list(eval(f.read()))
        # load weights and biases
        for i in range(len(self.layers)):
            filename = 'model-comparison/{}/model/weights{}.npy'.format(now, i+1)
            self.Weights.append(np.load(filename))
            filename2 = 'model-comparison/{}/model/biases{}.npy'.format(now, i+1)
            self.Biases.append(np.load(filename2))
        self.now = now


# %%
class EvaluateModel():
    # Class to evaluate model performance, similar to sklearn.metrics
ClassificationReport and ConfusionMatrix
    def __init__(self, y_true, y_pred, str1, now, save=True, print_result=True):
        self.y_true = np.argmax(y_true, axis=1)
        self.y_pred = y_pred
        if save == True:
            os.mkdir('model-comparison/'+now+'/'+str1)
            np.savetxt('model-comparison/{}/{}/pred.csv'.format(now,str1), y_pred,
delimiter=',', fmt='%d')

        result = self.classification_report()
        fpr0 = 100 - float(result['precision'][0][0:4])
        line1 = 'Accuracy is: ' + str(result['f1-score']['accuracy'])
        line2 = 'F1 Score is: ' + str(result['f1-score']['weighted avg'])
        line3 = 'Precision of Class 0 is: ' + '{0:.2f}'.format(100-fpr0)+ ' %'
        line4 = '\nClassification Report:'
        line5 = '\nConfusion Matrix:'
        cm = self.confusion_matrix()
        line6 = '\n'
        res_total = line1 + '\n' + line2 + '\n' + line3 + '\n' + line4 + '\n' +
str(result) + '\n' + line5 + '\n' + str(cm) + '\n' + line6
        # write to file
        if save == True:
            with open('model-comparison/{}/{}/report.txt'.format(now,str1), 'w') as
f:
                f.write(res_total)
        if print_result == True:
            print(res_total)

    def accuracy_score(self, y_t, y_p):
        correct = sum(y_t == y_p)
        return correct / len(y_t)

    def scores(self, y_t, y_p, class_label= 1):
        true = y_t == class_label
        pred = y_p == class_label
        tp = sum(true & pred)
```

```
        fp = sum(~true & pred)
        fn = sum(true & ~pred)
        tn = sum(~true & ~pred)
        precision = tp / (tp + fp)
        recall = tp / (tp + fn)
        f1 = 2 * (precision * recall) / (precision + recall)
        return precision, recall, f1

    def confusion_matrix(self,labels=None):
        labels = labels if labels else sorted(set(self.y_true) | set(self.y_pred))
        indexes = {v:i for i, v in enumerate(labels)}
        matrix = np.zeros((len(indexes),len(indexes))).astype(int)
        for t, p in zip(self.y_true, self.y_pred):
            matrix[indexes[t], indexes[p]] += 1
        # print('Confusion Matrix: ')
        # print(pd.DataFrame(matrix, index=labels, columns=labels))
        return pd.DataFrame(matrix, index=labels, columns=labels)

    def classification_report(self):
        output_dict = {}
        support_list = []
        precision_list = []
        recall_list = []
        f1_list = []
        for i in np.unique(self.y_true):
            support = sum(self.y_true == i)
            precision, recall, f1 = self.scores(self.y_true, self.y_pred,
class_label=i)
            output_dict[i] = {'precision':precision, 'recall':recall, 'f1-
score':f1, 'support':support}
            precision_list.append(precision)
            recall_list.append(recall)
            f1_list.append(f1)
            support_list.append(support)
        support = np.sum(support_list)
        output_dict['accuracy'] = {'precision':0, 'recall':0, 'f1-
score':self.accuracy_score(self.y_true, self.y_pred), 'support':support}
        # macro avg
        macro_precision = np.mean(precision_list)
        macro_recall = np.mean(recall_list)
        macro_f1 = np.mean(f1_list)
        output_dict['macro avg'] = {'precision':macro_precision,
'recall':macro_recall, 'f1-score':macro_f1, 'support':support}
        # weighted avg
        weighted_precision = np.average(precision_list, weights=support_list)
        weighted_recall = np.average(recall_list, weights=support_list)
        weighted_f1 = np.average(f1_list, weights=support_list)
        output_dict['weighted avg'] = {'precision':weighted_precision,
'recall':weighted_recall, 'f1-score':weighted_f1, 'support':support}
        # convert to dataframe and format
        report_d = pd.DataFrame(output_dict).T
```

```
        annot = report_d.copy()
        annot.iloc[:, 0:3] = (annot.iloc[:, 0:3]*100).applymap('{:.2f}'.format) + '
%'
        annot['support'] = annot['support'].astype(int)
        annot.loc['accuracy','precision'] = ''
        annot.loc['accuracy','recall'] = ''
        return annot

# %%
def GridSearch(model_options, X_train, y_train, X_val, y_val, X_test, y_test,
print_result=False, seed=42, history_steps=10):
    # Grid Search Function
    best_metric = 0
    for i in range(len(model_options)):
        models = model_options[i]
        model_number = i + 1
        now = datetime.datetime.now().strftime("%d-%m-%H-%M")
        # Create folder for current model
        if not os.path.exists('model-comparison/'+now):
            os.mkdir('model-comparison/'+now)
        else:
            now = now + str('--1')
            os.mkdir('model-comparison/'+now)
        model = NN(seed=seed)
        start_time = datetime.datetime.now()
        model.add_input_layer() # 10859
        hidden_layers = len(models[-1])
        for i in range(hidden_layers):
            model.add_hidden_layer(models[-1][i][0], activation=models[-1][i][1])
        model.add_output_layer()
        model.fit(X_train, y_train, X_val, y_val, now, print_result=print_result,
max_epoch=models[0], save=True, history_steps=history_steps,
                  weight_init= models[1], batch_size=models[2], lr=models[3],
lr_type=models[4], regularization=models[5])
        end_time = datetime.datetime.now()
        time_elapsed = str(end_time - start_time)[2:7]
        metric = model.validation_accuracy
        model.save_weights()
        model.plot()
        y_pred = model.predict(X_val)
        results = EvaluateModel(y_val, y_pred, 'val', now,
print_result=print_result)
        y_pred = model.predict(X_test)
        results = EvaluateModel(y_test, y_pred, 'test', now,
print_result=print_result)
        if metric > best_metric:
            best_metric = metric
            best_model = now
        print('Model ', str(model_number), ' saved with name: ', now)
        print(models, 'Val-Accuracy:', metric)
```

```
        # append to txt file
        lr_print = str(models[3]) + ' ' + models[4]
        model_specs = 'NN | Batch Size: {} | Weight Init: {} | Lr: {} | Reg: {} | 
Max Epoch: {}'.format(models[2], models[1], lr_print, models[5], models[0])
        with open('model-comparison/best-models.txt', 'a') as f:
            f.write(now + ' | ' + model_specs + ' | ' + str(metric) + ' | Time 
Elapsed: '+ time_elapsed + ' | Hidden Layers: '+ str(models[-1]) +'\n')
        print(len(model_options)-model_number, 'models left to train.')
    best_metric = str(best_metric*100)[:5]
    print('Best Model is:', best_model, 'with validation accuracy:', best_metric, 
'%')

# %%
def TrainModel(hidden_layers, max_epoch, batch_size, weight_init, lr, lr_type, 
regularization):
    model_options = [[max_epoch, weight_init, batch_size, lr, lr_type, 
regularization, hidden_layers]]
    return model_options

# %%
# Train New Model
hidden_layers = [(64, 'leaky-relu'),
                 (32, 'leaky-relu')]

model_parameters = TrainModel(hidden_layers,
    max_epoch=1000, batch_size=512, weight_init='he-normal',
    lr=0.01, lr_type='static', regularization='l2: 0.0001')

GridSearch(model_parameters, X_train, y_train, X_val, y_val, X_test, y_test, 
print_result=True, seed=42, history_steps=1)

os.system(finish_sound)

# %%
# Grid Search Combinations
hidden_layers = [[(64, 'leaky-relu'),
                 (32, 'leaky-relu')]]
max_epoch = [1000]
weight_init = ['he-normal']
batch_size = [512, 5120] # [1, 512, 5120]
lr = [0.01] # [0.1, 0.01, 0.001, 0.0001, 0.00001, 0.000001]
lr_type = ['static']
regularization = ['l2: 0.001', 'l2: 0.0001','l2: 0.0001']
params = [max_epoch, weight_init, batch_size, lr, lr_type, regularization, 
hidden_layers]
model_options = list(product(*params))
print('Number of combinations:', len(model_options))
print('Combination 1:', model_options[0])

# %%
# Train All Combinations
```

```
GridSearch(model_options[0:1], X_train, y_train, X_val, y_val, X_test, y_test,
seed=42, history_steps=10)
os.system(finish_sound)

# %%
"""
# Train Model
now = datetime.datetime.now().strftime("%d-%m-%H-%M")
model = NN()
model.add_input_layer() # 10859
model.add_hidden_layer(64, activation='leaky-relu')
model.add_hidden_layer(32, activation='leaky-relu')
model.add_output_layer()
model.fit(X_train, y_train, X_val, y_val, max_epoch=1000, now=now,
print_result=True, save=True,
          batch_size=5120, weight_init='he-normal', lr=0.01, lr_type='static',
regularization='l2: 0',
          history_steps=10, print_step=50)

model.save_weights()
model.plot()

y_pred = model.predict(X_val)
results = EvaluateModel(y_val, y_pred, 'val', now=now, save=True,
print_result=True)

y_pred = model.predict(X_test)
results = EvaluateModel(y_test, y_pred, 'test', now=now, save=True,
print_result=True)
"""

# %%
"""
# Load Model
now = '20-12-05-29'

model = NN()
model.load_weights(now)
model.load_history()
model.plot(save=False)

y_pred = model.predict(X_val)
results = EvaluateModel(y_val, y_pred, 'val', now=now, save=False,
print_result=True)

y_pred = model.predict(X_test)
results = EvaluateModel(y_test, y_pred, 'test', now=now, save=False,
print_result=True)
"""
```