BILKENT UNIVERSITY

Department of Electrical and Electronics Engineering

Bilkent 06800 Ankara Turkey

EEE 485/585 - Statistical Learning and Data Analytics

Final Project Report

Classification of Epileptic Seizure From Electroencephalogram (EEG) Signals Based on Machine Learning Approaches

Ecem Şimşek – Emirhan Koç

21802384 - 22101553

Section: 1

Group: 7

Course Instructor: Süleyman Serdar Kozat

19.12.2022

TABLE OF CONTENTS

1. Introduction and Problem Description

Epilepsy can be defined as a well-known and prevalent chronic neurological disorder that can affect male and female individuals of any race, ethnicity, or age [1]. According to the number of epilepsy patients around the world, it was observed that it mostly affects infants and the elderly. Epilepsy is characterized by the occurrence of epileptic seizures in which the activities in different parts of the brain become abnormal and these abnormalities arise from sudden and extreme discharge of brain nerve cells [2]. These abnormalities can cause the patients to display unusual behavior, and/or experience unusual sensations and loss of consciousness [1]. Hence, it is important to detect seizures in a correct and timely manner in order to control the progression of the disorder, prevent the possibility of self-harm of the patients and enhance their quality of life. As a result, various medical monitoring and detection devices are used in clinical examinations to detect epileptic seizures and these tools include electroencephalogram (EEG), magnetoencephalography (MEG), and functional magnetic resonance imaging (fMRI) [2]. Among all of these devices, EEG is preferred mostly due to its mobile, safe, and cheap nature. In general, EEG is used to evaluate different types of neurological disorders including Alzheimer's disease, tumors, and various sleep disorders [3]. EEG works by the placement of electrodes that are in the form of "small metal discs with thin wires" onto the scalp of the patient [3]. These electrodes are able to detect the very small electrical charges that occur as a result of ongoing brain activities and transform these charges into electronic graphs by amplifying them at the same time [3]. Therefore, EEG as a testing method is frequently used in the detection of epileptic seizures that may occur in different parts of the brain. EEG can detect these seizures by its single or multiple electrodes which correspond to single or multiple channels placed on different parts of the scalp.

In the literature review, it was observed that the detection of epileptic seizures is an important study and research area, and a wide range of articles focus on the evaluation of epileptic seizures using different machine learning (ML) tools and algorithms [4], [5], [6]. Considering the importance of distinguishing between healthy and epileptic brain signals, most of the studies perform binary classification applications although the datasets that they utilize may have multiple classes for determining the type of epileptic seizures according to different metrics [6], [7], [8]. Hence, our term project is aimed to perform a binary classification task that aims to classify between brain signals with and without epileptic seizure as an extension of the state-of-the-art (SOTA) ML studies by utilizing two datasets that are frequently used by research articles in the fields of EEG and epilepsy. As a part of its problem description, our project will consist of three ML algorithms that are namely Decision Tree (DT), K-Nearest Neighbors (kNN), and Multilayer Perceptron (MLP). Our project modules will be designed as generically as possible in order to be applicable to any dataset that consists of feature (X values) and target (Y values) variables that are related to EEG measurements and their labels (healthy/epileptic) respectively. Moreover, one of the aims of our project is to analyze and report the accuracy of the classification results of participants that are epilepsy patients with seizure-free recordings as well as the vice versa results hence exploring the overall accuracy and classification power of our models [2]. This will lead our project to evaluate both the quality of the datasets and our method's performance [2]. From a general perspective, our generic data preprocessing and ML models will enhance and automatize the epileptic seizure detection process performed by checking the EEG recordings that are applied in clinical examinations.

2. Description of Datasets

2.1 The Bonn EEG Time Series Dataset

For the first half of the project, a well-known, high volume, and publicly available dataset under the name of "**The Bonn EEG Time Series Dataset**" is utilized [9], [10]. This dataset is utilized in various SOTA studies that focus on epileptic seizure detection using EEG recordings [6], [11], [12]. This dataset is preferred due to its organized structure as well as the variety and abundance of its epileptic and healthy EEG recordings. In means of its properties, continuous multichannel EEG recordings were collected from a number of patients for this dataset; however, single-channel EEG signals of 23.6 seconds duration were used for investigation after visual inspection for artifacts

resulting from muscle activity or eye movement [2]. The continuous EEG signals were sampled with a frequency of 173.61 Hz, and a bandpass filter with cut-off frequencies at 0.53 Hz and 40 Hz was used to take the most useful frequencies of these signals [2]. After the pre-processing of the raw EEG signals, a dataset of size 500 x 4097 (number of patients x number of samples) was created for the study and for further studies. This dataset is mainly created for clinical and neurological investigation and research purposes [9]. Furthermore, it is used intensively in ML studies on epilepsy and EEG in recent studies [6], [13], [14], [15]. As briefly explained before, this dataset consists of EEG recordings of 500 participants. The dataset consists of 5 sets of conditions that each contain recordings from 100 participants: 2 sets are the EEG recordings obtained from the brain surfaces of the healthy participants with eyes-open and eyes-closed conditions. The other 2 sets are the intracranial EEG recordings (taken from inner regions of the brain) obtained from epilepsy patients placed in both neutral and seizure-stimulating experimental environments during their "seizure-free" condition. The last set also consists of intracranial EEG recordings obtained from epilepsy patients during their "seizure" condition [2]. By pre-processing and analyzing this dataset with the chosen tools and ML algorithms, our project's main aim is to perform a differentiation between EEG signals that contain and do not contain the "epileptic seizure" condition.

Importantly, a better-organized version of this dataset was obtained from Kaggle in the form of a "csv" file [16]. In this file, every 4097 data points were shuffled and divided into 23 groups in accordance with the 23.6 seconds duration for the EEG signals in the original dataset so that now these groups contain 178 data points (4097/23 = 178.13…≈ 178) lasting for 1 second [16]. As a result of this grouping, each data point got transformed to represent the EEG recording value at a different and unique point in time [16]. Moreover, the number of rows in the original dataset increased to 11,500 (23×500 = 11,500) corresponding to lines of information in the dataset. Each line (row) of information contains "178 data points for 1 second" [16] corresponding to different columns. A final column was created to label each row according to 5 categories that correspond to 5 labels. Among these labels, class 1 represents the patients that had an epileptic seizure whereas classes 2, 3, 4, and 5 represent the patients who did not have an epileptic seizure. Hence, the potential of this dataset being utilized for a binary classification problem was observed in accordance with a great number of SOTA studies performing binary classification instead of a 5-category classification. Consequently, we utilized this dataset in the first half of our project to perform binary classification in which class 0 represents the subjects that do not have an epileptic seizure whereas class 1 represents the subjects that have an epileptic seizure. In the figures below, more detailed explanations regarding the initial 5-classes, and the initial dataset that has 5 classes can be observed respectively.

```
Explanations for 5 Classes in the Original Dataset:

Class 1 - Epileptic seizure condition

Class 2 - Neutral experimental environment condition for the seizure-free subject during the recording of the EEG signals

Class 3 - Seizure-stimulating experimental environment condition for the seizure-free subject during the recording of the EEG signals

Class 4 - Eyes closed condition for the seizure-free subject during the recording of the EEG signals

Class 5 - Eyes open condition for the seizure-free subject during the recording of the EEG signals
```

Fig. 1: More detailed explanations for the 5 classes in the dataset

Unnamed	X1	X2	X3	X4	X5	X6	X7	X8	X9	X10
X21.V1.791	135	190	229	223	192	125	55	-9	-33	-38
X15.V1.924	386	382	356	331	320	315	307	272	244	232
X8.V1.1	-32	-39	-47	-37	-32	-36	-57	-73	-85	-94
X16.V1.60	-105	-101	-96	-92	-89	-95	-102	-100	-87	-79
X20.V1.54	-9	-65	-98	-102	-78	-48	-16	0	-21	-59
X14.V1.56	55	28	18	16	16	19	25	40	52	66
X3.V1.191	-55	-9	52	111	135	129	103	72	37	0
X11.V1.273	1	-2	-8	-11	-12	-17	-15	-16	-18	-17
X19.V1.874	-278	-246	-215	-191	-177	-167	-157	-139	-118	-92
X3.V1.491	8	15	13	3	-6	-8	-5	4	25	41
X3.V1.6	-5	15	28	28	9	-29	-41	-19	14	30
X21.V1.724	-167	-230	-280	-315	-338	-369	-405	-392	-298	-140
X7.V1.162	92	49	0	-32	-51	-65	-37	-19	-25	-29
X1.V1.211	15	12	0	-17	-28	-31	-39	-51	-44	-35
X1.V1.615	-24	-15	-5	-1	4	3	6	10	11	7
X22.V1.242	-135	-133	-125	-118	-111	-105	-102	-93	-94	-90
X1.V1.863	39	41	41	42	43	43	46	47	49	50
X9.V1.302	9	4	-5	-10	-22	-30	-33	-43	-41	-40
X7.V1.541	-21	-5	1	7	19	20	13	2	-1	-3

Fig. 2: A portion of the dataset representing the first 20 rows and their corresponding feature columns ranging from X1 to X10

y
4
1
5
5
5
5
4
2
1
4
5
1
3
4
2
3
2
3
4

Fig. 3: A portion of the dataset representing the labels of the first 20 rows

2.2 The Beirut Epileptic EEG Dataset

For the second half of the project, a well-known, high volume, and publicly available dataset under the name of "**The Beirut Epileptic EEG Dataset**" is utilized [10], [17]. This dataset is utilized in various SOTA studies that focus on epileptic seizure detection using EEG recordings [18], [19]. This dataset is preferred due to its organized structure as well as the variety and abundance of its

epileptic and healthy EEG recordings. Moreover, its recording conditions, properties, and labels are explained in great detail for the possible usage of the dataset in different ML studies. In means of its properties, this dataset was recorded and labeled at the epilepsy monitoring unit of the American University of Beirut Medical Center [10], [17]. The EEG signals were recorded from 6 epilepsy patients in a one-year period. The measurements were taken from multichannel (multiple numbers of electrodes placed on the scalp of the patients) EEG that consists of 19 channels, and they were sampled at 500 Hz [10], [17]. The data was labeled according to the seizure-free condition (class 0), complex partial seizure condition (class 1), electrographic seizure condition (class 2), and video-detected seizure condition (class 3). Hence, it was observed that this dataset can be transformed to be used for a binary classification task as well. By pre-processing and analyzing this dataset with the chosen tools and ML algorithms, one of our project's main aims is to perform a differentiation between EEG signals that contain and do not contain the "epileptic seizure" condition.

Importantly, this dataset was divided into train and test sets for both its features and labels by its recorders in the ratios of 90% and 10% respectively (for train and test sets), and these separate sets were saved as MATLAB files. Considering that the ratio is low for the feature and label test sets, both the train and test sets were transformed to "csv" files and were combined using Python in order to have a single dataset ("csv" file) that contains both the features and labels. This gave the opportunity of shuffling, normalizing, and splitting the dataset into train-test-validation sets according to the desired parameters. This overall dataset consists of 7790 rows that correspond to the summation, or in other words total representation, of all the recorded signals. Moreover, it has 9500 columns that correspond to the concetanation of the recorded channels and their duration in accordance with the sampling rate of 500 Hz at each row ($19 \times 500 = 9500$). This structure of the dataset was organized by the monitoring unit of the American University of Beirut Medical Center according to the information that each 19 channel record similar seizure or seizure-free condition when applied to a patient. The final column in the dataset represent the label for each row according to 4 classes that correspond to 4 labels. Among these labels, classes 1, 2, and 3 represent the epileptic seizure condition whereas class 0 represents the epileptic seizure condition. Noticeably, the number of classes 2 and 3 in the label column (both corresponding to the epileptic seizure condition) was very low compared to the number of classes 0 and 1 in the label column (corresponding to the epileptic seizure free and epileptic seizure conditions respectively). Hence, the potential of this dataset being utilized for a binary classification problem was observed in accordance with a great number of SOTA studies performing binary classification instead of a 4-class classification. The rows with labels 2 and 3 were deleted from the dataset and a binary classification was performed according classes 0 and 1 only. Consequently, we utilized this dataset in the second half of our project to perform binary classification in which class 0 represents the epileptic seizure free condition whereas class 1 represents the epileptic seizure condition. In the figures below, the initial dataset that has 4 classes can be observed respectively.

Unnamed	0	1	2	3	4	5	6	7	8	9	10
0	-2.98E-05	0.0206	-0.00626	-0.00687	0.00293	0.013184	0.015076	0.004823	-0.00418	-0.00735	0.012543
1	-0.00195	0.02121	-0.0192	-0.07059	0.009644	0.029114	0.022065	-0.00149	-0.01212	-0.01904	0.019105
2	-0.00153	-0.00302	-0.00067	0.001252	0.001008	0.000886	-0.00467	-0.00119	-9.09E-05	-0.0014	-0.00171
3	-0.00128	0.00055	-0.00116	0.000733	0.001191	0.001862	0.000855	-0.00674	-0.00583	-0.00214	-0.00232
4	0.007752	0.016724	0.01294	-0.00278	0.001984	0.010651	0.014954	0.015778	0.018647	0.003602	0.009888
5	0.038056	0.011536	0.017487	0.34171	0.016541	-0.03333	-0.01523	0.023621	0.004945	0.013856	-0.03375
6	0.001374	-9.09E-05	-6.03E-05	-0.00073	0.002076	0.000764	0.003297	0.006837	0.001588	0.003785	0.005311
7	-0.0047	0.004853	-0.01007	-0.00613	-0.00024	0.008149	0.007539	-0.00552	-0.01135	-0.0033	0.00528
8	-0.00912	-0.00738	-0.00296	-0.00293	-0.00363	-0.00326	-0.00342	0.00055	-0.00061	-0.00284	0.001954
9	0.000245	0.000275	-0.00061	0.001588	0.00171	0.001008	-0.00238	-0.00146	-0.00079	0.00113	-0.00015
10	0.025574	0.000397	-0.0101	-0.01468	-0.00357	-0.0003	0.025544	0.019196	-0.01184	0.007539	0.009126
11	0.006196	-0.00421	-0.00445	-0.0003	-0.00195	-0.00821	-0.01239	-0.01028	-0.00851	-0.00137	-0.00839
12	-0.0068	-0.00833	0.000672	-0.00436	-0.00354	-0.00601	-0.00565	0.003358	0.008942	0.005219	0.000489
13	-2.98E-05	0.005555	0.000733	0.008241	0.002869	-0.00095	-0.00223	-0.00565	0.001496	0.000886	-0.0029
14	-0.00095	0.000916	-0.0025	0.000245	0.000916	0.004456	0.003541	-0.00406	-0.00406	-0.00293	0.000581
15	0.002595	0.056306	0.001527	-0.01859	0.006196	0.017792	0.053834	0.035096	0.013917	-0.01703	-0.00659
16	0.000489	3.12E-05	0.001466	0.000153	-0.00095	-0.00153	0.000123	0.002808	0.002167	0.000886	-0.00119
17	0.003205	0.005555	0.00116	-0.00647	0.002778	0.013032	0.011323	0.004426	0.000642	-0.00085	0.015351
18	0.026826	0.025422	0.018677	0.005738	0.010468	0.016022	0.023103	0.019288	0.011414	-0.00302	0.015137
19	-0.00546	0.012055	-0.00568	0.40775	-9.09E-05	0.015382	0.009187	0.005647	-0.005	0.004792	0.014619
20	-0.00861	-0.00043	-0.00766	-0.00433	-0.00186	0.000672	0.002778	-0.00272	-0.00445	-0.00214	0.002991

Fig. 4: A portion of the dataset representing the first 20 rows and their corresponding feature columns ranging from 1 to 10

y
1
1
0
0
0
1
0
1
0
0
2
2
0
0
0
1
0
2
2
1

Fig. 5: A portion of the dataset representing the labels of the first 20 rows

3. Feature Engineering and Data Preprocessing

3.1 The Bonn EEG Time Series Dataset

In the project, Python was utilized as the main programming language due to its flexibility, simple coding structure, available data processing libraries, and user-friendly coding environments. In the feature engineering and data preprocessing parts, the Pandas and NumPy libraries were mainly used in order to manipulate the data. Once the aforementioned dataset was obtained as a "csv" file, the classes were converted to a binary form as explained previously. This was performed by considering class 1 labels as the "epileptic seizure" condition and transforming the other classes ranging from 2 to 5 to 0 in order to consider them as the "seizure-free" condition or in other words the "healthy" condition. This transformation was realized right after the dataset was read as a Pandas DataFrame. The dataset was stored both as a whole and by splitting it into two DataFrames that represent the feature and target variables (labels) respectively for different feature engineering and data preprocessing applications.

Firstly, the rows of the dataset that correspond to different subjects were visualized by plotting all of their data points. Hence, data visualization was performed by the obtainment of these plots in order to perform feature engineering correctly. In the figure below, four plots obtained for two healthy and two epileptic seizure conditions can be seen.

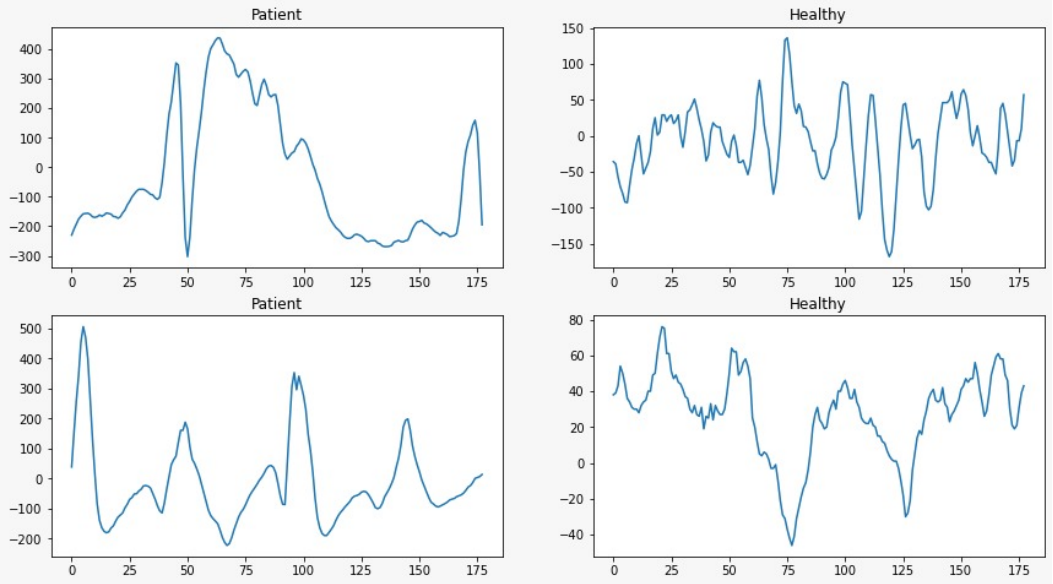


Fig. 6: Data visualization for two healthy and two epileptic seizure conditions

Through this visualization, it was observed that the amplitudes of the epileptic seizure condition are much greater than the healthy condition. In order to take advantage of these amplitude differences, 8 new nonlinear features were derived which correspond to the maximum of each row represented as a new feature (the "MAX" feature), the square of these maximum values (the "MAX2" feature), the cube of these maximum values (the "MAX3" feature), the minimum of each row represented as a new feature (the "MIN" feature), the square of these minimum values (the "MIN2" feature), the cube of these minimum values (the "MIN3" feature), the difference between the "MAX" and "MIN" columns represented as a new feature (the "MxMnDif" feature), and the square of these difference values (the "MxMnDif2" feature). Moreover, 5 new features were derived in accordance with the revisions made to the interim report. These new features correspond to the mean of each row represented as a new feature (the "MEAN" feature), the variance of each row represented as a new feature (the "VAR" feature), the standard deviation of each row represented as a new feature (the "STD" feature), the skewness of each row represented as a new feature (the "SKEW" feature), and the kurtosis of each row represented as a new feature (the "KURT" feature). These new features were

generated both according to the given feedbacks and in order to exploit the statistical properties of the data. It was observed in literature that these statistical features are commonly used in the studies related to the EEG data and their utilization in ML applications [18]. Hence, nonlinear transformations (considering that "max", "min", and their squares and cubes are all nonlinear) were applied to the already-existing 178 features of the dataset while the utilized models are linear. Additionally, a linear transformation was applied to the dataset as well considering that "mean" is a linear operator. The ML models were trained, tested, and validated using all of the 178 features versus using just the derived overall 13 features, and it was observed that the performance of the new features was better although they are fewer in number. This verified that the performed feature engineering was successful and the 13 features were used in the rest of the training, test, and validation processes. Moreover, the big feature number of 178 was decreased which was causing the training processes of the models to take long amounts of time, and the most beneficial relationships among the data points were represented.

In the data preprocessing part, the features of the dataset were normalized considering that they were in raw condition, the whole dataset (now along with their labels) was shuffled, and the whole dataset was separated into train and test sets (also validation in some cases) with the written generic Python functions that can work with any dataset. The z-score standardization was used as the normalization metric considering that min-max normalization can be problematic if the minimum and maximum values are equal to each other. The z-score standardization was applied in a column-wise manner by finding the mean and standard deviation of each column, subtracting this mean from each element of the column, and dividing each element of the column by the found standard deviation. The feature engineering and data preprocessing parts were completed through these steps. Finally, the mathematical equations applied for performing z-score standardization can be observed in the following figures.

$$z = \frac{x - \mu}{\sigma}$$

$$\mu = \text{Mean}$$

$$\sigma = \text{Standard Deviation}$$

Fig. 7: The application of z-score standardization (normalization)

$$\bar{x} = \frac{\sum x}{N}$$

$$\sum x = \text{the sum of } x$$

$$N = \text{number of data}$$

Fig. 8: The formula of the mean (μ)

$$\text{SD} = \sqrt{\frac{\sum |x - \bar{x}|^2}{n}}$$

Fig. 9: The formula of the standard deviation (σ)

3.2 The Beirut Epileptic EEG Dataset

Once the "**The Beirut Epileptic EEG Dataset**" was obtained as a "csv" file, the classes were converted to a binary form as explained previously. This was performed by considering class 0 labels as the "seizure-free" condition and considering class 1 labels as the "epileptic seizure" condition. The other classes that are 2 and 3 were removed from the dataset as they were very low in number and did not have a significant contribution to the dataset in terms of classification. Their removal from the dataset did not create a class imbalance as well. This transformation was realized right after the dataset was read as a Pandas DataFrame. The dataset was stored both as a whole and by splitting it

into two DataFrames that represent the feature and target variables (labels) respectively for different feature engineering and data preprocessing applications.

Firstly, the rows of the dataset that correspond to different channels and recordings were visualized by plotting all of their data points. Hence, data visualization was performed by the obtainment of these plots in order to perform feature engineering correctly. In the figure below, four plots obtained for two healthy and two epileptic seizure conditions can be seen.

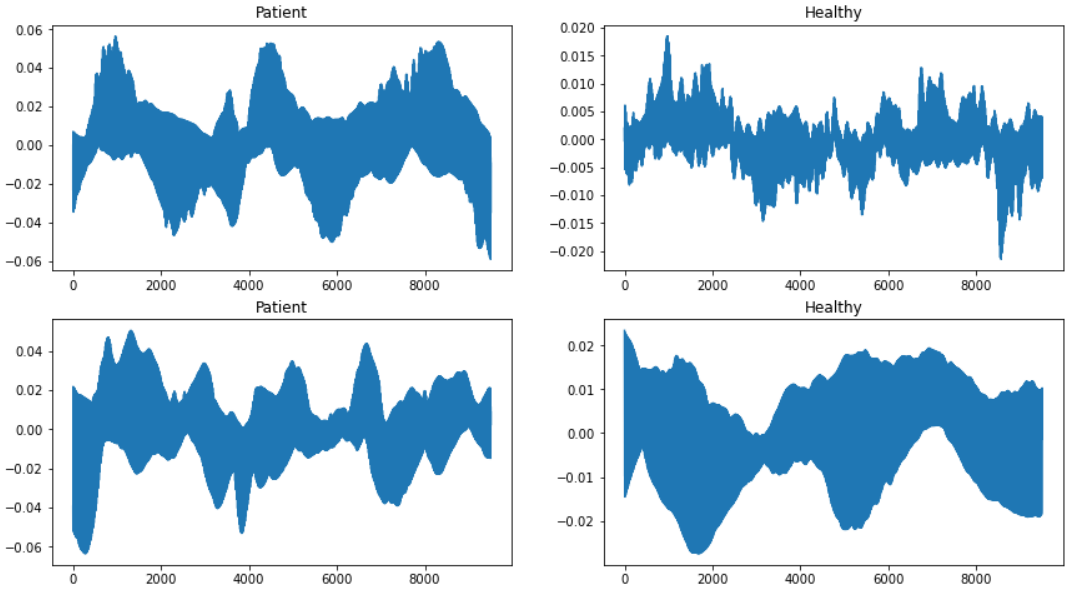


Fig. 10: Data visualization for two healthy and two epileptic seizure conditions

Through this visualization, it was observed that the amplitudes of the epileptic seizure and seizure-free condition are not very different from each other hence not much difference was observed in terms of the amplitude difference. As a result, the 8 nonlinear features that were derived in the first dataset which were the "MAX" feature, the "MAX2" feature, the "MAX3" feature, the "MIN" feature, the "MIN2" feature, the "MIN3" feature, the "MxMnDif" feature, and the "MxMnDif2" feature were not utilized for this dataset. On the other hand, 5 new features were derived from the second dataset in accordance with the revisions made to the interim report. These new features correspond to the mean of each row represented as a new feature (the "MEAN" feature), the variance of each row represented as a new feature (the "VAR" feature), the standard deviation of each row represented as a new feature (the "STD" feature), the skewness of each row represented as a new feature (the "SKEW" feature), and the kurtosis of each row represented as a new feature (the "KURT" feature). These new features were generated both according to the given feedbacks and in order to exploit the statistical properties of the data. It was observed in literature that these statistical features are commonly used in the studies related to the EEG data and their utilization in ML applications [18]. Hence, nonlinear transformations (considering that "variance", "standard deviation", "skewness", and "kurtosis" are all nonlinear) were applied to the already-existing 9500 columns of the dataset while the utilized models are linear. Additionally, a linear transformation was applied to the dataset as well considering that "mean" is a linear operator. The ML models were trained, tested, and validated using all of the 9500 columns versus using just the derived 5 features, and it was observed that the performance of the new features was better although they are fewer in number. This verified that the performed feature engineering was successful and the 5 features were used in the rest of the training, test, and validation processes. Moreover, the big column number of 9500 was decreased which was causing the training processes of the models to take long amounts of time, and the most beneficial relationships among the data points were represented. Importantly, performing feature engineering for this dataset was necessary considering that the already-existing 9500 columns were not features themselves. Hence, through feature engineering, features were derived from raw EEG data.

In the data preprocessing part, the features of the dataset were normalized considering that they were in raw condition, the whole dataset (now along with their labels) was shuffled, and the whole dataset was separated into train and test sets (also validation in some cases) with the written generic Python functions that can work with any dataset. The min-max normalization was used for this dataset as the normalization metric considering that min-max normalization is not problematic for this dataset since the minimum and maximum values are not equal to each other. Furthermore, min-max normalization was preferred for this dataset in other studies so that this method was chosen [18], [19]. The min-max normalization was applied in a column-wise manner by finding the minimum and maximum values of each column, subtracting this minimum value from each element of the column, and dividing each element of the column by the difference of the maximum and minimum values. The feature engineering and data preprocessing parts were completed through these steps. Finally, the mathematical equation applied for performing min-max normalization can be observed in the following figure.

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)}$$

Fig. 11: The application of min-max normalization

4. Implemented Machine Learning Approaches

In this project, we decided to implement three machine learning algorithms: **Decision Trees (DT), k-Nearest Neighbors (kNN), and Multilayer Perceptron (MLP)**. kNN is a long-standing, well-studied, and highly powerful algorithm in classification tasks. Moreover, it is a fast algorithm since it is not a gradient-based iterative algorithm and it can be implemented in a vectorized manner. DT is also another powerful algorithm that does not rely on iterative weight updates. It is also flexible in terms of tasks, easy to interpret and visualize, and does not require costly data preparation. MLPs are indeed highly nonlinear multivariate functions of unknown variables. These unknown variables are learned through the gradient descent on the loss which is the function of these unknown variables and data to process.

As part of the progress report, we have implemented the DT and kNN algorithms and presented our results. In the final report, MLP also will be implemented and its results will be provided.

4.1 Decision Tree (DT) Algorithm

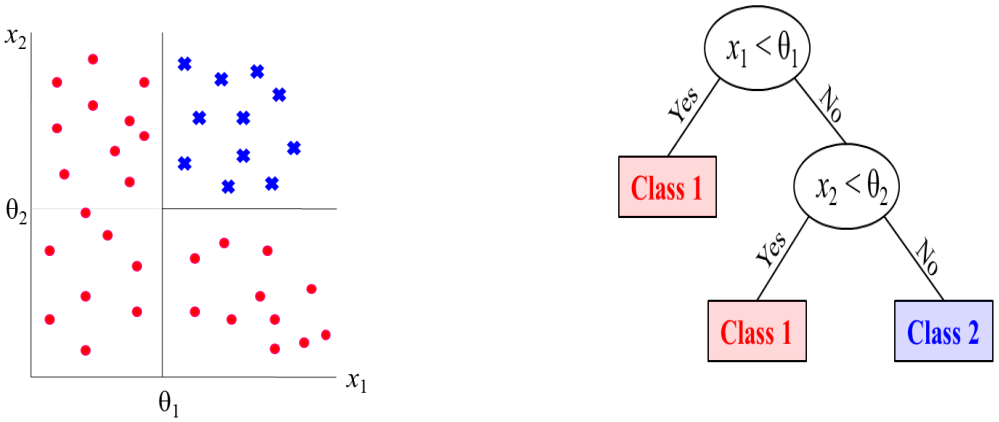


Fig.12: DT for binary classification [20]

DTs are built in the form of a tree structure that provides a model for classification and regression. DT corresponds to dividing the input space into different regions, each of which stands for a decision. In other words, in the DT algorithm, mainly, the goal is to find these decision regions. In DT, internal and leaf nodes correspond to the test function and final decision (a class for classification, and a value for a regression), respectively.

In the training stage, mainly, the goal is to build a tree with a possible minimum error. Starting from the root node, at each step, the "best split" is selected among all possible choices and iteratively continued until all leaf nodes are pure. It is worth indicating that "pure" in this context means that one class exists under the leaf node. In other words, entropy in a leaf node is 0.

In a classification tree, how to find the best split?

First, decide on the impurity function: **Entropy or Gini**

$I(m)_{entropy} = -\sum_{i=1}^{i=N_c}(P(C_i)_m * logP(C_i)_m)$,
$I(m)_{gini} = 1 - \sum_{i=1}^{i=N_c}(P(C_i)_m)^2$, where m and N_c are a feature and number of classes, respectively.

Second, calculate the total split impurity using one of the above impurity functions.

$I_{total}(S) = P_{left}I(left) + P_{right}I(right)$, where P_{left} and P_{right} correspond to the relative number of elements in the left and right branches.

Finally, select the best split which has the lowest total impurity and iteratively repeat this process until all leaf nodes are pure. By taking into consideration these, the training process of DT is an exhaustive process.

How to avoid overfitting?

Two options can be applied to refrain from overfitting: **Deterministic tree length and information gain.**

For deterministic tree length, a predefined maximum tree length is defined and splitting is stopped when the maximum length is reached. For information gain, if the information gain in a split is less than a threshold value, then splitting is stopped and the majority class in the node is evaluated as the decision class. In our study, we used pruning-based information gain to avoid overfitting.

For our project and our datasets, we chose DT as one of our ML algorithms since it does not take a great amount of time to train and test the datasets in terms of their classification accuracy, can work in limited computational power scenarios, and it can operate with entire datasets flexibly. Some explanations regarding our though processes behind choosing this model were given at the beginning of this section titled "Implemented Machine Learning Approaches" as well. Moreover, we observed that this model was frequently preferred for our datasets in the ML literature and similar studies so that this observation was another justification for us.

4.2 k-Nearest Neighbors (kNN) Algorithm

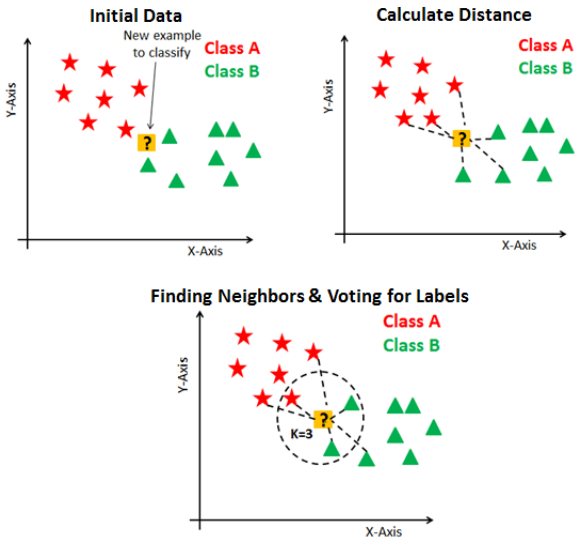


Fig. 13: k-Nearest neighbor algorithm for binary classification [21]

kNN is a highly simple but judicious algorithm. In Fig. 13, the process of binary classification of a single test sample is illustrated. It can also be applied to multiclass problems without loss of generalization.

Pseudocode:

Input: $X \in \mathbb{R}^{N \times p}$ (Training Set), $Y \in \mathbb{R}^{N}$(Training Labels), $x \in \mathbb{R}^{p}$(Test sample), y (test sample labels)

Output: $y_{predict}$

$y_{predict}$=[]
for each test sample x
    distance_all=[]
    for n = 0 → N:
        distance = distance_metric(X_n*,* x*)*
        distance_all.append(distance)
    end
    indices= choose(sort(distance, "ascending"),k)
    y_best = max(frequency(Y[indices]))
    $y_{predict}$*.append(y_best)*
return $y_{predict}$

In the above pseudocode, an algorithm for predictions of a set of test samples is shown. After collecting the predictions for each test sample, predicted labels $y_{predict}$ are compared to true labels y and test accuracy is calculated.

For our project and our datasets, we chose kNN as one of our ML algorithms since it does not follow the traditional approach for train and test processes as opposed to other ML algorithms (it follows a distance comparison algorithm) so that it does not require a training period for classification purposes. The kNN algorithm can work in limited computational power scenarios, and it can interact with different parts of the dataset easily without depending on the information coming from previous steps since it is not a cumulative algorithm. Some explanations regarding our though processes behind choosing this model were given at the beginning of this section titled "Implemented Machine Learning Approaches" as well. Moreover, we observed that this model was frequently preferred for our datasets in the ML literature and similar studies so that this observation was another justification for us.

4.3 Multilayer Perceptron (Feed-forward Neural Networks)

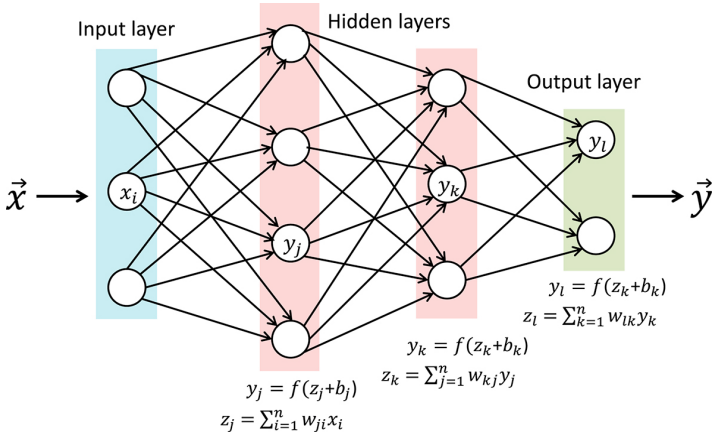


Fig. 14: Feed-forward Neural Network for Binary Classification [22]

The last algorithm that will be is MLP, also named as feed-forward neural networks. In MLP, a multidimensional input vector $x \in \mathbb{R}^p$ is fed to the network. The input vector is then mapped to a different multidimensional vector using a weighted sum of input features and a nonlinear activation function. This procedure is iteratively continued until the output layer. At the output layer, the loss of predicted value and true value is calculated. In binary classification tasks, binary cross entropy loss is used commonly. Moreover, the network weights are learned through backpropagation, which is a gradient-based learning algorithm, iteratively. In Fig.14, the forward propagation of input values from the input layer to the output layer is shown.

For our project and our datasets, we chose MLP as one of our ML algorithms since it well-suited for classification tasks in which the data is assigned classes or labels. Hence, it can work with tabular data (similar to our datasets) in which the dataset is organized in terms of rows and columns. The MLP algorithm can work in limited computational power scenarios in accordance with their number of layers. Increasing the number of layers of the MLP algorithm can enable it to achieve higher performance results although it may cause it to run slower. In addition, MLP algorithms can work with large datasets (similar to ours) well and importantly, we add nonlinear functions (activation functions in different layers) without increasing the number of parameters which can aid the performance of the model. Moreover, we observed that this model was frequently preferred for our datasets in the ML literature and similar studies so that this observation was another justification for us.

5. Simulation Results

In this report, we present the simulation results of the detection of epileptic seizures for two datasets using DT, kNN, and MLP algorithms. For the selection of the best hyperparameters, two different validation strategies were applied. In the first strategy, we split the data as training and test data by a ratio of 0.85/0.15. In order to select the best-performing hyperparameters, we use k-Fold cross-validation with grid-search of hyperparameters. In the second strategy, we split the data as training, test and validation data by a ratio of 0.70/0.15/0.15 respectively. In order to select the best-performing hyperparameters, we use the training and validation sets in a grid-search architecture. Afterwards, we find the accuracy results for the test set using the best-performing hyperparameters along with the training set. In the MLP algorithm, a different strategy was applied in which a part of the training set in the ratio of approximately 0.15 is taken as the validation set. For this strategy, the ratios of the training and test sets are again 0.85/0.15. Importantly, the train, test, and validation sets that are used for each algorithm (and for each strategy) are the same for each dataset in order to achieve a means of standardization.

5.1 The Bonn EEG Time Series Dataset

5.1.1 Decision Tree (DT) Algorithm with k-Fold Cross-Validation

In this DT, we used 5- Fold cross-validation to select the best hyperparameter combination and evaluated the best model on the test data. 5-fold cross-validation is applied on the training set, which is %85 of the entire dataset, and evaluations are made on the separate test set, which is %15 of the entire dataset. The validation sets in 5-fold cross-validation are approximately 15% as well. In Fig.15, classification accuracies are shown with respect to **k** (pruning rate) and **impurity criteria**. In Fig. 17, a grid-search with 5-Fold cross-validation results are shown. Results show that a significant pruning rate brings about underfitting of the model. ***Also, the total time for 5-fold is: 5624.09 seconds.***

impurity	entropy	gini
0.0	95.76	95.74
0.1	95.38	96.02
0.2	95.43	80.33
0.3	95.43	80.33
0.7	80.33	80.33
0.8	80.33	80.33
0.9	80.33	80.33

Fig. 15: 5-Fold cross-validation accuracies on the training set

```
--------------- TOTAL ACCURACY -------------------
Accuracy is 96.7
********* CLASS BASED ACCURACY********************
Class 0
Accuracy is 97.26
Class 1
Accuracy is 94.69
```

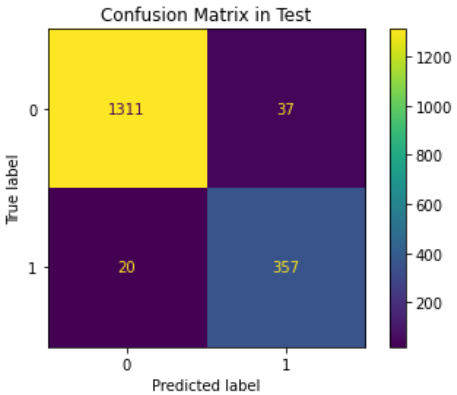


Fig. 16: Total and class-based accuracies on the best model with prune rate = 0.1 and “gini” on the test set

```
Fold  1 is completed.
Fold  2 is completed.
Fold  3 is completed.
Fold  4 is completed.
```

Fold 5 is completed.
Prune Rate: 0 Impurity: entropy : 95.76
Fold 1 is completed.
Fold 2 is completed.
Fold 3 is completed.
Fold 4 is completed.
Fold 5 is completed.
Prune Rate: 0 Impurity: gini : 95.74
Fold 1 is completed.
Fold 2 is completed.
Fold 3 is completed.
Fold 4 is completed.
Fold 5 is completed.
Prune Rate: 0.1 Impurity: entropy : 95.38
Fold 1 is completed.
Fold 2 is completed.
Fold 3 is completed.
Fold 4 is completed.
Fold 5 is completed.
Prune Rate: 0.1 Impurity: gini : 96.02
Fold 1 is completed.
Fold 2 is completed.
Fold 3 is completed.
Fold 4 is completed.
Fold 5 is completed.
Prune Rate: 0.2 Impurity: entropy : 95.43
Fold 1 is completed.
Fold 2 is completed.
Fold 3 is completed.
Fold 4 is completed.
Fold 5 is completed.
Prune Rate: 0.2 Impurity: gini : 80.33
Fold 1 is completed.
Fold 2 is completed.
Fold 3 is completed.
Fold 4 is completed.
Fold 5 is completed.
Prune Rate: 0.3 Impurity: entropy : 95.43
Fold 1 is completed.
Fold 2 is completed.
Fold 3 is completed.
Fold 4 is completed.
Fold 5 is completed.
Prune Rate: 0.3 Impurity: gini : 80.33
Fold 1 is completed.
Fold 2 is completed.
Fold 3 is completed.
Fold 4 is completed.
Fold 5 is completed.
Prune Rate: 0.7 Impurity: entropy : 80.33
Fold 1 is completed.
Fold 2 is completed.
Fold 3 is completed.
Fold 4 is completed.
Fold 5 is completed.
Prune Rate: 0.7 Impurity: gini : 80.33
Fold 1 is completed.
Fold 2 is completed.
Fold 3 is completed.
Fold 4 is completed.
Fold 5 is completed.
Prune Rate: 0.8 Impurity: entropy : 80.33
Fold 1 is completed.
Fold 2 is completed.
Fold 3 is completed.
Fold 4 is completed.
Fold 5 is completed.
Prune Rate: 0.8 Impurity: gini : 80.33
Fold 1 is completed.
Fold 2 is completed.
Fold 3 is completed.

Fold 4 is completed.
Fold 5 is completed.
Prune Rate: 0.9 Impurity: entropy : 80.33
Fold 1 is completed.
Fold 2 is completed.
Fold 3 is completed.
Fold 4 is completed.
Fold 5 is completed.
Prune Rate: 0.9 Impurity: gini : 80.33
Total time for 5 -fold is: 5624.085533618927 seconds.

Fig. 17: Grid-search with 5-Fold cross-validation on the training set

5.1.2 K-Nearest Neighbors (kNN) Algorithm with k-Fold Cross-Validation

In this kNN, we used 5- Fold cross-validation to select the best hyperparameter combination and evaluated the best model on the test data. 5-fold cross-validation is applied on the training set, which is %85 of the entire dataset, and evaluations are made on the separate test set, which is %15 of the entire dataset. The validation sets in 5-fold cross-validation are approximately 15% as well. In Fig.18, classification accuracies are shown with respect to **k** (number of nearest neighbors) and **distance metric**. In Fig. 20, a grid-search with 5-Fold cross-validation results are shown. ***Also, the total time for 5-fold is: 107.68 seconds.***

k	euclidian	manhattan
3	96.79	96.96
5	96.85	97.21
7	96.95	97.16
11	96.90	97.14
15	96.74	96.92

Fig. 18: 5-Fold cross-validation accuracies on the training set

```
--------------- TOTAL ACCURACY ------------------
Accuracy is 96.75
********* CLASS BASED ACCURACY**********
Class 0
Accuracy is 98.15
Class 1
Accuracy is 91.78
```

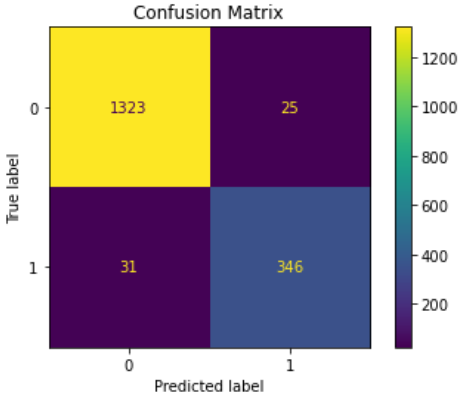


Fig. 19: Total and class-based accuracies on the best model with k = 5 and “manhattan” on the test set

```
Fold  1 is completed.
Fold  2 is completed.
Fold  3 is completed.
Fold  4 is completed.
Fold  5 is completed.
K-Nearest Neighbor: 3  Metric:  euclidian :  96.79
Fold  1 is completed.
Fold  2 is completed.
Fold  3 is completed.
Fold  4 is completed.
Fold  5 is completed.
K-Nearest Neighbor: 3  Metric:  manhattan :  96.96
Fold  1 is completed.
Fold  2 is completed.
Fold  3 is completed.
Fold  4 is completed.
Fold  5 is completed.
K-Nearest Neighbor: 5  Metric:  euclidian :  96.85
Fold  1 is completed.
Fold  2 is completed.
Fold  3 is completed.
Fold  4 is completed.
Fold  5 is completed.
K-Nearest Neighbor: 5  Metric:  manhattan :  97.21
Fold  1 is completed.
Fold  2 is completed.
Fold  3 is completed.
Fold  4 is completed.
Fold  5 is completed.
K-Nearest Neighbor: 7  Metric:  euclidian :  96.95
Fold  1 is completed.
Fold  2 is completed.
Fold  3 is completed.
Fold  4 is completed.
Fold  5 is completed.
K-Nearest Neighbor: 7  Metric:  manhattan :  97.16
Fold  1 is completed.
Fold  2 is completed.
Fold  3 is completed.
Fold  4 is completed.
Fold  5 is completed.
K-Nearest Neighbor: 11  Metric:  euclidian :  96.9
Fold  1 is completed.
Fold  2 is completed.
Fold  3 is completed.
Fold  4 is completed.
Fold  5 is completed.
K-Nearest Neighbor: 11  Metric:  manhattan :  97.14
Fold  1 is completed.
Fold  2 is completed.
Fold  3 is completed.
Fold  4 is completed.
Fold  5 is completed.
K-Nearest Neighbor: 15  Metric:  euclidian :  96.74
Fold  1 is completed.
Fold  2 is completed.
Fold  3 is completed.
Fold  4 is completed.
Fold  5 is completed.
K-Nearest Neighbor: 15  Metric:  manhattan :  96.92
Total time for  5 -fold is:  107.68478798866272 seconds.
```

Fig. 20: Grid-search with 5-Fold cross-validation on the training set

5.1.3 Decision Tree (DT) Algorithm with Validation Set

In this DT, we used a validation set that accounts for the 15% of all the dataset to select the best hyperparameter combination, and evaluated the best model on the test data that accounts for the

15% of all the dataset. The train set corresponds to the 70% of the dataset. In Fig. 21, classification accuracies are shown with respect to **k** (pruning rate) and **impurity criteria**. In Fig. 23, a grid-search with 15% validation set results are shown. Results show that a significant pruning rate brings about underfitting of the model. ***Also, the total time for the grid-search is: 838.72 seconds.***

impurity	entropy	gini
0.0	95.13	95.30
0.1	95.01	95.83
0.2	95.01	79.71
0.3	95.01	79.71
0.7	79.71	79.71
0.8	79.71	79.71
0.9	79.71	79.71

Fig. 21: Grid-search with 15% validation set accuracies on the training set

```
--------------- TOTAL ACCURACY -------------------
Accuracy is 96.17
********* CLASS BASED ACCURACY********************
Class 0
Accuracy is 96.27
Class 1
Accuracy is 95.78
```

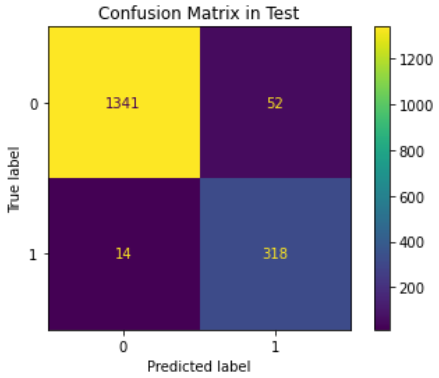


Fig. 22: Total and class-based accuracies on the best model with prune rate = 0.1 and "gini" on the test set

Prune Rate: 0 Impurity: entropy : 95.13
Prune Rate: 0 Impurity: gini : 95.3
Prune Rate: 0.1 Impurity: entropy : 95.01
Prune Rate: 0.1 Impurity: gini : 95.83
Prune Rate: 0.2 Impurity: entropy : 95.01
Prune Rate: 0.2 Impurity: gini : 79.71
Prune Rate: 0.3 Impurity: entropy : 95.01
Prune Rate: 0.3 Impurity: gini : 79.71
Prune Rate: 0.7 Impurity: entropy : 79.71
Prune Rate: 0.7 Impurity: gini : 79.71
Prune Rate: 0.8 Impurity: entropy : 79.71
Prune Rate: 0.8 Impurity: gini : 79.71
Prune Rate: 0.9 Impurity: entropy : 79.71
Prune Rate: 0.9 Impurity: gini : 79.71
Total time for 7 x 2 grid search is: 838.7186055183411 seconds.

Fig. 23: Grid-search with 15% validation set on the training set

5.1.4 K-Nearest Neighbors (kNN) Algorithm with Validation Set

In this kNN, we used a validation set that accounts for the 15% of all the dataset to select the best hyperparameter combination, and evaluated the best model on the test data that accounts for the 15% of all the dataset. The train set corresponds to 70% of the dataset.. In Fig.24, classification accuracies are shown with respect to **k** (number of nearest neighbors) and **distance metric**. In Fig. 26, a 15% validation set results are shown. ***Also, the total time for the grid-search is: 11.98 seconds.***

k	euclidian	manhattan
3	96.64	96.52
5	96.70	96.58
7	96.52	96.70
11	96.52	96.58
15	96.58	96.81

Fig. 24: Grid-search with 15% validation set accuracies on the training set

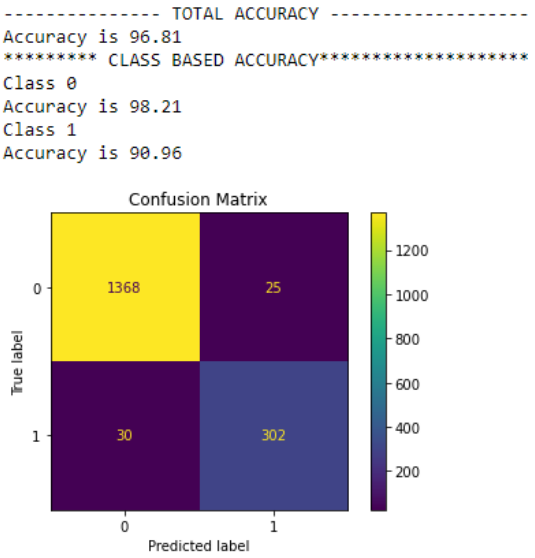


Fig. 25: Total and class-based accuracies on the best model with k = 15 and “manhattan” on the test set

K-Nearest Neighbor: 3 Metric: euclidian : 96.64
K-Nearest Neighbor: 3 Metric: manhattan : 96.52
K-Nearest Neighbor: 5 Metric: euclidian : 96.7
K-Nearest Neighbor: 5 Metric: manhattan : 96.58
K-Nearest Neighbor: 7 Metric: euclidian : 96.52
K-Nearest Neighbor: 7 Metric: manhattan : 96.7
K-Nearest Neighbor: 11 Metric: euclidian : 96.52
K-Nearest Neighbor: 11 Metric: manhattan : 96.58
K-Nearest Neighbor: 15 Metric: euclidian : 96.58
K-Nearest Neighbor: 15 Metric: manhattan : 96.81
Total time for 5 x 2 grid search is: 11.979809284210205 seconds.

Fig. 26: Grid-search with 15% validation set on the training set

5.1.5 Multilayer Perceptron (MLP) Algorithm

Table 1: Model performance in "**The Bonn EEG Time Series Dataset**" with respect to model structure and hyperparameters such as learning rate, activation functions in each layer, loss function, weight initialization, batch size, and epoch numbers

Layers	Activations	Learning R.	Loss	Initialization	Batch	Epoch	Val. Acc.	Test Acc.
[2]	Softmax	0.01	Cross Entropy	Xaiver	100	500	96.11	95.47
[2]	Softmax	0.01	MSE	Xaiver	100	500	96.57	95.59
[6,2]	[Sigmoid, Softmax]	0..05	Cross Entropy	Xaiver	100	500	96.82	95.94
[6,2]	[Sigmoid, Softmax]	0..05	MSE	Xaiver	100	500	96.47	95.53
[6,2]	[Sigmoid, Softmax]	0.05	Cross Entropy	Normal	100	500	96.62	96.17
[6,2]	[Tanh, Softmax]	0..05	Cross Entropy	Normal	100	500	96.93	96.00
[13,6,2]	[Tanh,Tanh,Softmax]	0.05	Cross Entropy	Normal	100	500	97.18	96.46
[13,6,2]	[Tanh,Tanh,Softmax]	0.05	MSE	Normal	100	500	96.77	95.84
[13,6,2]	[Tanh,Tanh,Softmax]	0.05	Cross Entropy	Xavier	100	500	97.34	96.28
[13,6,2]	[Tanh,Tanh,Softmax]	0.1	Cross Entropy	Xavier	100	500	97.74	96.28
[40,15,6,2]	[Tanh,Tanh,Tanh,Softmax]	0.1	Cross Entropy	Xavier	100	500	97.64	96.81
[40,15,6,2]	[Tanh,Tanh,Tanh,Softmax]	0.1	Cross Entropy	Xaiver	50	1000	98.92	96.92
[40,15,6,2]	[Tanh,Tanh,Tanh,Softmax]	0.1	Cross Entropy	Xaiver	50	2000	99.23	96.11
[40,15,6,2]	[Tanh,Tanh,Tanh,Softmax]	0.05	Cross Entropy	Xavier	500	1000	96.72	96.75

Validation and test accuracies can be observed in Table 1 above. We have implemented a generic and modular MLP similar to the built–in Keras MLP libraries. In this implementation, any size of new layers can be added with layer-specific activation functions according to the desire. To achieve this flexibility, we have created two classes: Layer and Sequential.

Although we work on a binary classification task, our implementation is generic and independent from our task. In other words, it also works for different classification tasks with more classes. We only show a portion of the parameter choices, however activation functions of the layer can differ and weights can be initialized differently such as uniform, normal, Xavier and He. We provide a concise and clear code and it can be easily understood by looking at the Appendix section.

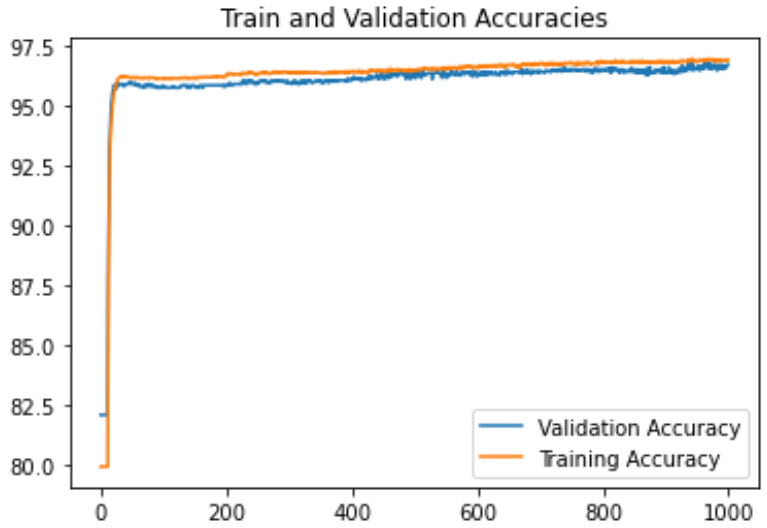


Fig. 27: Train and validation accuracies with respect to the epoch number

The best hyperparameters for this result above can be observed in the expression given below. Although there are several models with better validation accuracy, we refrain from presenting them since the model does not converge for a long period of epoch. Thus, we present a converged model with reasonable performance in training and test. Finally, the overall test accuracy with the best hyperparameters is obtained as **96.75%.**

Selected model: [40,15,6,2], [Tanh,Tanh,Tanh,Softmax], 0.05, Cross Entropy, Xavier, 500, 1000

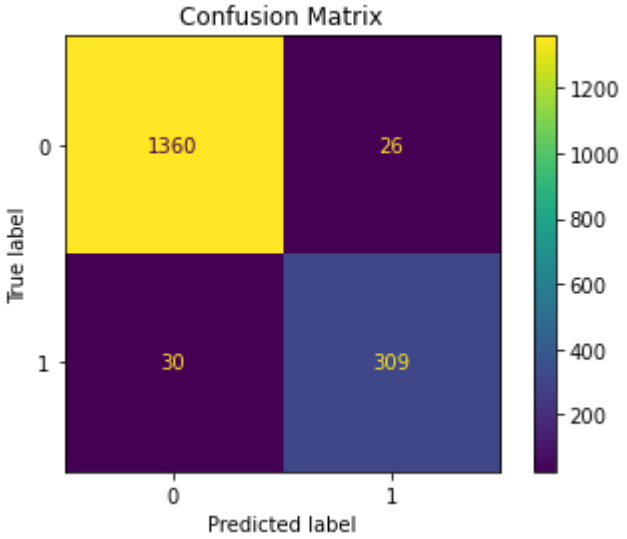


Fig. 28: Confusion matrix for the test set performance with respect to given model

5.2 The Beirut Epileptic EEG Dataset

5.2.1 Decision Tree (DT) Algorithm with k-Fold Cross-Validation

In this DT, we used 5- Fold cross-validation to select the best hyperparameter combination and evaluated the best model on the test data. 5-fold cross-validation is applied on the training set, which is %85 of the entire dataset, and evaluations are made on the separate test set, which is %15 of the entire dataset. The validation sets in 5-fold cross-validation are approximately 15% as well. In Fig. 29, classification accuracies are shown with respect to **k** (pruning rate) and **impurity criteria**. In Fig. 31, a grid-search with 5-Fold cross-validation results are shown. Results show that a significant pruning rate brings about underfitting of the model. ***Also, the total time for 5-fold is: 884.82 seconds.***

impurity	entropy	gini
0.0	79.44	79.76
0.1	79.46	56.04
0.2	79.46	56.04
0.3	56.04	56.04
0.7	56.04	56.04
0.8	56.04	56.04
0.9	56.04	56.04

Fig. 29: 5-Fold cross-validation accuracies on the training set

```
--------------- TOTAL ACCURACY -------------------
Accuracy is 76.61
********* CLASS BASED ACCURACY********************
Class 0
Accuracy is 87.54
Class 1
Accuracy is 62.02
```

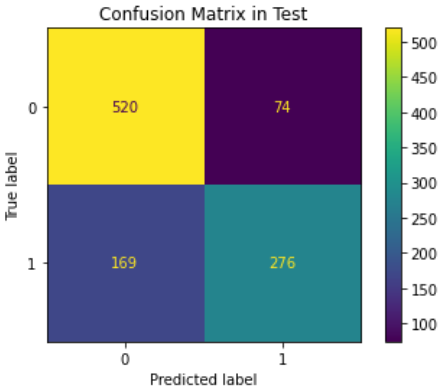


Fig. 30: Total and class-based accuracies on the best model with prune rate = 0.0 and “gini” on the test set

Fold 1 is completed.
Fold 2 is completed.
Fold 3 is completed.
Fold 4 is completed.
Fold 5 is completed.
Prune Rate: 0 Impurity: entropy : 79.44
Fold 1 is completed.
Fold 2 is completed.
Fold 3 is completed.
Fold 4 is completed.
Fold 5 is completed.
Prune Rate: 0 Impurity: gini : 79.76
Fold 1 is completed.
Fold 2 is completed.
Fold 3 is completed.
Fold 4 is completed.
Fold 5 is completed.
Prune Rate: 0.1 Impurity: entropy : 79.46
Fold 1 is completed.
Fold 2 is completed.
Fold 3 is completed.
Fold 4 is completed.
Fold 5 is completed.
Prune Rate: 0.1 Impurity: gini : 56.04
Fold 1 is completed.
Fold 2 is completed.
Fold 3 is completed.
Fold 4 is completed.
Fold 5 is completed.
Prune Rate: 0.2 Impurity: entropy : 79.46
Fold 1 is completed.
Fold 2 is completed.
Fold 3 is completed.
Fold 4 is completed.
Fold 5 is completed.
Prune Rate: 0.2 Impurity: gini : 56.04
Fold 1 is completed.
Fold 2 is completed.
Fold 3 is completed.
Fold 4 is completed.
Fold 5 is completed.
Prune Rate: 0.3 Impurity: entropy : 56.04
Fold 1 is completed.
Fold 2 is completed.

Fold 3 is completed.
Fold 4 is completed.
Fold 5 is completed.
Prune Rate: 0.3 Impurity: gini : 56.04
Fold 1 is completed.
Fold 2 is completed.
Fold 3 is completed.
Fold 4 is completed.
Fold 5 is completed.
Prune Rate: 0.7 Impurity: entropy : 56.04
Fold 1 is completed.
Fold 2 is completed.
Fold 3 is completed.
Fold 4 is completed.
Fold 5 is completed.
Prune Rate: 0.7 Impurity: gini : 56.04
Fold 1 is completed.
Fold 2 is completed.
Fold 3 is completed.
Fold 4 is completed.
Fold 5 is completed.
Prune Rate: 0.8 Impurity: entropy : 56.04
Fold 1 is completed.
Fold 2 is completed.
Fold 3 is completed.
Fold 4 is completed.
Fold 5 is completed.
Prune Rate: 0.8 Impurity: gini : 56.04
Fold 1 is completed.
Fold 2 is completed.
Fold 3 is completed.
Fold 4 is completed.
Fold 5 is completed.
Prune Rate: 0.9 Impurity: entropy : 56.04
Fold 1 is completed.
Fold 2 is completed.
Fold 3 is completed.
Fold 4 is completed.
Fold 5 is completed.
Prune Rate: 0.9 Impurity: gini : 56.04
Total time for 5 -fold is: 884.8189516067505 seconds.

Fig. 31: Grid-search with 5-Fold cross-validation on the training set

5.2.2 K-Nearest Neighbors (kNN) Algorithm with k-Fold Cross-Validation

In this kNN, we used 5- Fold cross-validation to select the best hyperparameter combination and evaluated the best model on the test data. 5-fold cross-validation is applied on the training set, which is %85 of the entire dataset, and evaluations are made on the separate test set, which is %15 of the entire dataset. The validation sets in 5-fold cross-validation are approximately 15% as well. In Fig. 32, classification accuracies are shown with respect to **k** (number of nearest neighbors) and **distance metric**. In Fig. 34, a grid-search with 5-Fold cross-validation results are shown. ***Also, the total time for 5-fold is: 19.97 seconds.***

k	euclidian	manhattan
3	78.95	79.03
5	78.03	78.57
7	77.45	78.12
11	78.81	78.68
15	79.29	79.35

Fig. 32: 5-Fold cross-validation accuracies on the training set

```
--------------- TOTAL ACCURACY ------------------
Accuracy is 77.96
********* CLASS BASED ACCURACY********************
Class 0
Accuracy is 89.06
Class 1
Accuracy is 63.15
```

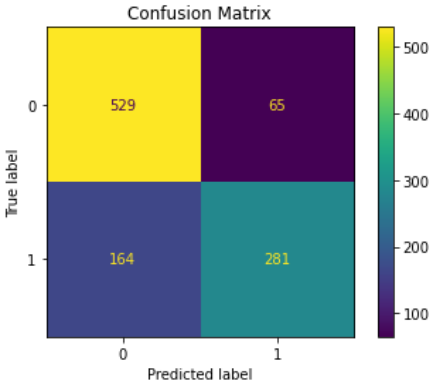


Fig. 33: Total and class-based accuracies on the best model with k = 15 and “manhattan” on the test set

Fold 1 is completed.
Fold 2 is completed.
Fold 3 is completed.
Fold 4 is completed.
Fold 5 is completed.
K-Nearest Neighbor: 3 Metric: euclidian : 78.95
Fold 1 is completed.
Fold 2 is completed.
Fold 3 is completed.
Fold 4 is completed.
Fold 5 is completed.
K-Nearest Neighbor: 3 Metric: manhattan : 79.03
Fold 1 is completed.
Fold 2 is completed.
Fold 3 is completed.
Fold 4 is completed.
Fold 5 is completed.
K-Nearest Neighbor: 5 Metric: euclidian : 78.03
Fold 1 is completed.
Fold 2 is completed.
Fold 3 is completed.
Fold 4 is completed.
Fold 5 is completed.
K-Nearest Neighbor: 5 Metric: manhattan : 78.57
Fold 1 is completed.
Fold 2 is completed.

Fold 3 is completed.
Fold 4 is completed.
Fold 5 is completed.
K-Nearest Neighbor: 7 Metric: euclidian : 77.45
Fold 1 is completed.
Fold 2 is completed.
Fold 3 is completed.
Fold 4 is completed.
Fold 5 is completed.
K-Nearest Neighbor: 7 Metric: manhattan : 78.12
Fold 1 is completed.
Fold 2 is completed.
Fold 3 is completed.
Fold 4 is completed.
Fold 5 is completed.
K-Nearest Neighbor: 11 Metric: euclidian : 78.81
Fold 1 is completed.
Fold 2 is completed.
Fold 3 is completed.
Fold 4 is completed.
Fold 5 is completed.
K-Nearest Neighbor: 11 Metric: manhattan : 78.68
Fold 1 is completed.
Fold 2 is completed.
Fold 3 is completed.
Fold 4 is completed.
Fold 5 is completed.
K-Nearest Neighbor: 15 Metric: euclidian : 79.29
Fold 1 is completed.
Fold 2 is completed.
Fold 3 is completed.
Fold 4 is completed.
Fold 5 is completed.
K-Nearest Neighbor: 15 Metric: manhattan : 79.35
Total time for 5 -fold is: 19.966706037521362 seconds.

Fig. 34: Grid-search with 5-Fold cross-validation on the training set

5.2.3 Decision Tree (DT) Algorithm with Validation Set

In this DT, we used a validation set that accounts for the 15% of all the dataset to select the best hyperparameter combination, and evaluated the best model on the test data that accounts for the 15% of all the dataset. The train set corresponds to 70% of the dataset. In Fig. 35, classification accuracies are shown with respect to **k** (pruning rate) and **impurity criteria**. In Fig. 37, a grid-search with 15% validation set results are shown. Results show that a significant pruning rate brings about underfitting of the model. ***Also, the total time for the grid-search is: 175.54 seconds.***

	entropy	gini
impurity		
0.0	82.29	81.23
0.1	78.83	58.33
0.2	78.83	58.33
0.3	58.33	58.33
0.7	58.33	58.33
0.8	58.33	58.33
0.9	58.33	58.33

Fig. 35: Grid-search with 15% validation set accuracies on the training set

```
--------------- TOTAL ACCURACY -------------------
Accuracy is 80.85
********* CLASS BASED ACCURACY********************
Class 0
Accuracy is 80.71
Class 1
Accuracy is 81.01
```

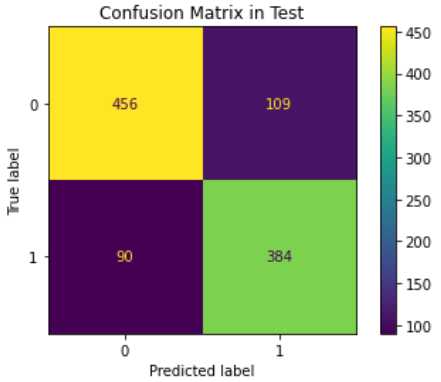


Fig. 36: Total and class-based accuracies on the best model with prune rate = 0.0 and "entropy" on the test set

Prune Rate: 0 Impurity: entropy : 82.29
Prune Rate: 0 Impurity: gini : 81.23
Prune Rate: 0.1 Impurity: entropy : 78.83
Prune Rate: 0.1 Impurity: gini : 58.33
Prune Rate: 0.2 Impurity: entropy : 78.83
Prune Rate: 0.2 Impurity: gini : 58.33
Prune Rate: 0.3 Impurity: entropy : 58.33
Prune Rate: 0.3 Impurity: gini : 58.33
Prune Rate: 0.7 Impurity: entropy : 58.33
Prune Rate: 0.7 Impurity: gini : 58.33
Prune Rate: 0.8 Impurity: entropy : 58.33
Prune Rate: 0.8 Impurity: gini : 58.33
Prune Rate: 0.9 Impurity: entropy : 58.33
Prune Rate: 0.9 Impurity: gini : 58.33
Total time for 7 x 2 grid search is: 175.54380583763123 seconds.

Fig. 37: Grid-search with 15% validation set on the training set

5.2.4 K-Nearest Neighbors (kNN) Algorithm with Validation Set

In this kNN, we used a validation set that accounts for the 15% of all the dataset to select the best hyperparameter combination, and evaluated the best model on the test data that accounts for the 15% of all the dataset. The train set corresponds to 70% of the dataset.. In Fig. 38, classification accuracies are shown with respect to **k** (number of nearest neighbors) and **distance metric**. In Fig. 40, a 15% validation set results are shown. ***Also, the total time for the grid-search is: 3.63 seconds.***

k	euclidian	manhattan
3	78.44	79.69
5	78.06	78.44
7	79.40	78.92
11	80.08	79.31
15	81.62	80.27

Fig. 38: Grid-search with 15% validation set accuracies on the training set

```
--------------- TOTAL ACCURACY ------------------
Accuracy is 78.44
********* CLASS BASED ACCURACY**********
Class 0
Accuracy is 88.5
Class 1
Accuracy is 66.46
```

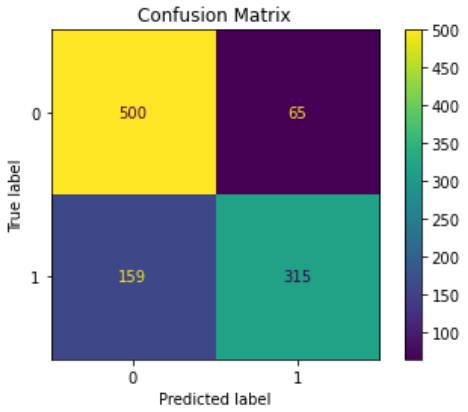


Fig. 39: Total and class-based accuracies on the best model with k = 15 and "euclidian" on the test set

K-Nearest Neighbor: 3 Metric: euclidian : 78.44
K-Nearest Neighbor: 3 Metric: manhattan : 79.69
K-Nearest Neighbor: 5 Metric: euclidian : 78.06
K-Nearest Neighbor: 5 Metric: manhattan : 78.44
K-Nearest Neighbor: 7 Metric: euclidian : 79.4
K-Nearest Neighbor: 7 Metric: manhattan : 78.92
K-Nearest Neighbor: 11 Metric: euclidian : 80.08
K-Nearest Neighbor: 11 Metric: manhattan : 79.31
K-Nearest Neighbor: 15 Metric: euclidian : 81.62
K-Nearest Neighbor: 15 Metric: manhattan : 80.27
Total time for 5 x 2 grid search is: 3.6341094970703125 seconds.

Fig. 40: Grid-search with 15% validation set on the training set

5.2.5 Multilayer Perceptron (MLP) Algorithm

Table 2: Model performance in "The Beirut Epileptic EEG Dataset" with respect to model structure and hyperparameters such as learning rate, activation functions in each layer, loss function, weight initialization, batch size, and epoch numbers

Layers	Activations	Learning R.	Loss	Initialization	Batch	Epoch	Val. Acc.	Test Acc.
[2]	Softmax	0.01	Cross Entropy	Xavier	100	1000	72.66	69.91
[2]	Softmax	0.01	MSE	Xavier	100	1000	57.80	55.82
[5,2]	[Sigmoid, Softmax]	0..05	MSE	Xavier	100	1000	58.98	56.68
[5,2]	[Sigmoid, Softmax]	0..05	Cross Entropy	Xavier	100	1000	79.37	77.57
[5,2]	[Tanh, Softmax]	0.05	Cross Entropy	Xavier	100	1000	80.39	78.44
[6,2]	[Tanh, Softmax]	0..05	Cross Entropy	Normal	100	500	79.20	77.09
[20,10,2]	[Tanh,ReLU,Softmax]	0.005	Cross Entropy	Xavier	100	5000	80.05	78.63
[20,10,2]	[Tanh,Tanh,Softmax]	0.005	Cross Entropy	Xavier	50	5000	80.6	79.30
[50,20,10,2]	[Tanh,Tanh,Tanh,Softmax]	0.005	Cross Entropy	Xavier	50	5000	80.15	79.4
[50,20,10,2]	[Tanh,Tanh,Tanh,Softmax]	0.005	Cross Entropy	Normal	50	5000	80.01	79.14

Similar to "**The Bonn EEG Time Series Dataset**", validation and test accuracies can be observed in Table 2 above. We followed a similar strategy that model performances are evaluated with respect to different loss, hidden size, initialization and network depth. As the network does not converge in early epochs, we set the epoch numbers high. The convergence in the training phase can be seen in Fig. 41.

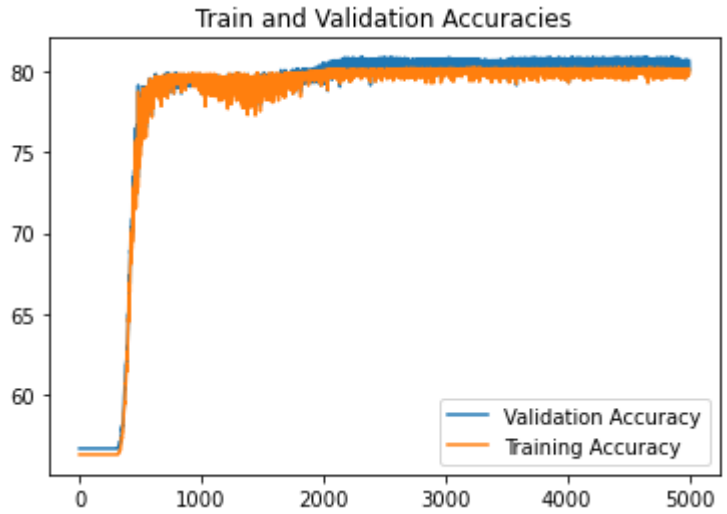


Fig. 41: Train and validation accuracies with respect to the epoch number

The best hyperparameters for this result above can be observed in the expression given below. As the model does not converge fastly in early epochs, we set the epoch numbers 5000 and visually observe the convergence. Here, we select the model that performs best in the validation set since the model converges. The test accuracy of this model is **79.30%**.

Selected model: [20,10,2], [Tanh,Tanh,Softmax], 0.005, Cross Entropy, Xavier, 100, 5000

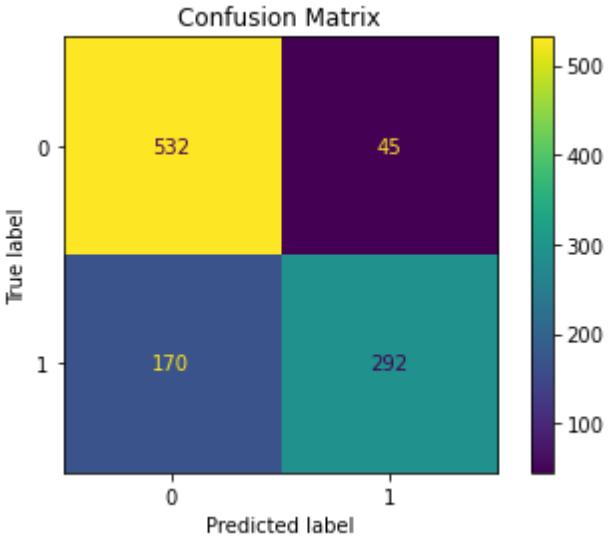


Fig. 42: Confusion matrix for the test set performance with respect to given model

6. Discussion on the Performance Results

For DT, we used 5-Fold cross-validation with a grid-search of pruning rate and impurity criterion firstly. Secondly, we used a separate validation set in order to select the best hyperparameters of pruning rate and impurity criterion. In our 5-Fold cross-validation implementation, we compared the information gained in a node with the pruning rate and splitting is continued according to their relative values. Results showed that the expectation of significant information gain in a node results in the early stopping of the tree building and underfitting of the model even in the training data. After exhaustively searching for the best classification performance, we tested the parameters of the best model under 5-Fold. We observed similar performance in test and training sets and concluded that our model is not overfitting. In both of our strategies, we observed that the classification accuracies were high (above 96%) for "**The Bonn EEG Time Series Dataset**" so that our DT model was successful in its classification. For "**The Beirut Epileptic EEG Dataset**", the accuracy was 76.61% (with the best parameters) with 5-Fold cross-validation and 80.85% (with the best parameters) with validation set hyperparameter tuning. The results are close to each other so that both strategies can be implemented for hyperparameter tuning. Noticeably, increasing the pruning rate negatively affected the performance results for the second dataset so lower pruning rates can be used for it. The classification accuracy results were lower for the second dataset as compared to the first dataset. The reason behind this can be identified as the class accuracy being low for class 1 as compared to class 0. Accoding to the literature review, there were observed to be some overlaps between the EEG data of the classes (as was also observed in data visualization) and this may have lowered the accuracies for both of the classes. However, the overall accuracies were still high so that our algorithms were successful. In order obtain classification results above 90%, more complex features in accordance with the nature of EEG data can be derived and/or more complex ML algorithms can be utilized (such as Recurrent Neural Networks (RNNs) and Convolutional Neural Networks (CNNs)).

For kNN, we applied the scheme in the training and test stages. It is a relatively fast algorithm compared to DT. In kNN, the distance metric does not significantly affect the performance. However, selecting the proper **k** (number of nearest neighbors) is crucial. In particular, small values of k show lower classification performance for the first dataset. In both of our strategies, we observed that the classification accuracies were high (above 96%, close to 97%) for "**The Bonn EEG Time Series Dataset**" so that our kNN model was successful in its classification. For "**The Beirut Epileptic EEG Dataset**", the accuracy was 77.96% (with the best parameters) with 5-Fold cross-validation and 78.44% (with the best parameters) with validation set hyperparameter tuning. The results are close to each other so that both strategies can be implemented for hyperparameter tuning. Noticeably, increasing the k value (the number of neighbors) positively affected the performance results for the

second dataset so higher k values can be used for it. The reason behind this is the increasing number of points that the algorithm checks to make its classification decision. The classification accuracy results were lower for the second dataset as compared to the first dataset. The reason behind this can be identified as the class accuracy again being low for class 1 as compared to class 0. According to the literature review, there were observed to be some overlaps between the EEG data of the classes (as was also observed in data visualization) and this may have lowered the accuracies for both of the classes. However, the overall accuracies were still high so that our algorithms were successful. In order to obtain classification results above 90%, the same methods explained above for the DT algorithm can be applied.

For MLP, we performed an extensive search using several models with different depth using various combinations of the hyperparameters such as learning rate, activation function, hidden unit size, batch size and weight initialization. In both of the datasets, we observe that the Xavier initialization yields better performance. In particular, in Beirute dataset, a light model with Xavier initialization remarkably outperforms the same model with normal weight initialization. The other observation is that even if the training process lasts longer with a small batch size, the model performance improves in both of the datasets. In "**The Bonn EEG Time Series Dataset**", with the best performing model, we obtain 96.72% and 96.75% accuracies in validation and test sets respectively. Although we obtain several models with better test performance, we select this model since it converges fast. In the "**The Beirut Epileptic EEG Dataset**", with the best performing model, we obtain 80.6% and 79.3% accuracies in validation and test sets respectively. We set high epoch numbers to ensure the model converges and visually observe the convergence process. Overall, we present the best performing models based on our exhaustive and extensive search.

7. Revisions Performed According to the Feedback

According to the received feedback, more explanations were included regarding our thought processes behind our choices for the ML models for our chosen datasets. These were included in the "Implemented Machine Learning Approaches" where brief descriptions about the ML models were included for the comprehensiveness and clarity of our final report. Moreover, the dataset splits were changed. The k-Fold cross-validation strategy for hyperparameter tuning was kept the same for kNN and DT algorithms as in the interim report although the split of train and test sets were changed in accordance with the feedback. The ratio of the test set was chosen as 15% and the ratio of the train test was chosen as 85%. The validation sets were chosen as parts of train set that have ratios of approximately 15% in accordance with the chosen 5-Fold cross-validation. Furthermore, another validation strategy was introduced in accordance with the requested revision. Three splits were performed as 70% train set, 15% test set, and 15% validation set. As for the MLP algorithm, we splitted the test set with the ratio of 15% and we used the rest of the data for both datasets for training and hyperparameter tuning. The validation sets separated in all the epochs were splitted in a ratio of 15%. Hence, we performed the dataset splits for both datasets according to the ratios given in the feedback. Importantly, we made sure that the test set contains the same samples for all three models in both validation strategies that are 5-Fold cross-validation and validation set strategies. In addition, we adde mean and variance as features to both datasets. We included more statistical features such as skewness and kurtosis to account for the statistical properties of both datasets. Lastly, we listed our hyperparameters for each of our methods in their relevant sections and used validation set(s) in different strategies to tune them.

8. Observed Challenges

In the first half of the project, the observed challenges were regarding the chosen dataset, and the created and simulated ML models. The challenge regarding the dataset was related to its high-volume structure. Since there were 11,500 rows corresponding to the same number of subjects and a total of 178 features, the training duration for the dataset was relatively long at the beginning. This challenge was solved by the application of feature engineering and data preprocessing. Moreover, the challenges regarding the ML algorithms were related to writing them from scratch for

the project. In general, it was observed that the ML tools and functions of the libraries such as Sklearn were fast due to their optimized and fast working principles that happen as background processes. Although implementing the kNN architecture was easier compared to implementing the DT architecture, it was seen that writing "for" loops made the Python codes very slow and inefficient. One of the reasons behind the long duration of the training processes was the cumbersome codes. In order to overcome this challenge, the ML algorithms were tried to be implemented by vectorizing the Python codes as much as possible as well as trying to recursive functions instead of loop structures.

For the second part of the project, the observed challenges are regarding the third ML algorithm which is MLP, and the second dataset that was introduced and used in the project. The training duration of the MLP algorithm was a challenge since the high-volume datasets introduced a great number of neural network layers that can slow down the working processes of the Python codes. Since we ran our codes in a large number of epochs and wrote a generic MLP to which any number of layers can be added, designing and implementing the MLP algorithm was challenging. Implementing the backpropagation architecture was also challenging since it was a costly, both in means of time and memory, operation that consists of taking back-to-back derivatives. Most of the time dimension errors were received and they were corrected. Moreover, organizing and utilizing "**The Beirut Epileptic EEG Dataset**" of Beirut Medical Center was a challenge since it is again a high-volume dataset and its accuracy results were not as good as the results of the current dataset although they were in an acceptable range. It required a great extent of feature engineering, data preprocessing, and data organization. The expected challenges were solved by applying similar strategies which are applying data visualization, feature engineering, and data preprocessing for the high-volume dataset and implementing the third ML algorithm as efficiently as possible again by vectorizing the codes and refraining from long-ranged loop structures.

9. Gantt Chart for Workload

The Gantt chart showing the planned and completed workload can be observed in the figure below.

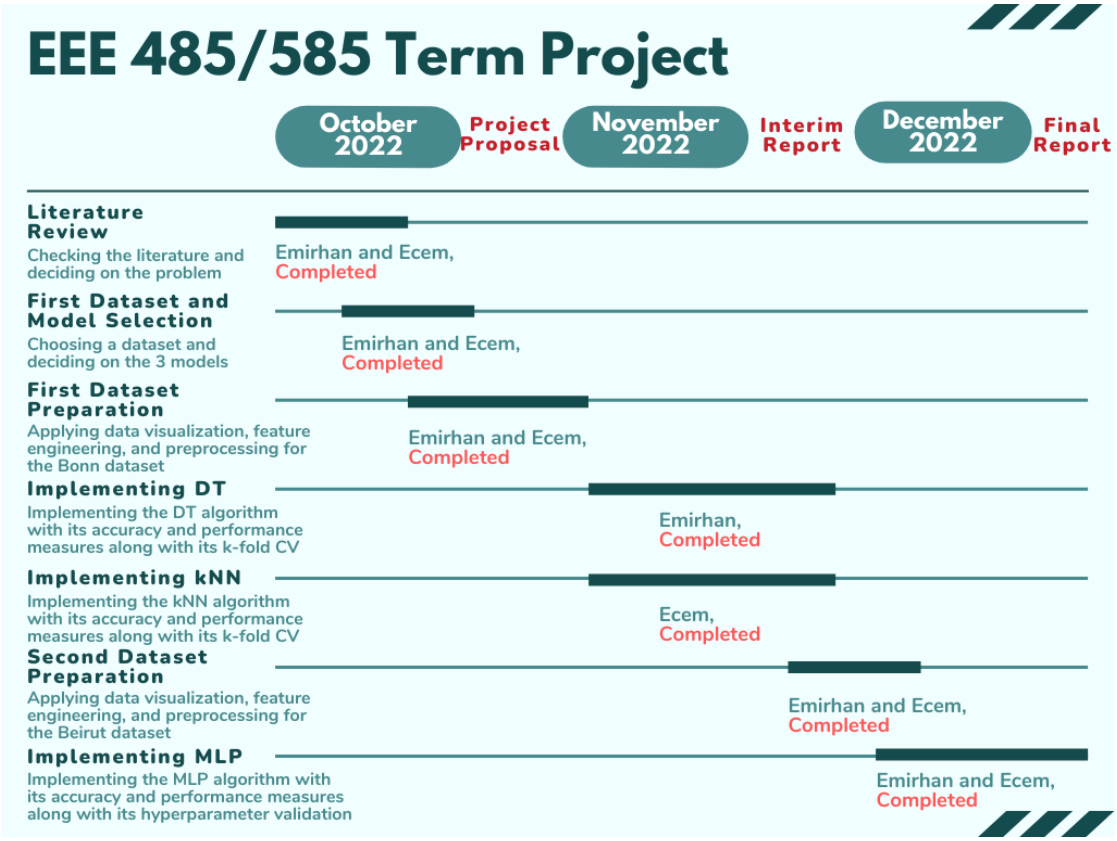


Fig. 43: The Gantt chart

10. Conclusion

In conclusion, three ML models were trained, tested, and validated by using "**The Bonn EEG Time Series Dataset**" and "**The Beirut Epileptic EEG Dataset**" with the purpose of performing a binary classification task for the detection of "epileptic seizure" and "healthy" (epileptic seizure-free) conditions in the project. For the utilized dataset, data visualization, feature engineering, and data preprocessing (z-score/min-max normalization, shuffling, and train-test/train-test-validation set splitting) were performed by implementing generic Python functions. The train and test sets (and in the second strategy separate validation sets) were then fed into the three ML models which are DT, kNN and MLP algorithms. These models were implemented in Python from scratch and their performance results were checked by observing their overall prediction accuracies for three models as well as the class accuracies for 0 and 1 for two models. Afterward, k-Fold cross-validation was applied to DT and kNN models in order to perform a grid-search for hyperparameter tuning (for choosing the optimum pruning rates and impurity functions for DT and for choosing the optimum k and distance metrics for kNN). Moreover, 15% validation set strategy was applied to these two models to again perform a grid-search for hyperparameter tuning according to the given feedback. For the MLP algorithm, the 15% validation sets splitted in the training of each epoch were utilized in hyperparameter tuning considering that it has a different structure as compared to the other two algorithms. The hyperparameters for the MLP algorithm were listed in its respective sections. Both the k-Fold cross-validation functions and the grid-search architectures were written from scratch. The best hyperparameters for all ML models were obtained and under these parameters, it was observed that the accuracy results for the models were around 95-96% for the first dataset. Importantly, the class accuracies were close to each other which verified that the models were not overfitting to the dataset although they have very high accuracies. For the second dataset, it was observed that the accuracy results for the models were around 75-80% under the best hyperparameters. The class accuracies were not very close to each other except for some simulations of the DT model. This reason behind this result could be due to the class overlaps explained by the studies that utilized the second dataset. Furthermore, the confusion matrices were plotted for DT and kNN models as a means of an accuracy representation on a graphical structure. Moreover, the training durations of the models were very short (less than 5 minutes) for DT and kNN algorithms. For these models, the grid-serach for hyperparameter tuning took a significant amount of time since the algorithms were ran multiple times in grid-search scenarios. In the MLP model, choosing low learning rates, high number of layers, and high number of dimensions causes the algorithm to last for long durations. On the other hand, when the batch size was chosen as high, the model was faster. Overall, it can be concluded that the MLP model works fast considering that the it is written in a vectorized manner so the number of epochs is chosen as high as a result. Importantly, we still the check the convergence whilst the epoch number is increased. All of these results showed that the feature engineering and data preprocessing procedures, and the model implementations were very successful. Finally, two datasets were preprocessed and applied to three different ML algorithms by means of training, testing, and validating them with the aim of binary classification for the detection of epileptic seizures at the end of the project as proposed at the beginning of the semester.

11. References

[1] "Epilepsy," mayoclinic.org. https://www.mayoclinic.org/diseases-conditions/epilepsy/symptoms-causes/syc-20350093 (accessed Nov. 19, 2022).

[2] E. Ş, and E. Koç, "Classification of Epileptic Seizure From Electroencephalogram (EEG) Signals Based on Machine Learning Approaches - Term Project Proposal," Bilkent University, Ankara, Turkey, 2022. Accessed on: Nov. 19, 2022. [Online].

[3] "Electroencephalogram (EEG)," hopkinsmedicine.org. https://www.hopkinsmedicine.org/health/treatment-tests-and-therapies/electroencephalogram-eeg#:~:t

ext=The%20electrodes%20detect%20tiny%20electrical,provider%20then%20interprets%20the%20reading. (accessed Nov. 19, 2022).

[4] I. Ahmad, X. Wang, M. Zhu, C. Wang, Y. Pi, J. A. Khan, S. Khan, O. W. Samuel, S. Chen, and G. Li, "EEG-based epileptic seizure detection via machine/Deep learning approaches: A systematic review," *Computational Intelligence and Neuroscience*, vol. 2022, pp. 1–20, 2022.

[5] M. Rashed-Al-Mahfuz, M. A. Moni, S. Uddin, S. A. Alyami, M. A. Summers, and V. Eapen, "A deep convolutional neural network method to detect seizures and characteristic frequencies using epileptic electroencephalogram (EEG) data," *IEEE Journal of Translational Engineering in Health and Medicine*, vol. 9, pp. 1–12, 2021.

[6] K. AlSharabi, S. Ibrahim, R. Djemal and A. Alsuwailem, "A DWT-entropy-ANN based architecture for epilepsy diagnosis using EEG signals," *2016 2nd International Conference on Advanced Technologies for Signal and Image Processing (ATSIP)*, pp. 288-291, 2016.

[7] H. R. A. Ghayab, Y. Li, S. Abdulla, M. Diykh, and X. Wan, "Classification of epileptic EEG signals based on simple random sampling and sequential feature selection," *Brain Informatics*, vol. 3, no. 2, pp. 85-91, 2016.

[8] S. Kumar, R. R. Janghel, and S. P. Sahu "Classification of EEG Signals for Detection of Epileptic Seizure Using Restricted Boltzmann Machine Classifier," in Data Mining and Machine Learning Applications. R. Raja, K. K. Nagwanshi, S. Kumari and K. R. Laxmi, Ed. New Jersey: John Wiley & Sons, 2022, pp. 397–421. Accessed: Nov. 19, 2022. [Online]. Available: https://onlinelibrary.wiley.com/doi/10.1002/9781119792529.ch15

[9] R. G. Andrzejak, K. Lehnertz, F. Mormann, C. Rieke, P. David, and C. E. Elger, "Indications of nonlinear deterministic and finite-dimensional structures in time series of brain electrical activity: Dependence on recording region and Brain State," *Physical Review E*, vol. 64, no. 6, 2001.

[10] P. Handa, M. Mathur, and N. Goel, "Open and free EEG datasets for epilepsy diagnosis," *arXiv*, 2021.

[11] S. Ibrahim, R. Djemal, and A. Alsuwailem, "Electroencephalography (EEG) signal processing for epilepsy and autism spectrum disorder diagnosis," *Biocybernetics and Biomedical Engineering*, vol. 38, no. 1, pp. 16-26, 2018.

[12] Ö.Türk, and M. S. Özerdem, "Epilepsy Detection by Using Scalogram Based Convolutional Neural Network from EEG Signals," *Brain Sciences*, vol. 9, no. 5, pp. 1-16, 2019.

[13] H. Al-Hadeethi, S. Abdulla, M. Diykh, R. C. Deo, and J. H. Green, "Adaptive boost LS-SVM classification approach for time-series signal classification in epileptic seizure diagnosis applications," *Expert Systems with Applications*, vol. 161, pp. 1-14, 2020.

[14] T. G. Altundoğan and M. Karaköse, "EEG Signal Classification with Deep Neural Networks using Visibility Graphs," *2022 26th International Conference on Information Technology (IT)*, pp. 1-4, 2022.

[15] P. Mathur and V. K. Chakka, "Graph Signal Processing of EEG signals for Detection of Epilepsy," *2020 7th International Conference on Signal Processing and Integrated Networks (SPIN)*, pp. 839-843, 2020.

[16] "Epileptic seizures dataset," kaggle.com. https://www.kaggle.com/datasets/chaditya95/epileptic-seizures-dataset (accessed Oct. 21, 2022).

[17] W. Nasreddine, “Epileptic EEG Dataset,” data.mendeley.com. https://data.mendeley.com/datasets/5pc2j46cbc/1 (accessed Nov. 14, 2022).

[18] H. Shayeste, and B. M. Asl, “Heterogeneous Recurrence Analysis of imaged-EEG for spatio-temporal epileptic seizure detection,” *Sensors*, vol. 22, no. 7, 2022.

[19] S. Y. Shah, H. Larijani, F. Mormann, R. M. Gibson, and D. Liarokapis, “Random Neural Network Based Epileptic Seizure Episode Detection Exploiting Electroencephalogram Signals,” *IEEE Journal of Biomedical and Health Informatics*, 2022.

[20] *Bilkent.edu.tr*, 2022. http://www.cs.bilkent.edu.tr/~s.rahimzadeh/CS550.html (accessed Nov. 20, 2022).

[21] “KNN Classification Tutorial using Sklearn Python,” www.datacamp.com. https://www.datacamp.com/tutorial/k-nearest-neighbor-classification-scikit-learn7 (accessed Nov. 4, 2022)

[22] H. Wang, R. Czerminski, and A. C. Jamieson, “Neural Networks and Deep Learning,” *The Machine Age of Customer Insight*, pp. 91–101, 2021.

Decision_Tree_Bonn_KFold

December 19, 2022

[1]:
```
import pandas as pd
from math import *
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import ConfusionMatrixDisplay
from sklearn.metrics import confusion_matrix
from sklearn.metrics import plot_confusion_matrix
from random import sample
import utils
import os
```

[2]:
```
#loading the data
X_train = np.load("X_train.npy")
X_test = np.load("X_test.npy")

y_train = np.load("y_train.npy")
y_test = np.load("y_test.npy")
```

[3]:
```
#loading the data
np_train_data = np.load("np_train_data.npy")
np_test_data = np.load("np_test_data.npy")

np_label_train = np.load("np_label_train.npy")
np_label_test = np.load("np_label_test.npy")
```

[4]:
```
#Define some functions
def impurityCalculation(impurity_choice, labelsOfnode):

    labels, numLabels = np.unique(labelsOfnode, return_counts = True)
    total_labels= np.sum(numLabels)
    p_numLabels = numLabels/total_labels

    if impurity_choice == "entropy":
        entropy = 0
        total_entropy = np.sum([entropy+ (-np.log2(ii)*ii)  for ii in␣
↪p_numLabels])
        total_impurity = total_entropy
```

```
    elif impurity_choice == "gini":
        entropy = 0
        gini= (1/2)*(1-np.sum(np.square(p_numLabels)))
        total_impurity = gini

    else:
        raise Exception("Sorry", impurity_choice, "is not an impurity type. ")

    return total_impurity

def LeftRightSplit(attribute, label, value,impurity_choice):


    total_entropy = impurityCalculation(impurity_choice,label)

    R_split = label[attribute>value]
    L_split = label[attribute<=value]
    L_prop= len(L_split)/len(label)
    R_prop= 1- L_prop

    L_impurity = impurityCalculation(impurity_choice,L_split)
    R_impurity = impurityCalculation(impurity_choice,R_split)

    split_impurity = L_prop*L_impurity + R_prop*R_impurity
    informationGain= total_entropy-split_impurity

    return split_impurity, informationGain


def exhaustive_search(attributes,label,impurity_choice):

    splitCheckAll= np.Inf
    gainAll=0
    valueSplitAll=0

    featureIndexSplit=0

    for ft in range(0,attributes.shape[1]):

        splitCheck= np.Inf

        for val in attributes[:,ft]:
            isSplit,gainCheck = LeftRightSplit(attributes[:
 ↪,ft],label,val,impurity_choice)

            if isSplit < splitCheck:
```

```
                splitCheck=isSplit
                tempSplitCheck=splitCheck
                splitValue= val
                gainSplit= gainCheck


        if tempSplitCheck < splitCheckAll:
            splitCheckAll=tempSplitCheck
            featureIndexSplit=ft
            valueSplitAll= splitValue
            gainAll=gainSplit


    return splitCheckAll,gainAll,valueSplitAll,featureIndexSplit


def splitNode(attributes,label,impurity_choice):

    bestSplitImpurity, bestSplitInfoGain,bestValue,bestFeatureIndex =␣
 ↪exhaustive_search(attributes,label,impurity_choice)

    node_impurity = impurityCalculation(impurity_choice,label)

    left_node = attributes[attributes[:,bestFeatureIndex]<=bestValue]
    left_node_labels = label[attributes[:,bestFeatureIndex]<=bestValue]


    right_node = attributes[attributes[:,bestFeatureIndex]>bestValue]
    right_node_labels = label[attributes[:,bestFeatureIndex]>bestValue]


    return left_node,left_node_labels, right_node,right_node_labels,␣
 ↪node_impurity,bestSplitImpurity, bestSplitInfoGain,bestValue,bestFeatureIndex
```

[5]:
```
class Node():
    def __init__(self,parent,depth):

        self.parent = parent
        self.depth=depth
        self.childen = []

        self.sampleNumber = None
        self.labelNumbers = []
        self.labelNames= []
        self.bestSplitValue= None
        self.bestSplitIndex= None
        self.splitImpurity = None
```

```
        self.isLeaf = 0
        self.leafLabelName= None


def Tree(attributes,label,node,prun,impurity_choice):

    node_labels, labelNumbers = np.unique(label,return_counts= True)
    node.labelNames= node_labels
    node.labelNumbers=labelNumbers
    node.sampleNumber= np.sum(labelNumbers)

    if len(node_labels)==1: # it is pure
        node.isLeaf=1
        node.leafLabelName=node_labels[0]
        return


    else :
        left_node_Feat,left_node_labels, right_node_Feat,right_node_labels,␣
↪node_impurity,bestSplitImpurity,\
        bestSplitInfoGain,bestValue,bestFeatureIndex =␣
↪splitNode(attributes,label,impurity_choice)
        #Preprunning
        if bestSplitInfoGain <= prun:

            labelname,labelNum= np.unique(label, return_counts=True)
            node.leafLabelName= labelname[np.argmax(labelNum)]
            node.isLeaf = 1
            return
        # You can also add another pruning methods

        node.bestSplitValue=bestValue

        node.bestSplitIndex= bestFeatureIndex
        node.splitImpurity=bestSplitImpurity


        node.L_child= Node(node,node.depth+1)
        node.R_child= Node(node,node.depth+1)
        Tree(left_node_Feat,left_node_labels,node.L_child, prun,impurity_choice)
        Tree(right_node_Feat,right_node_labels,node.R_child,prun,impurity_choice)



def TraverseTree(node,data):
```

```
    if node.isLeaf==1:
        prediction = node.leafLabelName

    else:
        if data[node.bestSplitIndex] <= node.bestSplitValue:
            prediction = TraverseTree(node.L_child,data)
        else:
            prediction = TraverseTree(node.R_child,data)
    return prediction
```

[6]:
```
class DecisionTreeClassifier():

    def __init__(self,criterion,isPrunned="yes"):
        self.beginTree= None
        self.impurity_choice=criterion
        self.isPrunned=isPrunned


    def fit(self,data,label,prun):

        rootNode = Node(None,0)
        if self.isPrunned == "yes":
            Tree(data,label,rootNode,prun,self.impurity_choice)

        else:
            Tree(data,label,rootNode,0,self.impurity_choice)

        self.beginTree=rootNode

    def predict(self,data):

        if self.beginTree==None:
            print(" You need to create a tree !")

        else:
            prediction_list =[]
            for ii in range(0,data.shape[0]):
                prediction_list.append(TraverseTree(self.beginTree,data[ii,:]))

            prediction = np.asarray(prediction_list)
            return prediction
```

[7]:
```
def test_accuracy(test_label,predicted_label):

  print("Accuracy is", round(np.mean(test_label==predicted_label)*100,2))
  return round(100*np.mean(test_label==predicted_label),2)
```

```
def class_accuracy(model,data,label):
   class_labels = np.unique(label,return_counts=False)
   class_acc_list = []
   for cl in class_labels:
        print("Class", cl)
        class_pred= model.predict(data[label==cl])
        class_acc= test_accuracy(label[label==cl],class_pred)
        #print("Test accuracy for class ", cl, "is : ", class_acc )
        class_acc_list.append(class_acc)


def confusion_matrix_plot(true_label,predictions):


    class_labels = np.unique(true_label,return_counts=False)

    cm= confusion_matrix(true_label,predictions)
    cp=ConfusionMatrixDisplay(cm,display_labels=class_labels)
    cp.plot()
    plt.title("Confusion Matrix in Test")
    plt.show()

  # Confusion Matrix Creation
    '''
    cm= confusion_matrix(label,preds)
    cp=ConfusionMatrixDisplay(cm,display_labels=class_labels)
    cp.plot()
    plt.title(mode + " Confusion Matrix")
    plt.show()
    '''



# Here I will implement graph plotting
```

[8]:
```
def KFoldCrossValidation(data_train,label_train,model="decision_tree", Kfold=5,␣
↪impurity="entropy", prune_rate=0.1):
    sample_count= data_train.shape[0]
    fold_size = int(sample_count/Kfold)
    accuracy_all = list()

    for ff in range(Kfold):

        test_data=data_train[ff*fold_size:(ff+1)*fold_size,:]
        test_label= label_train[ff*fold_size:(ff+1)*fold_size]
```

```
        train_data= np.append(data_train[:ff*fold_size,:
↪],data_train[(ff+1)*fold_size:,:],axis=0)
        train_label= np.append(label_train[:
↪ff*fold_size],label_train[(ff+1)*fold_size:],axis=0)


        decision_tree= DecisionTreeClassifier(impurity,"yes")
        decision_tree.fit( train_data,train_label,prune_rate)

        preds = decision_tree.predict(test_data)
        acc= np.mean(preds==test_label)
        accuracy_all.append(acc)
        print("Fold ",ff+1,"is completed.")
    return sum(accuracy_all)/len(accuracy_all)*100
```

[9]:
```
# K-fold cross validation with grid-search
grid_search = [[0,0.1,0.2,0.3,0.7,0.8,0.9],["entropy","gini"]]
accuracy_grid=np.zeros((len(grid_search[0]),len(grid_search[1])))
K_fold=5
import time

b=time.time()
for i,k in enumerate(grid_search[0]):
    for j,m in enumerate(grid_search[1]):
        acc=KFoldCrossValidation(X_train,y_train,Kfold=5,prune_rate=k,impurity=m)
        print("Prune Rate:", k, " Impurity: ", m, ": ",round(acc,2))
        accuracy_grid[i][j]=round(acc,2)
e=time.time()
print("Total time for ",K_fold, "-fold is: ", e-b, "seconds.")
def organize_results(grid_search,accuracy_grid):

    df=pd.DataFrame(accuracy_grid)
    df.columns = grid_search[1]
    df.set_index(pd.Index(grid_search[0]),inplace=True)
    df.index.name="impurity"
    return df
```

```
Fold  1 is completed.
Fold  2 is completed.
Fold  3 is completed.
Fold  4 is completed.
Fold  5 is completed.
Prune Rate: 0  Impurity:  entropy :  95.76
Fold  1 is completed.
Fold  2 is completed.
Fold  3 is completed.
Fold  4 is completed.
Fold  5 is completed.
```

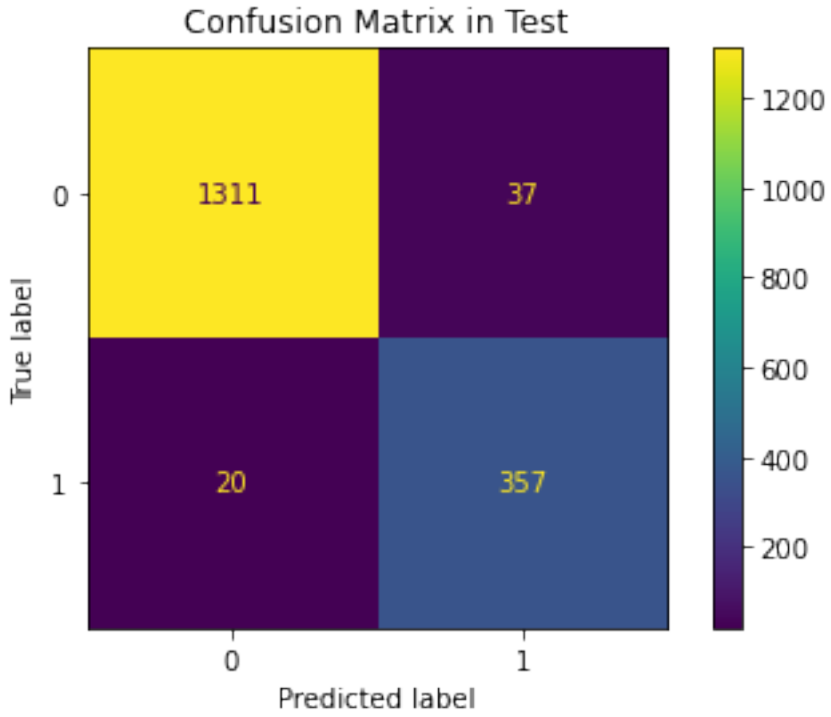
```
Prune Rate: 0  Impurity:  gini :  95.74
Fold  1 is completed.
Fold  2 is completed.
Fold  3 is completed.
Fold  4 is completed.
Fold  5 is completed.
Prune Rate: 0.1  Impurity:  entropy :  95.38
Fold  1 is completed.
Fold  2 is completed.
Fold  3 is completed.
Fold  4 is completed.
Fold  5 is completed.
Prune Rate: 0.1  Impurity:  gini :  96.02
Fold  1 is completed.
Fold  2 is completed.
Fold  3 is completed.
Fold  4 is completed.
Fold  5 is completed.
Prune Rate: 0.2  Impurity:  entropy :  95.43
Fold  1 is completed.
Fold  2 is completed.
Fold  3 is completed.
Fold  4 is completed.
Fold  5 is completed.
Prune Rate: 0.2  Impurity:  gini :  80.33
Fold  1 is completed.
Fold  2 is completed.
Fold  3 is completed.
Fold  4 is completed.
Fold  5 is completed.
Prune Rate: 0.3  Impurity:  entropy :  95.43
Fold  1 is completed.
Fold  2 is completed.
Fold  3 is completed.
Fold  4 is completed.
Fold  5 is completed.
Prune Rate: 0.3  Impurity:  gini :  80.33
Fold  1 is completed.
Fold  2 is completed.
Fold  3 is completed.
Fold  4 is completed.
Fold  5 is completed.
Prune Rate: 0.7  Impurity:  entropy :  80.33
Fold  1 is completed.
Fold  2 is completed.
Fold  3 is completed.
Fold  4 is completed.
Fold  5 is completed.
```

```
Prune Rate: 0.7  Impurity:  gini :  80.33
Fold  1 is completed.
Fold  2 is completed.
Fold  3 is completed.
Fold  4 is completed.
Fold  5 is completed.
Prune Rate: 0.8  Impurity:  entropy :  80.33
Fold  1 is completed.
Fold  2 is completed.
Fold  3 is completed.
Fold  4 is completed.
Fold  5 is completed.
Prune Rate: 0.8  Impurity:  gini :  80.33
Fold  1 is completed.
Fold  2 is completed.
Fold  3 is completed.
Fold  4 is completed.
Fold  5 is completed.
Prune Rate: 0.9  Impurity:  entropy :  80.33
Fold  1 is completed.
Fold  2 is completed.
Fold  3 is completed.
Fold  4 is completed.
Fold  5 is completed.
Prune Rate: 0.9  Impurity:  gini :  80.33
Total time for  5 -fold is:  5624.085533618927 seconds.
```

[10]:
```
# Print the model results

accuracy_table= organize_results(grid_search,accuracy_grid)
import dataframe_image as dfi
dfi.export(accuracy_table, "decision_gridsearch.png")
```

[11]:
```
# Automatically select the best model

from numpy import unravel_index
best_index=unravel_index(accuracy_grid.argmax(),  accuracy_grid.shape)
decision_tree_best= DecisionTreeClassifier(grid_search[1][best_index[0]],"yes")
decision_tree_best.
 ↪fit(np_train_data,np_label_train,grid_search[0][best_index[1]])
y_pred= decision_tree_best.predict(np_test_data)

print("--------------- TOTAL ACCURACY -------------------")
acc= test_accuracy(np_label_test,y_pred)
print("********* CLASS BASED ACCURACY*********************")
class_accuracy(decision_tree_best,np_test_data,np_label_test)
confusion_matrix_plot(np_label_test,y_pred)
```

```
--------------- TOTAL ACCURACY ------------------
Accuracy is 96.7
********* CLASS BASED ACCURACY********************
Class 0
Accuracy is 97.26
Class 1
Accuracy is 94.69
```



[ ]:

Decision_Tree_Bonn_Val

December 19, 2022

[1]:
```
import pandas as pd
from math import *
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import ConfusionMatrixDisplay
from sklearn.metrics import confusion_matrix
from sklearn.metrics import plot_confusion_matrix
from random import sample
import os
import utils
```

[2]:
```
#loading the data
X_train = np.load("X_train.npy")
X_test = np.load("X_test.npy")
X_val = np.load("X_val.npy")

y_train = np.load("y_train.npy")
y_test = np.load("y_test.npy")
y_val = np.load("y_val.npy")
```

[3]:
```
#loading the data
np_train_data = np.load("np_train_data.npy")
np_test_data = np.load("np_test_data.npy")
np_val_data = np.load("np_val_data.npy")

np_label_train = np.load("np_label_train.npy")
np_label_test = np.load("np_label_test.npy")
np_label_val = np.load("np_label_val.npy")
```

[4]:
```
#Define some functions
def impurityCalculation(impurity_choice, labelsOfnode):

    labels, numLabels = np.unique(labelsOfnode, return_counts = True)
    total_labels= np.sum(numLabels)
    p_numLabels = numLabels/total_labels

    if impurity_choice == "entropy":
        entropy = 0
```

```
        total_entropy = np.sum([entropy+ (-np.log2(ii)*ii)  for ii in␣
↪p_numLabels])
        total_impurity = total_entropy


    elif impurity_choice == "gini":
        entropy = 0
        gini= (1/2)*(1-np.sum(np.square(p_numLabels)))
        total_impurity = gini

    else:
        raise Exception("Sorry", impurity_choice, "is not an impurity type. ")

    return total_impurity

def LeftRightSplit(attribute, label, value,impurity_choice):


    total_entropy = impurityCalculation(impurity_choice,label)

    R_split = label[attribute>value]
    L_split = label[attribute<=value]
    L_prop= len(L_split)/len(label)
    R_prop= 1- L_prop

    L_impurity = impurityCalculation(impurity_choice,L_split)
    R_impurity = impurityCalculation(impurity_choice,R_split)

    split_impurity = L_prop*L_impurity + R_prop*R_impurity
    informationGain= total_entropy-split_impurity

    return split_impurity, informationGain


def exhaustive_search(attributes,label,impurity_choice):

    splitCheckAll= np.Inf
    gainAll=0
    valueSplitAll=0

    featureIndexSplit=0

    for ft in range(0,attributes.shape[1]):

        splitCheck= np.Inf

        for val in attributes[:,ft]:
```

```
            isSplit,gainCheck = LeftRightSplit(attributes[:
↪,ft],label,val,impurity_choice)

            if isSplit < splitCheck:
                splitCheck=isSplit
                tempSplitCheck=splitCheck
                splitValue= val
                gainSplit= gainCheck


        if tempSplitCheck < splitCheckAll:
            splitCheckAll=tempSplitCheck
            featureIndexSplit=ft
            valueSplitAll= splitValue
            gainAll=gainSplit


    return splitCheckAll,gainAll,valueSplitAll,featureIndexSplit


def splitNode(attributes,label,impurity_choice):

    bestSplitImpurity, bestSplitInfoGain,bestValue,bestFeatureIndex =␣
↪exhaustive_search(attributes,label,impurity_choice)

    node_impurity = impurityCalculation(impurity_choice,label)

    left_node = attributes[attributes[:,bestFeatureIndex]<=bestValue]
    left_node_labels = label[attributes[:,bestFeatureIndex]<=bestValue]


    right_node = attributes[attributes[:,bestFeatureIndex]>bestValue]
    right_node_labels = label[attributes[:,bestFeatureIndex]>bestValue]


    return left_node,left_node_labels, right_node,right_node_labels,␣
↪node_impurity,bestSplitImpurity, bestSplitInfoGain,bestValue,bestFeatureIndex
```

[5]:
```
class Node():
    def __init__(self,parent,depth):

        self.parent = parent
        self.depth=depth
        self.childen = []

        self.sampleNumber = None
        self.labelNumbers = []
```

```
        self.labelNames= []
        self.bestSplitValue= None
        self.bestSplitIndex= None
        self.splitImpurity = None
        self.isLeaf = 0
        self.leafLabelName= None


def Tree(attributes,label,node,prun,impurity_choice):

    node_labels, labelNumbers = np.unique(label,return_counts= True)
    node.labelNames= node_labels
    node.labelNumbers=labelNumbers
    node.sampleNumber= np.sum(labelNumbers)

    if len(node_labels)==1: # it is pure
        node.isLeaf=1
        node.leafLabelName=node_labels[0]
        return


    else :
        left_node_Feat,left_node_labels, right_node_Feat,right_node_labels,␣
 ↪node_impurity,bestSplitImpurity,\
        bestSplitInfoGain,bestValue,bestFeatureIndex =␣
 ↪splitNode(attributes,label,impurity_choice)
        #Preprunning
        if bestSplitInfoGain <= prun:

            labelname,labelNum= np.unique(label, return_counts=True)
            node.leafLabelName= labelname[np.argmax(labelNum)]
            node.isLeaf = 1
            return
        # You can also add another pruning methods

        node.bestSplitValue=bestValue

        node.bestSplitIndex= bestFeatureIndex
        node.splitImpurity=bestSplitImpurity


        node.L_child= Node(node,node.depth+1)
        node.R_child= Node(node,node.depth+1)
        Tree(left_node_Feat,left_node_labels,node.L_child, prun,impurity_choice)
        Tree(right_node_Feat,right_node_labels,node.R_child,prun,impurity_choice)
```

```
def TraverseTree(node,data):

    if node.isLeaf==1:
        prediction = node.leafLabelName

    else:
        if data[node.bestSplitIndex] <= node.bestSplitValue:
            prediction = TraverseTree(node.L_child,data)
        else:
            prediction = TraverseTree(node.R_child,data)
    return prediction
```

[6]:
```
class DecisionTreeClassifier():

    def __init__(self,criterion,isPrunned="yes"):
        self.beginTree= None
        self.impurity_choice=criterion
        self.isPrunned=isPrunned


    def fit(self,data,label,prun):

        rootNode = Node(None,0)
        if self.isPrunned == "yes":
            Tree(data,label,rootNode,prun,self.impurity_choice)

        else:
            Tree(data,label,rootNode,0,self.impurity_choice)

        self.beginTree=rootNode

    def predict(self,data):

        if self.beginTree==None:
            print(" You need to create a tree !")

        else:
            prediction_list =[]
            for ii in range(0,data.shape[0]):
                prediction_list.append(TraverseTree(self.beginTree,data[ii,:]))

            prediction = np.asarray(prediction_list)
            return prediction
```

[7]:
```
def test_accuracy(test_label,predicted_label):

  print("Accuracy is", round(np.mean(test_label==predicted_label)*100,2))
  return round(100*np.mean(test_label==predicted_label),2)

def class_accuracy(model,data,label):
   class_labels = np.unique(label,return_counts=False)
   class_acc_list = []
   for cl in class_labels:
        print("Class", cl)
        class_pred= model.predict(data[label==cl])
        class_acc= test_accuracy(label[label==cl],class_pred)
        #print("Test accuracy for class ", cl, "is : ", class_acc )
        class_acc_list.append(class_acc)


def confusion_matrix_plot(true_label,predictions):


    class_labels = np.unique(true_label,return_counts=False)

    cm= confusion_matrix(true_label,predictions)
    cp=ConfusionMatrixDisplay(cm,display_labels=class_labels)
    cp.plot()
    plt.title("Confusion Matrix in Test")
    plt.show()

  # Confusion Matrix Creation
    '''
    cm= confusion_matrix(label,preds)
    cp=ConfusionMatrixDisplay(cm,display_labels=class_labels)
    cp.plot()
    plt.title(mode + " Confusion Matrix")
    plt.show()
    '''



# Here I will implement graph plotting
```

[8]:
```
# grid-search
grid_search = [[0,0.1,0.2,0.3,0.7,0.8,0.9],["entropy","gini"]]
accuracy_grid=np.zeros((len(grid_search[0]),len(grid_search[1])))
prune_rate = len(grid_search[0])
impurity = len(grid_search[1])
accuracy_all = list()
import time
```

```
b=time.time()
for i,k in enumerate(grid_search[0]):
    for j,m in enumerate(grid_search[1]):
        decision_tree= DecisionTreeClassifier(m,"yes")
        decision_tree.fit(X_train,y_train,k)
        preds = decision_tree.predict(X_val)
        acc = np.mean(preds == y_val)
        accuracy_all.append(acc)
        acc_percent = round(100*acc,2)
        print("Prune Rate:", k, " Impurity: ", m, ": ",acc_percent)
        accuracy_grid[i][j]=acc_percent
e=time.time()
print("Total time for", prune_rate, "x", impurity, "grid search is: ", e-b,␣
 ↪"seconds.")
def organize_results(grid_search,accuracy_grid):

    df=pd.DataFrame(accuracy_grid)
    df.columns = grid_search[1]
    df.set_index(pd.Index(grid_search[0]),inplace=True)
    df.index.name="impurity"
    return df
```

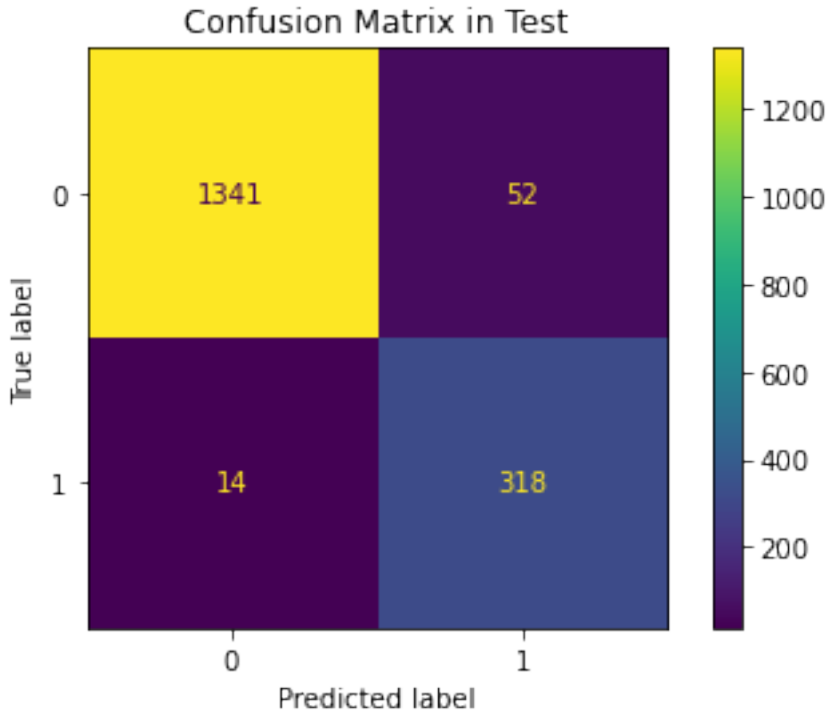
```
Prune Rate: 0  Impurity:  entropy :  95.13
Prune Rate: 0  Impurity:  gini :  95.3
Prune Rate: 0.1  Impurity:  entropy :  95.01
Prune Rate: 0.1  Impurity:  gini :  95.83
Prune Rate: 0.2  Impurity:  entropy :  95.01
Prune Rate: 0.2  Impurity:  gini :  79.71
Prune Rate: 0.3  Impurity:  entropy :  95.01
Prune Rate: 0.3  Impurity:  gini :  79.71
Prune Rate: 0.7  Impurity:  entropy :  79.71
Prune Rate: 0.7  Impurity:  gini :  79.71
Prune Rate: 0.8  Impurity:  entropy :  79.71
Prune Rate: 0.8  Impurity:  gini :  79.71
Prune Rate: 0.9  Impurity:  entropy :  79.71
Prune Rate: 0.9  Impurity:  gini :  79.71
Total time for 7 x 2 grid search is:  838.7186055183411 seconds.
```

[9]:
```
# Print the model results

accuracy_table= organize_results(grid_search,accuracy_grid)
import dataframe_image as dfi
dfi.export(accuracy_table, "decision_gridsearch.png")
```

[10]:
```
# Automatically select the best model
```

```
from numpy import unravel_index
best_index=unravel_index(accuracy_grid.argmax(),  accuracy_grid.shape)
decision_tree_best= DecisionTreeClassifier(grid_search[1][best_index[0]],"yes")
decision_tree_best.
 ↪fit(np_train_data,np_label_train,grid_search[0][best_index[1]])
y_pred= decision_tree_best.predict(np_test_data)

print("--------------- TOTAL ACCURACY -------------------")
acc= test_accuracy(np_label_test,y_pred)
print("********* CLASS BASED ACCURACY**********************")
class_accuracy(decision_tree_best,np_test_data,np_label_test)
confusion_matrix_plot(np_label_test,y_pred)
```

```
--------------- TOTAL ACCURACY -------------------
Accuracy is 96.17
********* CLASS BASED ACCURACY*******************
Class 0
Accuracy is 96.27
Class 1
Accuracy is 95.78
```



[ ]:

Decision_Tree_Beirut_KFold

December 19, 2022

[13]:
```
import pandas as pd
from math import *
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import ConfusionMatrixDisplay
from sklearn.metrics import confusion_matrix
from sklearn.metrics import plot_confusion_matrix
from random import sample
import os
import utils
from utils import shuffling
from utils import normalization
from utils import traintestsplit
from utils import feature_extract
```

[14]:
```
#loading the data
X_train = np.load("X_train.npy")
X_test = np.load("X_test.npy")

y_train = np.load("y_train.npy")
y_test = np.load("y_test.npy")
```

[15]:
```
#loading the data
np_train_data = np.load("np_train_data.npy")
np_test_data = np.load("np_test_data.npy")

np_label_train = np.load("np_label_train.npy")
np_label_test = np.load("np_label_test.npy")
```

[16]:
```
#Define some functions
def impurityCalculation(impurity_choice, labelsOfnode):

    labels, numLabels = np.unique(labelsOfnode, return_counts = True)
    total_labels= np.sum(numLabels)
    p_numLabels = numLabels/total_labels

    if impurity_choice == "entropy":
        entropy = 0
```

```
        total_entropy = np.sum([entropy+ (-np.log2(ii)*ii)  for ii in␣
↪p_numLabels])
        total_impurity = total_entropy


    elif impurity_choice == "gini":
        entropy = 0
        gini= (1/2)*(1-np.sum(np.square(p_numLabels)))
        total_impurity = gini

    else:
        raise Exception("Sorry", impurity_choice, "is not an impurity type. ")

    return total_impurity

def LeftRightSplit(attribute, label, value,impurity_choice):


    total_entropy = impurityCalculation(impurity_choice,label)

    R_split = label[attribute>value]
    L_split = label[attribute<=value]
    L_prop= len(L_split)/len(label)
    R_prop= 1- L_prop

    L_impurity = impurityCalculation(impurity_choice,L_split)
    R_impurity = impurityCalculation(impurity_choice,R_split)

    split_impurity = L_prop*L_impurity + R_prop*R_impurity
    informationGain= total_entropy-split_impurity

    return split_impurity, informationGain


def exhaustive_search(attributes,label,impurity_choice):

    splitCheckAll= np.Inf
    gainAll=0
    valueSplitAll=0

    featureIndexSplit=0

    for ft in range(0,attributes.shape[1]):

        splitCheck= np.Inf

        for val in attributes[:,ft]:
```

```
            isSplit,gainCheck = LeftRightSplit(attributes[:
↪,ft],label,val,impurity_choice)

            if isSplit < splitCheck:
                splitCheck=isSplit
                tempSplitCheck=splitCheck
                splitValue= val
                gainSplit= gainCheck


        if tempSplitCheck < splitCheckAll:
            splitCheckAll=tempSplitCheck
            featureIndexSplit=ft
            valueSplitAll= splitValue
            gainAll=gainSplit


    return splitCheckAll,gainAll,valueSplitAll,featureIndexSplit


def splitNode(attributes,label,impurity_choice):

    bestSplitImpurity, bestSplitInfoGain,bestValue,bestFeatureIndex =␣
↪exhaustive_search(attributes,label,impurity_choice)

    node_impurity = impurityCalculation(impurity_choice,label)

    left_node = attributes[attributes[:,bestFeatureIndex]<=bestValue]
    left_node_labels = label[attributes[:,bestFeatureIndex]<=bestValue]


    right_node = attributes[attributes[:,bestFeatureIndex]>bestValue]
    right_node_labels = label[attributes[:,bestFeatureIndex]>bestValue]


    return left_node,left_node_labels, right_node,right_node_labels,␣
↪node_impurity,bestSplitImpurity, bestSplitInfoGain,bestValue,bestFeatureIndex
```

[17]:
```
class Node():
    def __init__(self,parent,depth):

        self.parent = parent
        self.depth=depth
        self.childen = []

        self.sampleNumber = None
        self.labelNumbers = []
```

```
        self.labelNames= []
        self.bestSplitValue= None
        self.bestSplitIndex= None
        self.splitImpurity = None
        self.isLeaf = 0
        self.leafLabelName= None


def Tree(attributes,label,node,prun,impurity_choice):

    node_labels, labelNumbers = np.unique(label,return_counts= True)
    node.labelNames= node_labels
    node.labelNumbers=labelNumbers
    node.sampleNumber= np.sum(labelNumbers)

    if len(node_labels)==1: # it is pure
        node.isLeaf=1
        node.leafLabelName=node_labels[0]
        return


    else :
        left_node_Feat,left_node_labels, right_node_Feat,right_node_labels,␣
 ↪node_impurity,bestSplitImpurity,\
        bestSplitInfoGain,bestValue,bestFeatureIndex =␣
 ↪splitNode(attributes,label,impurity_choice)
        #Preprunning
        if bestSplitInfoGain <= prun:

            labelname,labelNum= np.unique(label, return_counts=True)
            node.leafLabelName= labelname[np.argmax(labelNum)]
            node.isLeaf = 1
            return
        # You can also add another pruning methods

        node.bestSplitValue=bestValue

        node.bestSplitIndex= bestFeatureIndex
        node.splitImpurity=bestSplitImpurity


        node.L_child= Node(node,node.depth+1)
        node.R_child= Node(node,node.depth+1)
        Tree(left_node_Feat,left_node_labels,node.L_child, prun,impurity_choice)
        Tree(right_node_Feat,right_node_labels,node.R_child,prun,impurity_choice)
```

```
def TraverseTree(node,data):

    if node.isLeaf==1:
        prediction = node.leafLabelName

    else:
        if data[node.bestSplitIndex] <= node.bestSplitValue:
            prediction = TraverseTree(node.L_child,data)
        else:
            prediction = TraverseTree(node.R_child,data)
    return prediction
```

[18]:
```
class DecisionTreeClassifier():

    def __init__(self,criterion,isPrunned="yes"):
        self.beginTree= None
        self.impurity_choice=criterion
        self.isPrunned=isPrunned


    def fit(self,data,label,prun):

        rootNode = Node(None,0)
        if self.isPrunned == "yes":
            Tree(data,label,rootNode,prun,self.impurity_choice)

        else:
            Tree(data,label,rootNode,0,self.impurity_choice)

        self.beginTree=rootNode

    def predict(self,data):

        if self.beginTree==None:
            print(" You need to create a tree !")

        else:
            prediction_list =[]
            for ii in range(0,data.shape[0]):
                prediction_list.append(TraverseTree(self.beginTree,data[ii,:]))

            prediction = np.asarray(prediction_list)
            return prediction
```

[19]:
```
def test_accuracy(test_label,predicted_label):

  print("Accuracy is", round(np.mean(test_label==predicted_label)*100,2))
  return round(100*np.mean(test_label==predicted_label),2)

def class_accuracy(model,data,label):
   class_labels = np.unique(label,return_counts=False)
   class_acc_list = []
   for cl in class_labels:
        print("Class", cl)
        class_pred= model.predict(data[label==cl])
        class_acc= test_accuracy(label[label==cl],class_pred)
        #print("Test accuracy for class ", cl, "is : ", class_acc )
        class_acc_list.append(class_acc)


def confusion_matrix_plot(true_label,predictions):


    class_labels = np.unique(true_label,return_counts=False)

    cm= confusion_matrix(true_label,predictions)
    cp=ConfusionMatrixDisplay(cm,display_labels=class_labels)
    cp.plot()
    plt.title("Confusion Matrix in Test")
    plt.show()

  # Confusion Matrix Creation
    '''
    cm= confusion_matrix(label,preds)
    cp=ConfusionMatrixDisplay(cm,display_labels=class_labels)
    cp.plot()
    plt.title(mode + " Confusion Matrix")
    plt.show()
    '''



# Here I will implement graph plotting
```

[20]:
```
def KFoldCrossValidation(data_train,label_train,model="decision_tree", Kfold=5,␣
 ↪impurity="entropy", prune_rate=0.1):
    sample_count= data_train.shape[0]
    fold_size = int(sample_count/Kfold)
    accuracy_all = list()

    for ff in range(Kfold):
```

```
        test_data=data_train[ff*fold_size:(ff+1)*fold_size,:]
        test_label= label_train[ff*fold_size:(ff+1)*fold_size]

        train_data= np.append(data_train[:ff*fold_size,:
↪],data_train[(ff+1)*fold_size:,:],axis=0)
        train_label= np.append(label_train[:
↪ff*fold_size],label_train[(ff+1)*fold_size:],axis=0)



        decision_tree= DecisionTreeClassifier(impurity,"yes")
        decision_tree.fit( train_data,train_label,prune_rate)

        preds = decision_tree.predict(test_data)
        acc= np.mean(preds==test_label)
        accuracy_all.append(acc)
        print("Fold ",ff+1,"is completed.")
    return sum(accuracy_all)/len(accuracy_all)*100
```

[21]:
```
# K-fold cross validation with grid-search
grid_search = [[0,0.1,0.2,0.3,0.7,0.8,0.9],["entropy","gini"]]
accuracy_grid=np.zeros((len(grid_search[0]),len(grid_search[1])))
K_fold=5
import time

b=time.time()
for i,k in enumerate(grid_search[0]):
    for j,m in enumerate(grid_search[1]):
        acc=KFoldCrossValidation(X_train,y_train,Kfold=5,prune_rate=k,impurity=m)
        print("Prune Rate:", k, " Impurity: ", m, ": ",round(acc,2))
        accuracy_grid[i][j]=round(acc,2)
e=time.time()
print("Total time for ",K_fold, "-fold is: ", e-b, "seconds.")
def organize_results(grid_search,accuracy_grid):

    df=pd.DataFrame(accuracy_grid)
    df.columns = grid_search[1]
    df.set_index(pd.Index(grid_search[0]),inplace=True)
    df.index.name="impurity"
    return df
```

```
Fold  1 is completed.
Fold  2 is completed.
Fold  3 is completed.
Fold  4 is completed.
Fold  5 is completed.
Prune Rate: 0  Impurity:  entropy :  79.44
Fold  1 is completed.
```

```
Fold  2 is completed.
Fold  3 is completed.
Fold  4 is completed.
Fold  5 is completed.
Prune Rate: 0  Impurity:  gini :  79.76
Fold  1 is completed.
Fold  2 is completed.
Fold  3 is completed.
Fold  4 is completed.
Fold  5 is completed.
Prune Rate: 0.1  Impurity:  entropy :  79.46
Fold  1 is completed.
Fold  2 is completed.
Fold  3 is completed.
Fold  4 is completed.
Fold  5 is completed.
Prune Rate: 0.1  Impurity:  gini :  56.04
Fold  1 is completed.
Fold  2 is completed.
Fold  3 is completed.
Fold  4 is completed.
Fold  5 is completed.
Prune Rate: 0.2  Impurity:  entropy :  79.46
Fold  1 is completed.
Fold  2 is completed.
Fold  3 is completed.
Fold  4 is completed.
Fold  5 is completed.
Prune Rate: 0.2  Impurity:  gini :  56.04
Fold  1 is completed.
Fold  2 is completed.
Fold  3 is completed.
Fold  4 is completed.
Fold  5 is completed.
Prune Rate: 0.3  Impurity:  entropy :  56.04
Fold  1 is completed.
Fold  2 is completed.
Fold  3 is completed.
Fold  4 is completed.
Fold  5 is completed.
Prune Rate: 0.3  Impurity:  gini :  56.04
Fold  1 is completed.
Fold  2 is completed.
Fold  3 is completed.
Fold  4 is completed.
Fold  5 is completed.
Prune Rate: 0.7  Impurity:  entropy :  56.04
Fold  1 is completed.
```

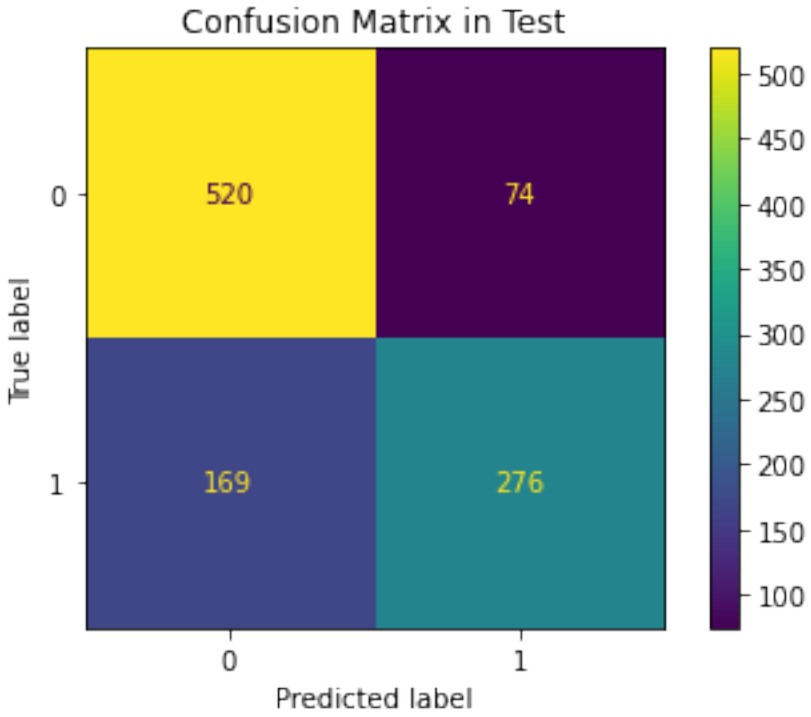
```
Fold  2 is completed.
Fold  3 is completed.
Fold  4 is completed.
Fold  5 is completed.
Prune Rate: 0.7  Impurity:  gini :  56.04
Fold  1 is completed.
Fold  2 is completed.
Fold  3 is completed.
Fold  4 is completed.
Fold  5 is completed.
Prune Rate: 0.8  Impurity:  entropy :  56.04
Fold  1 is completed.
Fold  2 is completed.
Fold  3 is completed.
Fold  4 is completed.
Fold  5 is completed.
Prune Rate: 0.8  Impurity:  gini :  56.04
Fold  1 is completed.
Fold  2 is completed.
Fold  3 is completed.
Fold  4 is completed.
Fold  5 is completed.
Prune Rate: 0.9  Impurity:  entropy :  56.04
Fold  1 is completed.
Fold  2 is completed.
Fold  3 is completed.
Fold  4 is completed.
Fold  5 is completed.
Prune Rate: 0.9  Impurity:  gini :  56.04
Total time for  5 -fold is:  884.8189516067505 seconds.
```

[22]:
```
# Print the model results

accuracy_table= organize_results(grid_search,accuracy_grid)
import dataframe_image as dfi
dfi.export(accuracy_table, "decision_gridsearch.png")
```

[23]:
```
# Automatically select the best model

from numpy import unravel_index
best_index=unravel_index(accuracy_grid.argmax(),  accuracy_grid.shape)
decision_tree_best= DecisionTreeClassifier(grid_search[1][best_index[0]],"yes")
decision_tree_best.
 ↪fit(np_train_data,np_label_train,grid_search[0][best_index[1]])
y_pred= decision_tree_best.predict(np_test_data)

print("--------------- TOTAL ACCURACY -------------------")
```

```
acc= test_accuracy(np_label_test,y_pred)
print("********* CLASS BASED ACCURACY********************")
class_accuracy(decision_tree_best,np_test_data,np_label_test)
confusion_matrix_plot(np_label_test,y_pred)
```

```
--------------- TOTAL ACCURACY ------------------
Accuracy is 76.61
********* CLASS BASED ACCURACY********************
Class 0
Accuracy is 87.54
Class 1
Accuracy is 62.02
```



[ ]:

Decision_Tree_Beirut_Val

December 19, 2022

[23]:
```
import pandas as pd
from math import *
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import ConfusionMatrixDisplay
from sklearn.metrics import confusion_matrix
from sklearn.metrics import plot_confusion_matrix
from random import sample
import os
import utils
```

[24]:
```
#loading the data
X_train = np.load("X_train.npy")
X_test = np.load("X_test.npy")
X_val = np.load("X_val.npy")

y_train = np.load("y_train.npy")
y_test = np.load("y_test.npy")
y_val = np.load("y_val.npy")
```

[25]:
```
#loading the data
np_train_data = np.load("np_train_data.npy")
np_test_data = np.load("np_test_data.npy")
np_val_data = np.load("np_val_data.npy")

np_label_train = np.load("np_label_train.npy")
np_label_test = np.load("np_label_test.npy")
np_label_val = np.load("np_label_val.npy")
```

[26]:
```
#Define some functions
def impurityCalculation(impurity_choice, labelsOfnode):

    labels, numLabels = np.unique(labelsOfnode, return_counts = True)
    total_labels= np.sum(numLabels)
    p_numLabels = numLabels/total_labels

    if impurity_choice == "entropy":
        entropy = 0
```

```
        total_entropy = np.sum([entropy+ (-np.log2(ii)*ii)  for ii in␣
↪p_numLabels])
        total_impurity = total_entropy


    elif impurity_choice == "gini":
        entropy = 0
        gini= (1/2)*(1-np.sum(np.square(p_numLabels)))
        total_impurity = gini

    else:
        raise Exception("Sorry", impurity_choice, "is not an impurity type. ")

    return total_impurity

def LeftRightSplit(attribute, label, value,impurity_choice):


    total_entropy = impurityCalculation(impurity_choice,label)

    R_split = label[attribute>value]
    L_split = label[attribute<=value]
    L_prop= len(L_split)/len(label)
    R_prop= 1- L_prop

    L_impurity = impurityCalculation(impurity_choice,L_split)
    R_impurity = impurityCalculation(impurity_choice,R_split)

    split_impurity = L_prop*L_impurity + R_prop*R_impurity
    informationGain= total_entropy-split_impurity

    return split_impurity, informationGain


def exhaustive_search(attributes,label,impurity_choice):

    splitCheckAll= np.Inf
    gainAll=0
    valueSplitAll=0

    featureIndexSplit=0

    for ft in range(0,attributes.shape[1]):

        splitCheck= np.Inf

        for val in attributes[:,ft]:
```

```
            isSplit,gainCheck = LeftRightSplit(attributes[:
↪,ft],label,val,impurity_choice)

            if isSplit < splitCheck:
                splitCheck=isSplit
                tempSplitCheck=splitCheck
                splitValue= val
                gainSplit= gainCheck


        if tempSplitCheck < splitCheckAll:
            splitCheckAll=tempSplitCheck
            featureIndexSplit=ft
            valueSplitAll= splitValue
            gainAll=gainSplit


    return splitCheckAll,gainAll,valueSplitAll,featureIndexSplit


def splitNode(attributes,label,impurity_choice):

    bestSplitImpurity, bestSplitInfoGain,bestValue,bestFeatureIndex =␣
↪exhaustive_search(attributes,label,impurity_choice)

    node_impurity = impurityCalculation(impurity_choice,label)

    left_node = attributes[attributes[:,bestFeatureIndex]<=bestValue]
    left_node_labels = label[attributes[:,bestFeatureIndex]<=bestValue]


    right_node = attributes[attributes[:,bestFeatureIndex]>bestValue]
    right_node_labels = label[attributes[:,bestFeatureIndex]>bestValue]


    return left_node,left_node_labels, right_node,right_node_labels,␣
↪node_impurity,bestSplitImpurity, bestSplitInfoGain,bestValue,bestFeatureIndex
```

[27]:
```
class Node():
    def __init__(self,parent,depth):

        self.parent = parent
        self.depth=depth
        self.childen = []

        self.sampleNumber = None
        self.labelNumbers = []
```

```
        self.labelNames= []
        self.bestSplitValue= None
        self.bestSplitIndex= None
        self.splitImpurity = None
        self.isLeaf = 0
        self.leafLabelName= None


def Tree(attributes,label,node,prun,impurity_choice):

    node_labels, labelNumbers = np.unique(label,return_counts= True)
    node.labelNames= node_labels
    node.labelNumbers=labelNumbers
    node.sampleNumber= np.sum(labelNumbers)

    if len(node_labels)==1: # it is pure
        node.isLeaf=1
        node.leafLabelName=node_labels[0]
        return


    else :
        left_node_Feat,left_node_labels, right_node_Feat,right_node_labels,␣
 ↪node_impurity,bestSplitImpurity,\
        bestSplitInfoGain,bestValue,bestFeatureIndex =␣
 ↪splitNode(attributes,label,impurity_choice)
        #Preprunning
        if bestSplitInfoGain <= prun:

            labelname,labelNum= np.unique(label, return_counts=True)
            node.leafLabelName= labelname[np.argmax(labelNum)]
            node.isLeaf = 1
            return
        # You can also add another pruning methods

        node.bestSplitValue=bestValue

        node.bestSplitIndex= bestFeatureIndex
        node.splitImpurity=bestSplitImpurity


        node.L_child= Node(node,node.depth+1)
        node.R_child= Node(node,node.depth+1)
        Tree(left_node_Feat,left_node_labels,node.L_child, prun,impurity_choice)
        Tree(right_node_Feat,right_node_labels,node.R_child,prun,impurity_choice)
```

```
def TraverseTree(node,data):

    if node.isLeaf==1:
        prediction = node.leafLabelName

    else:
        if data[node.bestSplitIndex] <= node.bestSplitValue:
            prediction = TraverseTree(node.L_child,data)
        else:
            prediction = TraverseTree(node.R_child,data)
    return prediction
```

[28]:
```
class DecisionTreeClassifier():

    def __init__(self,criterion,isPrunned="yes"):
        self.beginTree= None
        self.impurity_choice=criterion
        self.isPrunned=isPrunned


    def fit(self,data,label,prun):

        rootNode = Node(None,0)
        if self.isPrunned == "yes":
            Tree(data,label,rootNode,prun,self.impurity_choice)

        else:
            Tree(data,label,rootNode,0,self.impurity_choice)

        self.beginTree=rootNode

    def predict(self,data):

        if self.beginTree==None:
            print(" You need to create a tree !")

        else:
            prediction_list =[]
            for ii in range(0,data.shape[0]):
                prediction_list.append(TraverseTree(self.beginTree,data[ii,:]))

            prediction = np.asarray(prediction_list)
            return prediction
```

[29]:
```
def test_accuracy(test_label,predicted_label):

  print("Accuracy is", round(np.mean(test_label==predicted_label)*100,2))
  return round(100*np.mean(test_label==predicted_label),2)

def class_accuracy(model,data,label):
   class_labels = np.unique(label,return_counts=False)
   class_acc_list = []
   for cl in class_labels:
        print("Class", cl)
        class_pred= model.predict(data[label==cl])
        class_acc= test_accuracy(label[label==cl],class_pred)
        #print("Test accuracy for class ", cl, "is : ", class_acc )
        class_acc_list.append(class_acc)


def confusion_matrix_plot(true_label,predictions):


    class_labels = np.unique(true_label,return_counts=False)

    cm= confusion_matrix(true_label,predictions)
    cp=ConfusionMatrixDisplay(cm,display_labels=class_labels)
    cp.plot()
    plt.title("Confusion Matrix in Test")
    plt.show()

  # Confusion Matrix Creation
    '''
    cm= confusion_matrix(label,preds)
    cp=ConfusionMatrixDisplay(cm,display_labels=class_labels)
    cp.plot()
    plt.title(mode + " Confusion Matrix")
    plt.show()
    '''



# Here I will implement graph plotting
```

[30]:
```
# grid-search
grid_search = [[0,0.1,0.2,0.3,0.7,0.8,0.9],["entropy","gini"]]
accuracy_grid=np.zeros((len(grid_search[0]),len(grid_search[1])))
prune_rate = len(grid_search[0])
impurity = len(grid_search[1])
accuracy_all = list()
import time
```

```
b=time.time()
for i,k in enumerate(grid_search[0]):
    for j,m in enumerate(grid_search[1]):
        decision_tree= DecisionTreeClassifier(m,"yes")
        decision_tree.fit(X_train,y_train,k)
        preds = decision_tree.predict(X_val)
        acc = np.mean(preds == y_val)
        accuracy_all.append(acc)
        acc_percent = round(100*acc,2)
        print("Prune Rate:", k, " Impurity: ", m, ": ",acc_percent)
        accuracy_grid[i][j]=acc_percent
e=time.time()
print("Total time for", prune_rate, "x", impurity, "grid search is: ", e-b,␣
 ↪"seconds.")
def organize_results(grid_search,accuracy_grid):

    df=pd.DataFrame(accuracy_grid)
    df.columns = grid_search[1]
    df.set_index(pd.Index(grid_search[0]),inplace=True)
    df.index.name="impurity"
    return df
```

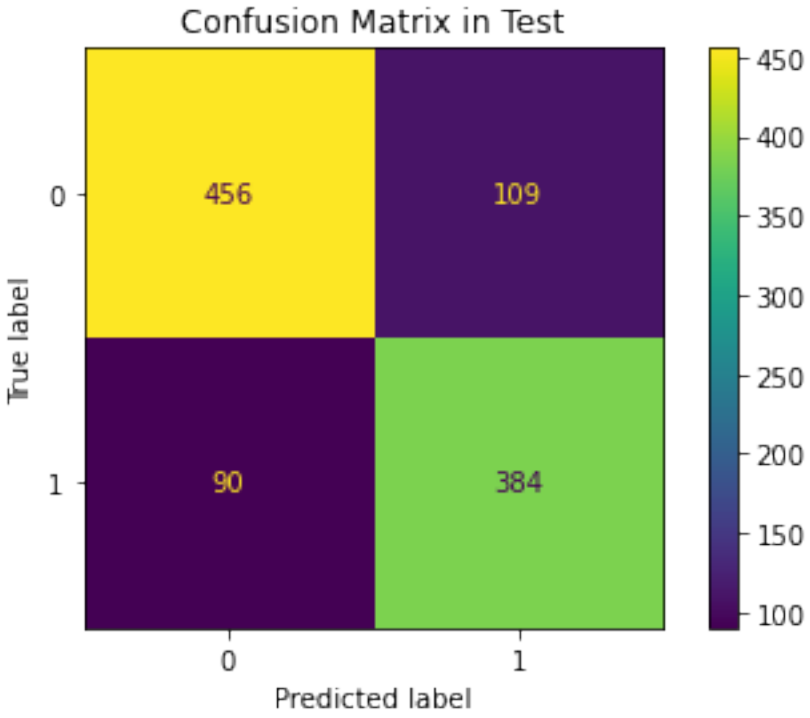
```
Prune Rate: 0  Impurity:  entropy :  82.29
Prune Rate: 0  Impurity:  gini :  81.23
Prune Rate: 0.1  Impurity:  entropy :  78.83
Prune Rate: 0.1  Impurity:  gini :  58.33
Prune Rate: 0.2  Impurity:  entropy :  78.83
Prune Rate: 0.2  Impurity:  gini :  58.33
Prune Rate: 0.3  Impurity:  entropy :  58.33
Prune Rate: 0.3  Impurity:  gini :  58.33
Prune Rate: 0.7  Impurity:  entropy :  58.33
Prune Rate: 0.7  Impurity:  gini :  58.33
Prune Rate: 0.8  Impurity:  entropy :  58.33
Prune Rate: 0.8  Impurity:  gini :  58.33
Prune Rate: 0.9  Impurity:  entropy :  58.33
Prune Rate: 0.9  Impurity:  gini :  58.33
Total time for 7 x 2 grid search is:  175.54380583763123 seconds.
```

[31]:
```
# Print the model results

accuracy_table= organize_results(grid_search,accuracy_grid)
import dataframe_image as dfi
dfi.export(accuracy_table, "decision_gridsearch.png")
```

[32]:
```
# Automatically select the best model
```

```
from numpy import unravel_index
best_index=unravel_index(accuracy_grid.argmax(),  accuracy_grid.shape)
decision_tree_best= DecisionTreeClassifier(grid_search[1][best_index[0]],"yes")
decision_tree_best.
 ↪fit(np_train_data,np_label_train,grid_search[0][best_index[1]])
y_pred= decision_tree_best.predict(np_test_data)

print("--------------- TOTAL ACCURACY -------------------")
acc= test_accuracy(np_label_test,y_pred)
print("********* CLASS BASED ACCURACY*********************")
class_accuracy(decision_tree_best,np_test_data,np_label_test)
confusion_matrix_plot(np_label_test,y_pred)
```

```
--------------- TOTAL ACCURACY -------------------
Accuracy is 80.85
********* CLASS BASED ACCURACY*******************
Class 0
Accuracy is 80.71
Class 1
Accuracy is 81.01
```



[ ]:

kNN_Bonn_KFold

December 19, 2022

[9]:
```
import pandas as pd
from math import *
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import ConfusionMatrixDisplay
from sklearn.metrics import confusion_matrix
from sklearn.metrics import plot_confusion_matrix
from random import sample
import utils
from utils import shuffling
from utils import normalization
from utils import traintestsplit
from utils import feature_extract
import os
```

[10]:
```
#loading the data
X_train = np.load("X_train.npy")
X_test = np.load("X_test.npy")

y_train = np.load("y_train.npy")
y_test = np.load("y_test.npy")
```

[11]:
```
#loading the data
np_train_data = np.load("np_train_data.npy")
np_test_data = np.load("np_test_data.npy")

np_label_train = np.load("np_label_train.npy")
np_label_test = np.load("np_label_test.npy")
```

[12]:
```
class kNearestNeighbor:

  def __init__(self,train_data, train_label, k=3, dmetric="euclidian"):

    self.train_data= train_data
    self.train_label= train_label
    self.k=k
    self.dmetric=dmetric
```

```
  def distance_metric(self,vector1,vector2):

    if self.dmetric=="manhattan":
      return np.abs(vector1-vector2).sum(axis=1)
    if self.dmetric=="euclidian":
      return np.square(vector1-vector2).sum(axis=1)

  def get_neighbors(self, test_data):

    distances= self.distance_metric(self.train_data,test_data)
    indices= np.argsort(distances)[:self.k]
    return indices

  def predict(self,test_data, test_label):
    y_predict= np.zeros(test_label.shape)

    for tt in range(0,test_data.shape[0]):

      indx= self.get_neighbors(test_data[tt])
      y_indices= self.train_label[indx]
      y_pred=np.bincount(y_indices).argmax()
      y_predict[tt]=y_pred

    return y_predict
```

[13]:
```
def test_accuracy(test_label,predicted_label):

  print("Accuracy is", round(np.mean(test_label==predicted_label)*100,2))
  return round(100*np.mean(test_label==predicted_label),2)

def class_accuracy(model,data,label):
   class_labels = np.unique(label,return_counts=False)
   class_acc_list = []
   for cl in class_labels:
        print("Class", cl)
        class_pred= model.predict(data[label==cl],label[label==cl])
        class_acc= test_accuracy(label[label==cl],class_pred)
        #print("Test accuracy for class ", cl, "is : ", class_acc )
        class_acc_list.append(class_acc)

def confusion_matrix_plot(true_label,predictions):


    class_labels = np.unique(true_label,return_counts=False)
```

```
    cm= confusion_matrix(true_label,predictions)
    cp=ConfusionMatrixDisplay(cm,display_labels=class_labels)
    cp.plot()
    plt.title("Confusion Matrix")
    plt.show()
```

[14]:
```
def KFoldCrossValidation(data_train,label_train,model="knn", Kfold=5, k=3,␣
↪metric="euclidian"):
    sample_count= data_train.shape[0]
    fold_size = int(sample_count/Kfold)
    accuracy_all = list()

    for ff in range(Kfold):

        test_data=data_train[ff*fold_size:(ff+1)*fold_size,:]
        test_label= label_train[ff*fold_size:(ff+1)*fold_size]

        train_data= np.append(data_train[:ff*fold_size,:
↪],data_train[(ff+1)*fold_size:,:],axis=0)
        train_label= np.append(label_train[:
↪ff*fold_size],label_train[(ff+1)*fold_size:],axis=0)
        model= kNearestNeighbor(train_data, train_label,k,metric)
        preds = model.predict(test_data,test_label)
        acc= np.mean(preds==test_label)
        accuracy_all.append(acc)
        print("Fold ",ff+1,"is completed.")
    return sum(accuracy_all)/len(accuracy_all)*100
```

[15]:
```
# K-fold cross validation with grid-search
grid_search = [[3,5,7,11,15],["euclidian","manhattan"]]
accuracy_grid=np.zeros((len(grid_search[0]),len(grid_search[1])))
K_fold=5
import time
b= time.time()
for i,k in enumerate(grid_search[0]):
    for j,m in enumerate(grid_search[1]):
        acc=KFoldCrossValidation(X_train,y_train,Kfold=K_fold,k=k,metric=m)
        print("K-Nearest Neighbor:", k, " Metric: ", m, ": ",round(acc,2))
        accuracy_grid[i][j]=round(acc,2)
e= time.time()
print("Total time for ",K_fold, "-fold is: ", e-b, "seconds.")
def organize_results(grid_search,accuracy_grid):

    df=pd.DataFrame(accuracy_grid)
    df.columns = grid_search[1]
    df.set_index(pd.Index(grid_search[0]),inplace=True)
```

```
    df.index.name="k"
    return df
```

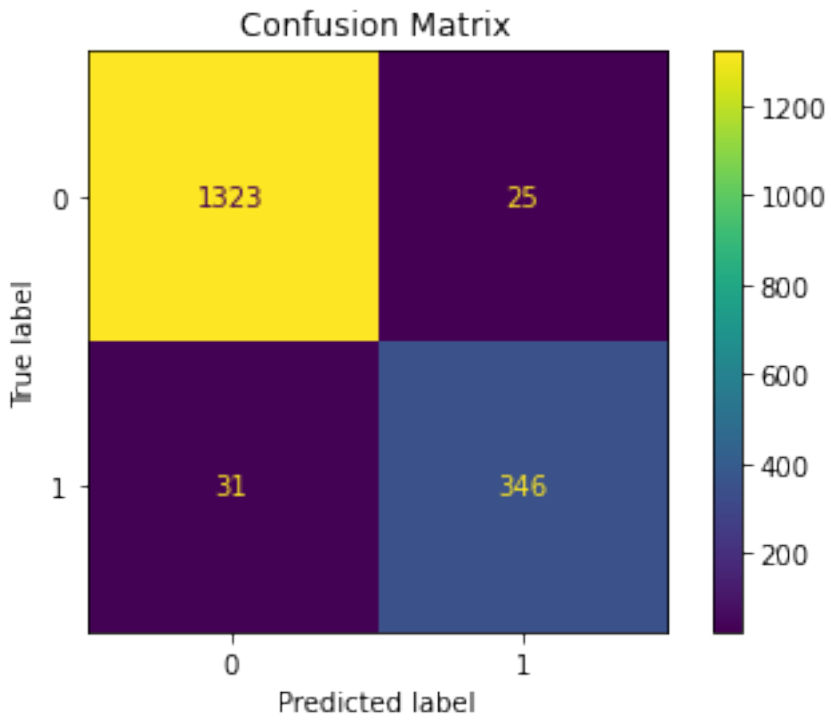
```
Fold  1 is completed.
Fold  2 is completed.
Fold  3 is completed.
Fold  4 is completed.
Fold  5 is completed.
K-Nearest Neighbor: 3  Metric:  euclidian :  96.79
Fold  1 is completed.
Fold  2 is completed.
Fold  3 is completed.
Fold  4 is completed.
Fold  5 is completed.
K-Nearest Neighbor: 3  Metric:  manhattan :  96.96
Fold  1 is completed.
Fold  2 is completed.
Fold  3 is completed.
Fold  4 is completed.
Fold  5 is completed.
K-Nearest Neighbor: 5  Metric:  euclidian :  96.85
Fold  1 is completed.
Fold  2 is completed.
Fold  3 is completed.
Fold  4 is completed.
Fold  5 is completed.
K-Nearest Neighbor: 5  Metric:  manhattan :  97.21
Fold  1 is completed.
Fold  2 is completed.
Fold  3 is completed.
Fold  4 is completed.
Fold  5 is completed.
K-Nearest Neighbor: 7  Metric:  euclidian :  96.95
Fold  1 is completed.
Fold  2 is completed.
Fold  3 is completed.
Fold  4 is completed.
Fold  5 is completed.
K-Nearest Neighbor: 7  Metric:  manhattan :  97.16
Fold  1 is completed.
Fold  2 is completed.
Fold  3 is completed.
Fold  4 is completed.
Fold  5 is completed.
K-Nearest Neighbor: 11  Metric:  euclidian :  96.9
Fold  1 is completed.
Fold  2 is completed.
Fold  3 is completed.
```

```
Fold  4 is completed.
Fold  5 is completed.
K-Nearest Neighbor: 11  Metric:  manhattan :  97.14
Fold  1 is completed.
Fold  2 is completed.
Fold  3 is completed.
Fold  4 is completed.
Fold  5 is completed.
K-Nearest Neighbor: 15  Metric:  euclidian :  96.74
Fold  1 is completed.
Fold  2 is completed.
Fold  3 is completed.
Fold  4 is completed.
Fold  5 is completed.
K-Nearest Neighbor: 15  Metric:  manhattan :  96.92
Total time for  5 -fold is:  107.68478798866272 seconds.
```

[16]:
```
# Print the model results

accuracy_table= organize_results(grid_search,accuracy_grid)
import dataframe_image as dfi
dfi.export(accuracy_table, "knn_gridsearch.png")
```

[17]:
```
# Automatically select the best model

from numpy import unravel_index
best_index=unravel_index(accuracy_grid.argmax(),  accuracy_grid.shape)
knn_best=  kNearestNeighbor(np_train_data,np_label_train,␣
 ↪grid_search[0][best_index[0]],grid_search[1][best_index[1]])
y_pred=knn_best.predict(np_test_data,np_label_test)
print("--------------- TOTAL ACCURACY -------------------")
acc= test_accuracy(np_label_test,y_pred)
print("********* CLASS BASED ACCURACY*******************")
class_accuracy(knn_best,np_test_data,np_label_test)
confusion_matrix_plot(np_label_test,y_pred)
```

```
--------------- TOTAL ACCURACY -------------------
Accuracy is 96.75
********* CLASS BASED ACCURACY*******************
Class 0
Accuracy is 98.15
Class 1
Accuracy is 91.78
```


Confusion Matrix
1323
25
31
346
0
1
True label
0
1
Predicted label
1200
1000
800
600
400
200

[ ]:

kNN_Bonn_val

December 19, 2022

[59]:
```
import pandas as pd
from math import *
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import ConfusionMatrixDisplay
from sklearn.metrics import confusion_matrix
from sklearn.metrics import plot_confusion_matrix
from random import sample
import utils
import os
```

[60]:
```
#loading the data
X_train = np.load("X_train.npy")
X_test = np.load("X_test.npy")
X_val = np.load("X_val.npy")

y_train = np.load("y_train.npy")
y_test = np.load("y_test.npy")
y_val = np.load("y_val.npy")
```

[61]:
```
#loading the data
np_train_data = np.load("np_train_data.npy")
np_test_data = np.load("np_test_data.npy")
np_val_data = np.load("np_val_data.npy")

np_label_train = np.load("np_label_train.npy")
np_label_test = np.load("np_label_test.npy")
np_label_val = np.load("np_label_val.npy")
```

[62]:
```
class kNearestNeighbor:

  def __init__(self,train_data, train_label, k=3, dmetric="euclidian"):

    self.train_data= train_data
    self.train_label= train_label
    self.k=k
    self.dmetric=dmetric
```

```
  def distance_metric(self,vector1,vector2):

    if self.dmetric=="manhattan":
      return np.abs(vector1-vector2).sum(axis=1)
    if self.dmetric=="euclidian":
      return np.square(vector1-vector2).sum(axis=1)

  def get_neighbors(self, test_data):

    distances= self.distance_metric(self.train_data,test_data)
    indices= np.argsort(distances)[:self.k]
    return indices

  def predict(self,test_data, test_label):
    y_predict= np.zeros(test_label.shape)

    for tt in range(0,test_data.shape[0]):

      indx= self.get_neighbors(test_data[tt])
      y_indices= self.train_label[indx]
      y_pred=np.bincount(y_indices).argmax()
      y_predict[tt]=y_pred

    return y_predict
```

[63]:
```
def test_accuracy(test_label,predicted_label):

  print("Accuracy is", round(np.mean(test_label==predicted_label)*100,2))
  return round(100*np.mean(test_label==predicted_label),2)

def class_accuracy(model,data,label):
   class_labels = np.unique(label,return_counts=False)
   class_acc_list = []
   for cl in class_labels:
        print("Class", cl)
        class_pred= model.predict(data[label==cl],label[label==cl])
        class_acc= test_accuracy(label[label==cl],class_pred)
        #print("Test accuracy for class ", cl, "is : ", class_acc )
        class_acc_list.append(class_acc)

def confusion_matrix_plot(true_label,predictions):


    class_labels = np.unique(true_label,return_counts=False)
```

```
    cm= confusion_matrix(true_label,predictions)
    cp=ConfusionMatrixDisplay(cm,display_labels=class_labels)
    cp.plot()
    plt.title("Confusion Matrix")
    plt.show()
```

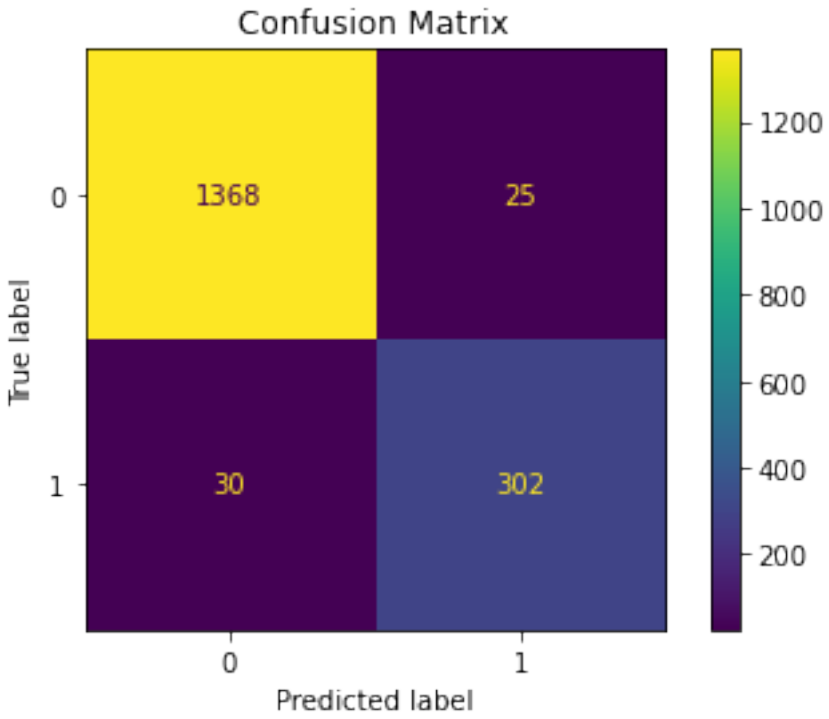
[64]:
```
# grid-search
grid_search = [[3,5,7,11,15],["euclidian","manhattan"]]
k_number = len(grid_search[0])
dis_met = len(grid_search[1])
accuracy_grid=np.zeros((len(grid_search[0]),len(grid_search[1])))
K_fold=5
import time
b= time.time()
for i,k in enumerate(grid_search[0]):
    for j,m in enumerate(grid_search[1]):
        model= kNearestNeighbor(X_train, y_train, k, m)
        preds = model.predict(X_val,y_val)
        acc = np.mean(preds==y_val)
        acc_percent = round(100*acc,2)
        print("K-Nearest Neighbor:", k, " Metric: ", m, ": ",acc_percent)
        accuracy_grid[i][j] = acc_percent
e= time.time()
print("Total time for", k_number, "x", dis_met, "grid search is: ", e-b,␣
 ↪"seconds.")
def organize_results(grid_search,accuracy_grid):

    df=pd.DataFrame(accuracy_grid)
    df.columns = grid_search[1]
    df.set_index(pd.Index(grid_search[0]),inplace=True)
    df.index.name="k"
    return df
```

```
K-Nearest Neighbor: 3  Metric:  euclidian :  96.64
K-Nearest Neighbor: 3  Metric:  manhattan :  96.52
K-Nearest Neighbor: 5  Metric:  euclidian :  96.7
K-Nearest Neighbor: 5  Metric:  manhattan :  96.58
K-Nearest Neighbor: 7  Metric:  euclidian :  96.52
K-Nearest Neighbor: 7  Metric:  manhattan :  96.7
K-Nearest Neighbor: 11  Metric:  euclidian :  96.52
K-Nearest Neighbor: 11  Metric:  manhattan :  96.58
K-Nearest Neighbor: 15  Metric:  euclidian :  96.58
K-Nearest Neighbor: 15  Metric:  manhattan :  96.81
Total time for 5 x 2 grid search is:  11.979809284210205 seconds.
```

[65]:
```
# Print the model results

accuracy_table= organize_results(grid_search,accuracy_grid)
import dataframe_image as dfi
dfi.export(accuracy_table, "knn_gridsearch.png")
```

[66]:
```
# Automatically select the best model

from numpy import unravel_index
best_index=unravel_index(accuracy_grid.argmax(),  accuracy_grid.shape)
knn_best=  kNearestNeighbor(np_train_data,np_label_train,␣
 ↪grid_search[0][best_index[0]],grid_search[1][best_index[1]])
y_pred=knn_best.predict(np_test_data,np_label_test)
print("--------------- TOTAL ACCURACY -------------------")
acc= test_accuracy(np_label_test,y_pred)
print("********* CLASS BASED ACCURACY**********************")
class_accuracy(knn_best,np_test_data,np_label_test)
confusion_matrix_plot(np_label_test,y_pred)
```

```
--------------- TOTAL ACCURACY -------------------
Accuracy is 96.81
********* CLASS BASED ACCURACY**********************
Class 0
Accuracy is 98.21
Class 1
Accuracy is 90.96
```



[ ]:

kNN_Beirut_KFold

December 19, 2022

[40]:
```
import pandas as pd
from math import *
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import ConfusionMatrixDisplay
from sklearn.metrics import confusion_matrix
from sklearn.metrics import plot_confusion_matrix
from random import sample
import os
import utils
from utils import shuffling
from utils import maxminnorm
from utils import traintestsplit
from utils import feature_extract
```

[41]:
```
#loading the data
X_train = np.load("X_train.npy")
X_test = np.load("X_test.npy")

y_train = np.load("y_train.npy")
y_test = np.load("y_test.npy")
```

[42]:
```
#loading the data
np_train_data = np.load("np_train_data.npy")
np_test_data = np.load("np_test_data.npy")

np_label_train = np.load("np_label_train.npy")
np_label_test = np.load("np_label_test.npy")
```

[43]:
```
class kNearestNeighbor:

  def __init__(self,train_data, train_label, k=3, dmetric="euclidian"):

    self.train_data= train_data
    self.train_label= train_label
    self.k=k
    self.dmetric=dmetric
```

```
  def distance_metric(self,vector1,vector2):

    if self.dmetric=="manhattan":
      return np.abs(vector1-vector2).sum(axis=1)
    if self.dmetric=="euclidian":
      return np.square(vector1-vector2).sum(axis=1)

  def get_neighbors(self, test_data):

    distances= self.distance_metric(self.train_data,test_data)
    indices= np.argsort(distances)[:self.k]
    return indices

  def predict(self,test_data, test_label):
    y_predict= np.zeros(test_label.shape)

    for tt in range(0,test_data.shape[0]):

      indx= self.get_neighbors(test_data[tt])
      y_indices= self.train_label[indx]
      y_pred=np.bincount(y_indices).argmax()
      y_predict[tt]=y_pred

    return y_predict
```

[44]:
```
def test_accuracy(test_label,predicted_label):

  print("Accuracy is", round(np.mean(test_label==predicted_label)*100,2))
  return round(100*np.mean(test_label==predicted_label),2)

def class_accuracy(model,data,label):
   class_labels = np.unique(label,return_counts=False)
   class_acc_list = []
   for cl in class_labels:
        print("Class", cl)
        class_pred= model.predict(data[label==cl],label[label==cl])
        class_acc= test_accuracy(label[label==cl],class_pred)
        #print("Test accuracy for class ", cl, "is : ", class_acc )
        class_acc_list.append(class_acc)

def confusion_matrix_plot(true_label,predictions):


    class_labels = np.unique(true_label,return_counts=False)

    cm= confusion_matrix(true_label,predictions)
```

```
    cp=ConfusionMatrixDisplay(cm,display_labels=class_labels)
    cp.plot()
    plt.title("Confusion Matrix")
    plt.show()
```

[45]:
```
def KFoldCrossValidation(data_train,label_train,model="knn", Kfold=5, k=3,␣
↪metric="euclidian"):
    sample_count= data_train.shape[0]
    fold_size = int(sample_count/Kfold)
    accuracy_all = list()

    for ff in range(Kfold):

        test_data=data_train[ff*fold_size:(ff+1)*fold_size,:]
        test_label= label_train[ff*fold_size:(ff+1)*fold_size]

        train_data= np.append(data_train[:ff*fold_size,:
↪],data_train[(ff+1)*fold_size:,:],axis=0)
        train_label= np.append(label_train[:
↪ff*fold_size],label_train[(ff+1)*fold_size:],axis=0)
        model= kNearestNeighbor(train_data, train_label,k,metric)
        preds = model.predict(test_data,test_label)
        acc= np.mean(preds==test_label)
        accuracy_all.append(acc)
        print("Fold ",ff+1,"is completed.")
    return sum(accuracy_all)/len(accuracy_all)*100
```

[46]:
```
# K-fold cross validation with grid-search
grid_search = [[3,5,7,11,15],["euclidian","manhattan"]]
accuracy_grid=np.zeros((len(grid_search[0]),len(grid_search[1])))
K_fold=5
import time
b= time.time()
for i,k in enumerate(grid_search[0]):
    for j,m in enumerate(grid_search[1]):
        acc=KFoldCrossValidation(X_train,y_train,Kfold=K_fold,k=k,metric=m)
        print("K-Nearest Neighbor:", k, " Metric: ", m, ": ",round(acc,2))
        accuracy_grid[i][j]=round(acc,2)
e= time.time()
print("Total time for ",K_fold, "-fold is: ", e-b, "seconds.")
def organize_results(grid_search,accuracy_grid):

    df=pd.DataFrame(accuracy_grid)
    df.columns = grid_search[1]
    df.set_index(pd.Index(grid_search[0]),inplace=True)
    df.index.name="k"
    return df
```

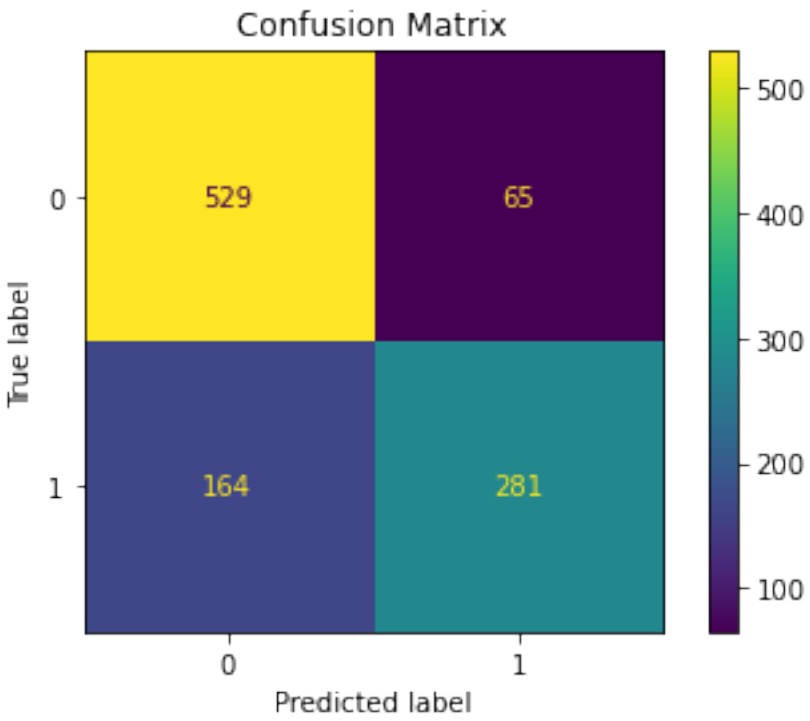
```
Fold  1 is completed.
Fold  2 is completed.
Fold  3 is completed.
Fold  4 is completed.
Fold  5 is completed.
K-Nearest Neighbor: 3  Metric:  euclidian :  78.95
Fold  1 is completed.
Fold  2 is completed.
Fold  3 is completed.
Fold  4 is completed.
Fold  5 is completed.
K-Nearest Neighbor: 3  Metric:  manhattan :  79.03
Fold  1 is completed.
Fold  2 is completed.
Fold  3 is completed.
Fold  4 is completed.
Fold  5 is completed.
K-Nearest Neighbor: 5  Metric:  euclidian :  78.03
Fold  1 is completed.
Fold  2 is completed.
Fold  3 is completed.
Fold  4 is completed.
Fold  5 is completed.
K-Nearest Neighbor: 5  Metric:  manhattan :  78.57
Fold  1 is completed.
Fold  2 is completed.
Fold  3 is completed.
Fold  4 is completed.
Fold  5 is completed.
K-Nearest Neighbor: 7  Metric:  euclidian :  77.45
Fold  1 is completed.
Fold  2 is completed.
Fold  3 is completed.
Fold  4 is completed.
Fold  5 is completed.
K-Nearest Neighbor: 7  Metric:  manhattan :  78.12
Fold  1 is completed.
Fold  2 is completed.
Fold  3 is completed.
Fold  4 is completed.
Fold  5 is completed.
K-Nearest Neighbor: 11  Metric:  euclidian :  78.81
Fold  1 is completed.
Fold  2 is completed.
Fold  3 is completed.
Fold  4 is completed.
Fold  5 is completed.
K-Nearest Neighbor: 11  Metric:  manhattan :  78.68
```

```
Fold  1 is completed.
Fold  2 is completed.
Fold  3 is completed.
Fold  4 is completed.
Fold  5 is completed.
K-Nearest Neighbor: 15  Metric:  euclidian :  79.29
Fold  1 is completed.
Fold  2 is completed.
Fold  3 is completed.
Fold  4 is completed.
Fold  5 is completed.
K-Nearest Neighbor: 15  Metric:  manhattan :  79.35
Total time for  5 -fold is:  19.966706037521362 seconds.
```

[47]:
```
# Print the model results

accuracy_table= organize_results(grid_search,accuracy_grid)
import dataframe_image as dfi
dfi.export(accuracy_table, "knn_gridsearch.png")
```

[48]:
```
# Automatically select the best model

from numpy import unravel_index
best_index=unravel_index(accuracy_grid.argmax(),  accuracy_grid.shape)
knn_best=  kNearestNeighbor(np_train_data,np_label_train,␣
 ↪grid_search[0][best_index[0]],grid_search[1][best_index[1]])
y_pred=knn_best.predict(np_test_data,np_label_test)
print("--------------- TOTAL ACCURACY -------------------")
acc= test_accuracy(np_label_test,y_pred)
print("********* CLASS BASED ACCURACY********************")
class_accuracy(knn_best,np_test_data,np_label_test)
confusion_matrix_plot(np_label_test,y_pred)
```

```
--------------- TOTAL ACCURACY -------------------
Accuracy is 77.96
********* CLASS BASED ACCURACY********************
Class 0
Accuracy is 89.06
Class 1
Accuracy is 63.15
```


Confusion Matrix
529
65
164
281
0
1
True label
0
1
Predicted label
500
400
300
200
100

[ ]:

kNN_Beirut_val

December 19, 2022

[41]:
```
import pandas as pd
from math import *
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import ConfusionMatrixDisplay
from sklearn.metrics import confusion_matrix
from sklearn.metrics import plot_confusion_matrix
from random import sample
import utils
import os
```

[42]:
```
#loading the data
X_train = np.load("X_train.npy")
X_test = np.load("X_test.npy")
X_val = np.load("X_val.npy")

y_train = np.load("y_train.npy")
y_test = np.load("y_test.npy")
y_val = np.load("y_val.npy")
```

[43]:
```
#loading the data
np_train_data = np.load("np_train_data.npy")
np_test_data = np.load("np_test_data.npy")
np_val_data = np.load("np_val_data.npy")

np_label_train = np.load("np_label_train.npy")
np_label_test = np.load("np_label_test.npy")
np_label_val = np.load("np_label_val.npy")
```

[44]:
```
class kNearestNeighbor:

  def __init__(self,train_data, train_label, k=3, dmetric="euclidian"):

    self.train_data= train_data
    self.train_label= train_label
    self.k=k
    self.dmetric=dmetric
```

```
  def distance_metric(self,vector1,vector2):

    if self.dmetric=="manhattan":
      return np.abs(vector1-vector2).sum(axis=1)
    if self.dmetric=="euclidian":
      return np.square(vector1-vector2).sum(axis=1)

  def get_neighbors(self, test_data):

    distances= self.distance_metric(self.train_data,test_data)
    indices= np.argsort(distances)[:self.k]
    return indices

  def predict(self,test_data, test_label):
    y_predict= np.zeros(test_label.shape)

    for tt in range(0,test_data.shape[0]):

      indx= self.get_neighbors(test_data[tt])
      y_indices= self.train_label[indx]
      y_pred=np.bincount(y_indices).argmax()
      y_predict[tt]=y_pred

    return y_predict
```

[45]:
```
def test_accuracy(test_label,predicted_label):

  print("Accuracy is", round(np.mean(test_label==predicted_label)*100,2))
  return round(100*np.mean(test_label==predicted_label),2)

def class_accuracy(model,data,label):
   class_labels = np.unique(label,return_counts=False)
   class_acc_list = []
   for cl in class_labels:
        print("Class", cl)
        class_pred= model.predict(data[label==cl],label[label==cl])
        class_acc= test_accuracy(label[label==cl],class_pred)
        #print("Test accuracy for class ", cl, "is : ", class_acc )
        class_acc_list.append(class_acc)

def confusion_matrix_plot(true_label,predictions):


    class_labels = np.unique(true_label,return_counts=False)
```

```
    cm= confusion_matrix(true_label,predictions)
    cp=ConfusionMatrixDisplay(cm,display_labels=class_labels)
    cp.plot()
    plt.title("Confusion Matrix")
    plt.show()
```

[46]:
```
# grid-search
grid_search = [[3,5,7,11,15],["euclidian","manhattan"]]
k_number = len(grid_search[0])
dis_met = len(grid_search[1])
accuracy_grid=np.zeros((len(grid_search[0]),len(grid_search[1])))
import time
b= time.time()
for i,k in enumerate(grid_search[0]):
    for j,m in enumerate(grid_search[1]):
        model= kNearestNeighbor(X_train, y_train, k, m)
        preds = model.predict(X_val,y_val)
        acc = np.mean(preds==y_val)
        acc_percent = round(100*acc,2)
        print("K-Nearest Neighbor:", k, " Metric: ", m, ": ",acc_percent)
        accuracy_grid[i][j] = acc_percent
e= time.time()
print("Total time for", k_number, "x", dis_met, "grid search is: ", e-b,␣
 ↪"seconds.")
def organize_results(grid_search,accuracy_grid):

    df=pd.DataFrame(accuracy_grid)
    df.columns = grid_search[1]
    df.set_index(pd.Index(grid_search[0]),inplace=True)
    df.index.name="k"
    return df
```

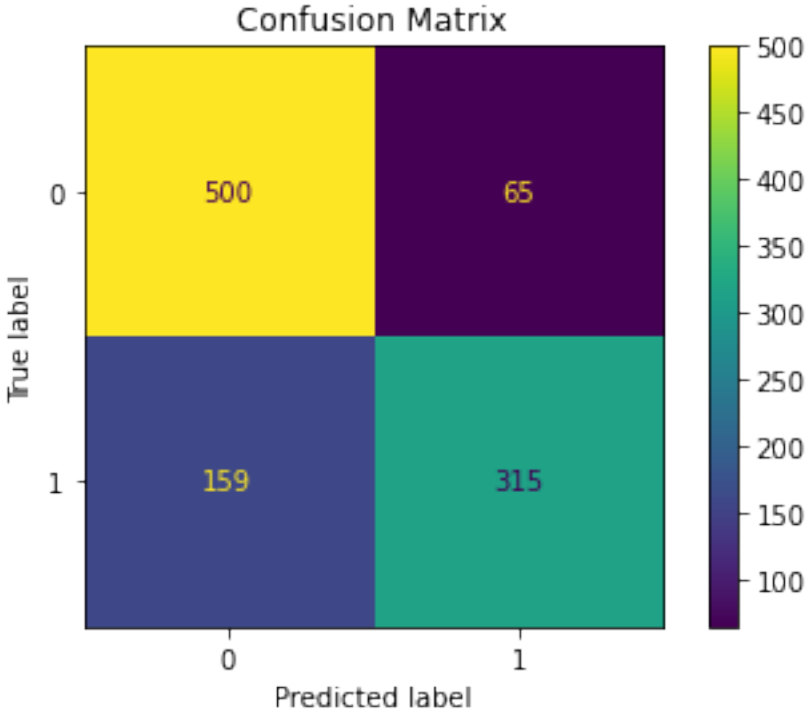
```
K-Nearest Neighbor: 3  Metric:  euclidian :  78.44
K-Nearest Neighbor: 3  Metric:  manhattan :  79.69
K-Nearest Neighbor: 5  Metric:  euclidian :  78.06
K-Nearest Neighbor: 5  Metric:  manhattan :  78.44
K-Nearest Neighbor: 7  Metric:  euclidian :  79.4
K-Nearest Neighbor: 7  Metric:  manhattan :  78.92
K-Nearest Neighbor: 11  Metric:  euclidian :  80.08
K-Nearest Neighbor: 11  Metric:  manhattan :  79.31
K-Nearest Neighbor: 15  Metric:  euclidian :  81.62
K-Nearest Neighbor: 15  Metric:  manhattan :  80.27
Total time for 5 x 2 grid search is:  3.6341094970703125 seconds.
```

[47]:
```
# Print the model results
```

```
accuracy_table= organize_results(grid_search,accuracy_grid)
import dataframe_image as dfi
dfi.export(accuracy_table, "knn_gridsearch.png")
```

[48]:
```
# Automatically select the best model

from numpy import unravel_index
best_index=unravel_index(accuracy_grid.argmax(),  accuracy_grid.shape)
knn_best=  kNearestNeighbor(np_train_data,np_label_train,␣
 ↪grid_search[0][best_index[0]],grid_search[1][best_index[1]])
y_pred=knn_best.predict(np_test_data,np_label_test)
print("--------------- TOTAL ACCURACY -------------------")
acc= test_accuracy(np_label_test,y_pred)
print("********* CLASS BASED ACCURACY********************")
class_accuracy(knn_best,np_test_data,np_label_test)
confusion_matrix_plot(np_label_test,y_pred)
```

```
--------------- TOTAL ACCURACY -------------------
Accuracy is 78.44
********* CLASS BASED ACCURACY********************
Class 0
Accuracy is 88.5
Class 1
Accuracy is 66.46
```



[ ]:

neuralnets

December 19, 2022

[ ]:
```
import pandas as pd
from math import *
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import ConfusionMatrixDisplay
from sklearn.metrics import confusion_matrix
from sklearn.metrics import plot_confusion_matrix
from random import sample
import utils
from utils import shuffling
from utils import normalization
from utils import traintestsplit
from utils import feature_extractbonn
from utils import feature_extractbeirut
from utils import initializer
from utils import encode_label
from utils import maxminnorm
import os
```

[ ]:
```
dataset="beirute"
```

[ ]:
```
if dataset=="bonn":

        epilepsy_data = pd.read_csv("bonn_epilepsy.csv", sep =",")
        epilepsy_data.drop("Unnamed",axis=1,inplace=True)
        epilepsy_data.head()
        epilepsy_data.y = epilepsy_data.y==1
        epilepsy_data.y = epilepsy_data.y.astype(int)

        epilepsy_data= epilepsy_data[epilepsy_data.isnull().any(axis=1)==False]

        label = epilepsy_data["y"].astype("category").to_numpy()
        label2 = epilepsy_data["y"]

        epilepsy_data.drop("y",axis=1,inplace=True)
        epilepsy_data= feature_extractbonn(epilepsy_data)
        epilepsy_data= epilepsy_data.iloc[:,-13:]
        #normalize the data
```

```
        normalized = normalization(epilepsy_data, label2)
elif dataset=="beirute":
    epilepsy_data = pd.read_csv("beirut_epilepsy.csv", sep =",")

    epilepsy_data.drop("Unnamed",axis=1,inplace=True)

    epilepsy_data = epilepsy_data.loc[(epilepsy_data["y"] == 0) |␣
 ↪(epilepsy_data["y"] == 1)].reset_index()
    epilepsy_data.drop("index",axis=1,inplace=True)

    epilepsy_data.y = epilepsy_data.y.astype(int)
    epilepsy_data= epilepsy_data[epilepsy_data.isnull().any(axis=1)==False]

    label = epilepsy_data["y"].astype("category").to_numpy()
    label2 = epilepsy_data["y"]
    epilepsy_data.drop("y",axis=1,inplace=True)

    epilepsy_data = feature_extractbeirut(epilepsy_data)

    epilepsy_data= epilepsy_data.iloc[:,-5:]
    #normalize the data
    normalized = maxminnorm(epilepsy_data, label2)
```

[ ]:
```
#shuffle the data
shuffled = shuffling(normalized)
#split the data into train and test sets
X_train, X_test, Y_train, Y_test = traintestsplit(shuffled,0.15)
```

[ ]:
```
np_train_data= X_train
np_test_data= X_test

np_label_train=Y_train
np_label_test=Y_test
```

[ ]:
```
def confusion_matrix_plot(true_label,predictions):



    class_labels = np.unique(true_label,return_counts=False)

    cm= confusion_matrix(true_label,predictions)
    cp=ConfusionMatrixDisplay(cm,display_labels=class_labels)
    cp.plot()
    plt.title("Confusion Matrix")
    plt.show()
```

[ ]:
```
class Layer:
```

```
    def __init__(self,currNeurons, activFunction,name):
        self.currNeurons=currNeurons
        self.activFunction=activFunction
        self.name=name

        self.delta=None
        self.error=None
        self.cacheActiv=None
        self.weights=None

    def layer_activation(self,net):
        if self.activFunction=="relu":
            return np.maximum(0,net)
        elif self.activFunction=="sigmoid":
            return 1/(1+np.exp(-net))
        elif self.activFunction=="tanh":
            return np.tanh(net)
        elif self.activFunction=="softmax":
            return np.exp(net)/np.sum(np.exp(net),axis=0)

    def layer_derivative(self,activ):

        if self.activFunction=="relu":
            return 1*(activ>0)
        elif self.activFunction=="sigmoid":
            return activ*(1-activ)
        elif self.activFunction=="tanh":
            return 1-activ**2
        elif self.activFunction=="softmax":
               return activ*(1-activ)
```

[ ]:
```
class Sequential:

    def __init__(self,inputDim):
        self.layers=[]
        self.inputDim=inputDim

        self.loss=None
        self.lr=0.01
```

```
    def layer_addition(self,layer):
        self.layers.append(layer)

    def compile_(self,lossFn,learning_rate,weight_initializer):

        if lossFn=="mse":
            self.loss="mse"
        elif lossFn=="cross_entropy":
            self.loss="cross_entropy"

        self.lr=learning_rate
        self.weight_initializer= weight_initializer

        for i in range(len(self.layers)):

            lyr=self.layers[i]
            if i==0:
                lyr.weights= initializer(init_choice=self.
↪weight_initializer,shape=(self.layers[0].currNeurons,self.inputDim+1))
            else:
                lyr.weights= initializer(init_choice=self.
↪weight_initializer,shape=(self.layers[i].currNeurons,self.layers[i-1].
↪currNeurons+1))

    def forward_propagation(self,inp):

        output=inp

        for lyr in self.layers:

            num_samples=output.shape[1]
            output=np.r_[output,[np.ones(num_samples)*1]]
            output=lyr.layer_activation(np.matmul(lyr.weights,output))

            lyr.cacheActiv=output


        return output

    def predict(self, data):
        out = self.forward_propagation(data)
        return np.argmax(out, axis=0)
```

```
   def backward_propagation(self,inp,out):
       batch_size= inp.shape[1]
       network_output=self.forward_propagation(inp)

       for ly in reversed(range(len(self.layers))):

               layer= self.layers[ly]
               if layer == self.layers[len(self.layers)-1]:

                   if self.loss=="cross_entropy" and layer.
↪activFunction=="softmax":
                       layer.delta=network_output-out


                   elif self.loss=="mse":

                       layer.error = network_output-out
                       errorDerivative = layer.layer_derivative(layer.
↪cacheActiv)
                       layer.delta=layer.error*errorDerivative
                   else:
                       print("Sorry, I do not want to perform this operation.")

               else:
                       layerNext = self.layers[ly+1]
                       layerNextWeights= layerNext.weights[:,:-1]
                       layer.error = np.matmul(layerNextWeights.T,layerNext.
↪delta)
                       errorDerivative= layer.layer_derivative(layer.cacheActiv)
                       layer.delta=errorDerivative*layer.error


       for ly in range(len(self.layers)):

           if ly==0:

               activationInput=np.r_[inp,[np.ones(batch_size)*1]]

           else:
               activationInput=np.r_[self.layers[ly-1].cacheActiv,[np.
↪ones(batch_size)*1]]


           dW= np.matmul(self.layers[ly].delta,activationInput.T)/batch_size

           self.layers[ly].weights=self.layers[ly].weights-self.lr*dW
```

```
   def loss_calculation(self,data,label):

            val_out = self.forward_propagation(data.T)
            val_label=label
            if(self.loss == 'cross_entropy'):
                err = - np.sum(np.log(val_out)*encode_label(val_label).T)/
↪val_out.shape[1]
            elif(self.loss == 'mse'):
                err = np.sum((encode_label(val_label).T - val_out)**2)/val_out.
↪shape[1]
            return err




   def fit(self,data,label,val_ratio,epoch,batch_size):
       trainError= []
       trainAcc=[]
       valError=[]
       valAcc=[]


       for e in range(epoch):



            ␣
↪print('=========================================================\nEpoch', e+1)
            indx= np.random.permutation(data.shape[0])
            data_temp=data[indx]
            label_temp=label[indx]

            val_range=int(data.shape[0]*val_ratio)

            val_data= data[:val_range]

            val_label=label[:val_range]

            data_temp=data_temp[val_range:]
            label_temp=label_temp[val_range:]

            iterations = int(np.floor(len(data)/batch_size))

            for it in range(iterations):
```

```
                data_batch=data_temp[it*batch_size:it*batch_size+batch_size]
                label_batch=label_temp[it*batch_size:it*batch_size+batch_size]
                if label_batch.shape[0]!=0:
                    label_batch = encode_label(label_batch)
                    self.backward_propagation(data_batch.T, label_batch.T)

            err_tr= self.loss_calculation(data_temp,label_temp)
            err_val=self.loss_calculation(val_data,val_label)
            valError.append(err_val)
            trainError.append(err_tr)
            if self.loss=="cross_entropy":
                print("CE loss in training: ", err_tr)
                print("CE loss in training: ", err_val)
            else:
                print("MSE loss in training: ", err_tr)
                print("MSE loss in training: ", err_val)




            pred_train = self.predict(data.T)

            #trAcc = np.sum(trPred.reshape((trPred.shape[0],1)) == out)/trPred.
↪shape[0]*100
            acc_train=np.sum(pred_train==label)/pred_train.shape[0]
            print('Training Accuracy: ', acc_train*100)
            trainAcc.append(acc_train*100)

            pred_val = self.predict(val_data.T)
            acc_val= np.sum(pred_val==val_label)/pred_val.shape[0]
            print('Validation Accuracy: ', acc_val*100)
            valAcc.append(acc_val*100)


        return trainAcc, valAcc, valError,trainError
```

1 Here, create your models

[ ]:
```
mlp= Sequential(5)
mlp.layer_addition(Layer(50,"tanh","layer1"))
mlp.layer_addition(Layer(20,"tanh","layer1"))
mlp.layer_addition(Layer(10,"tanh","layer1"))
mlp.layer_addition(Layer(2,"softmax","layer1"))
mlp.compile_("cross_entropy",0.5,"normal")
```

[ ]:
```
trainAcc, valAcc, valError,trainError = mlp.fit(np_train_data,np_label_train,0.
↪2,5000,50)
```

[ ]:
```
testPred = mlp.predict(np_test_data.T)
testAcc = np.sum(testPred ==np_label_test)/testPred.shape[0]
print(testAcc*100)
```

[ ]:
```
confusion_matrix_plot(np_label_test,testPred)
```

[ ]:
```
plt.plot(valAcc)
plt.plot(trainAcc)
plt.legend(["Validation Accuracy","Training Accuracy"])
plt.title("Train and Validation Accuracies")
```

2 END

utils

December 19, 2022

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

# # Helper Functions

# In[ ]:


import pandas as pd
from math import *
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import ConfusionMatrixDisplay
from sklearn.metrics import confusion_matrix
from sklearn.metrics import plot_confusion_matrix
from random import sample
import os


# In[ ]:


def shuffling(data): #works
    shuf_data = data.sample(frac = 1).reset_index()
    shuf_data.pop('index')
    return shuf_data


# In[ ]:


def standardization(data, label): #works, very very slow
    norm_data = data.copy()
    row_num = norm_data.shape[0]
    for column in norm_data.columns:
        col_obj = norm_data[column]
        mu_m = col_obj.sum()/row_num
        std_dev = col_obj.std()
```

```
        col_obj = (col_obj - mu_m)/std_dev
        norm_data[column] = col_obj
    norm_data["y"] = label
    return data


# In[ ]:


def z_norm(data, label): #works, very slow
    data_z_norm = data.copy()
    for column in data_z_norm.columns: #or don't put .columns, doesn't matter␣
 ↪but it becomes slower
        data_z_norm[column] = (data_z_norm[column] - data_z_norm[column].mean())/
 ↪data_z_norm[column].std()
    data_z_norm["y"] = label
    return data_z_norm


# In[ ]:


def normalization(data, label): #works, very fast
    means = data.mean()
    std_devs = data.std()
    nor_data = (data - means)/std_devs
    nor_data["y"] = label
    return nor_data


# In[ ]:


def traintestsplit(data,ratio): #works
    row_num = data.shape[0] #row number
    label = data["y"].to_numpy()
    data.pop("y")
    data_n = data.to_numpy()
    if row_num%10 == 0 or row_num%100 == 0 or row_num%1000 == 0:
        test_num = int(row_num*ratio)
        X_test = data_n[0:test_num,:]
        y_test = label[0:test_num]
        X_train = data_n[test_num:,:]
        y_train = label[test_num:]
        return X_train, X_test, y_train, y_test
    else:
        test_num = int(floor(row_num*ratio))
```

```
        X_test = data_n[0:test_num,:]
        y_test = label[0:test_num]
        X_train = data_n[test_num:,:]
        y_train = label[test_num:]
        return X_train, X_test, y_train, y_test


# In[ ]:


def feature_extractbonn(data):
    max_data = data.max(axis=1)
    max_data_s= max_data**2
    max_data_c= max_data**3

    min_data = data.min(axis=1)
    min_data_s= min_data**2
    min_data_c= max_data**3

    max_min_dif= max_data-min_data
    max_min_dif_s= max_min_dif**2

    mean_data = data.mean(axis=1)
    var_data = data.var(axis=1)
    std_data = data.std(axis=1)

    skew_data = data.skew(axis=1)
    kurt_data = data.kurtosis(axis=1)

    data["MAX"] = max_data
    data["MIN"] = min_data
    data["MAX2"]= max_data_s
    data["MIN2"]= min_data_s
    data["MAX3"]= max_data_c
    data["MIN3"]= min_data_c
    data["MxMnDif"] =  max_min_dif
    data["MxMnDif2"] =  max_min_dif_s

    data["MEAN"] = mean_data
    data["VAR"] = var_data
    data["STD"] = std_data

    data["SKEW"] = skew_data
    data["KURT"] = kurt_data

    return data
```

```
def feature_extractbeirut(data):
    max_data = data.max(axis=1)
    max_data_s= max_data**2
    max_data_c= max_data**3

    min_data = data.min(axis=1)
    min_data_s= min_data**2
    min_data_c= max_data**3

    max_min_dif= max_data-min_data
    max_min_dif_s= max_min_dif**2

    mean_data = data.mean(axis=1)
    var_data = data.var(axis=1)
    std_data = data.std(axis=1)

    skew_data = data.skew(axis=1)
    kurt_data = data.kurtosis(axis=1)

    #data["MAX"] = max_data
    #data["MIN"] = min_data
    #data["MAX2"]= max_data_s
    #data["MIN2"]= min_data_s
    #data["MAX3"]= max_data_c
    #data["MIN3"]= min_data_c
    #data["MxMnDif"] =  max_min_dif
    #data["MxMnDif2"] =  max_min_dif_s

    data["MEAN"] = mean_data
    data["VAR"] = var_data
    data["STD"] = std_data

    data["SKEW"] = skew_data
    data["KURT"] = kurt_data

    return data

def traintestvalsplit(data,ratio): #works
    row_num = data.shape[0] #row number
    label = data["y"].to_numpy()
    data.pop("y")
    data_n = data.to_numpy()
    if row_num%10 == 0 or row_num%100 == 0 or row_num%1000 == 0:
        test_num = int(row_num*ratio)
        X_test = data_n[0:test_num,:]
        y_test = label[0:test_num]
        X_val = data_n[test_num:2*test_num,:]
```

```
        y_val = label[test_num:2*test_num]
        X_train = data_n[2*test_num:,:]
        y_train = label[2*test_num:]
        return X_train, X_test, X_val, y_train, y_test, y_val
    else:
        test_num = int(floor(row_num*ratio))
        X_test = data_n[0:test_num,:]
        y_test = label[0:test_num]
        X_val = data_n[test_num:2*test_num,:]
        y_val = label[test_num:2*test_num]
        X_train = data_n[2*test_num:,:]
        y_train = label[2*test_num:]
        return X_train, X_test, X_val, y_train, y_test, y_val

def maxminnorm(data, label): #works, very fast
    mins = data.min()
    maxs = data.max()
    nor_data = (data - mins)/(maxs - mins)
    nor_data["y"] = label
    return nor_data


# In[1]:


def initializer(init_choice="normal",shape=(1,1)):

    if init_choice== "normal":
        return np.random.normal(0.1,0.01,shape)
    elif init_choice== "uniform":
        return np.random.uniform(-0.1,0.1,shape)
    elif init_choice == "he":
        return np.random.normal(0.1,0.01,shape)*np.sqrt(2/shape[1])
    elif init_choice== "xavier":
        return np.random.normal(0.1,0.01,shape)*np.sqrt(2/(shape[0]+shape[1]))
    else:
        raise ValueError("Initialization Error")



# In[ ]:



def encode_label(y):
    max_label= np.max(y)+1
    encoded_matrix= np.zeros((y.shape[0],max_label))
```

```
    for i in range(len(y)):
        encoded_matrix[i,y[i]]=1

    return encoded_matrix
```

Data_Visualization_Bonn

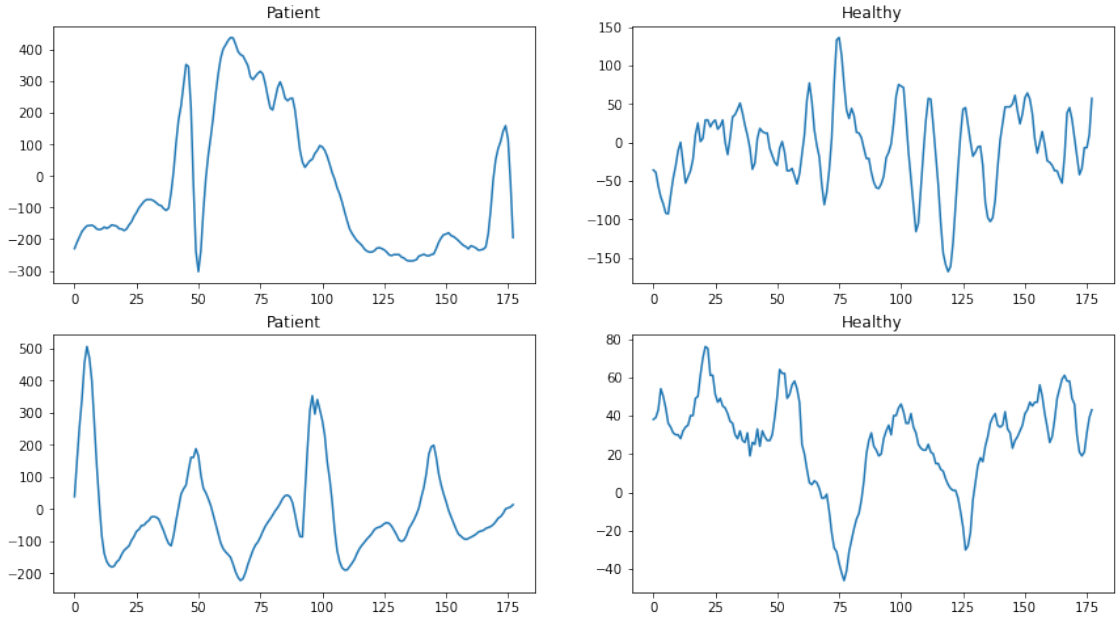
December 19, 2022

[97]:
```
import pandas as pd
from math import *
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import ConfusionMatrixDisplay
from sklearn.metrics import confusion_matrix
from sklearn.metrics import plot_confusion_matrix
from random import sample
import os
import random
```

[98]:
```
epilepsy_data = pd.read_csv("bonn_epilepsy.csv", sep =",")

epilepsy_data.drop("Unnamed",axis=1,inplace=True)
epilepsy_data.y = epilepsy_data.y==1
epilepsy_data.y = epilepsy_data.y.astype(int)

epilepsy_data= epilepsy_data[epilepsy_data.isnull().any(axis=1)==False]
label = epilepsy_data["y"].astype("category").to_numpy()
epileptic_data=epilepsy_data.drop("y",axis=1,inplace=False).to_numpy()
```

[99]:
```
patient_list=epilepsy_data.index[epilepsy_data["y"] == 1].tolist()
healthy_list = epilepsy_data.index[epilepsy_data["y"] == 0].tolist()
random.shuffle(patient_list)
random.shuffle(healthy_list)
```

[100]:
```
for p in range(2):
    plt.subplot(2, 2, 2*p+1)
    plt.plot(epileptic_data[patient_list[p],:])
    plt.title("Patient")
    plt.subplot(2, 2, 2*(p+1))
    plt.plot(epileptic_data[healthy_list[p],:])
    plt.title("Healthy")
    plt.subplots_adjust(bottom=0.5, right=2, top=2)
```


Patient
Healthy
Patient
Healthy

[ ]:

Data_Visualization_Beirut

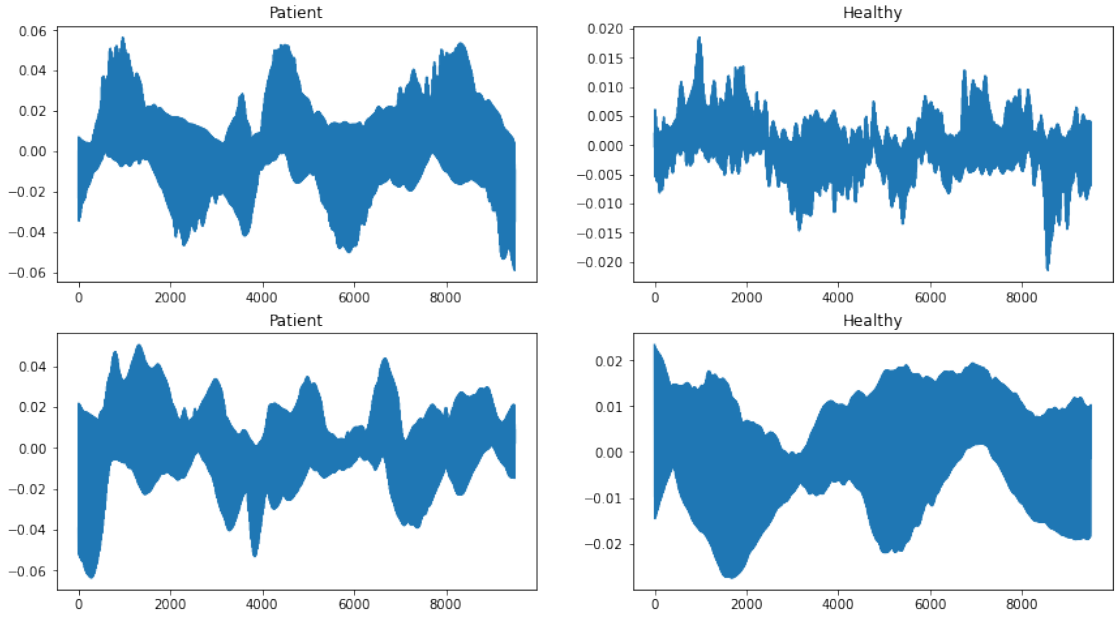
December 19, 2022

[22]:
```
import pandas as pd
from math import *
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import ConfusionMatrixDisplay
from sklearn.metrics import confusion_matrix
from sklearn.metrics import plot_confusion_matrix
from random import sample
import os
import random
```

[35]:
```
epilepsy_data = pd.read_csv("beirut_epilepsy.csv", sep =",")

epilepsy_data.drop("Unnamed",axis=1,inplace=True)
epilepsy_data = epilepsy_data.loc[(epilepsy_data["y"] == 0) |␣
↪(epilepsy_data["y"] == 1)].reset_index()
epilepsy_data.drop("index",axis=1,inplace=True)
epilepsy_data.y = epilepsy_data.y.astype(int)

epilepsy_data= epilepsy_data[epilepsy_data.isnull().any(axis=1)==False]
label = epilepsy_data["y"].astype("category").to_numpy()
epileptic_data=epilepsy_data.drop("y",axis=1,inplace=False).to_numpy()
```

[36]:
```
patient_list=epilepsy_data.index[epilepsy_data["y"] == 1].tolist()
healthy_list = epilepsy_data.index[epilepsy_data["y"] == 0].tolist()
random.shuffle(patient_list)
random.shuffle(healthy_list)
```

[37]:
```
for p in range(2):
    plt.subplot(2, 2, 2*p+1)
    plt.plot(epileptic_data[patient_list[p],:])
    plt.title("Patient")
    plt.subplot(2, 2, 2*(p+1))
    plt.plot(epileptic_data[healthy_list[p],:])
    plt.title("Healthy")
    plt.subplots_adjust(bottom=0.5, right=2, top=2)
```


Patient
Healthy
Patient
Healthy

[ ]:

train_test_split_Bonn

December 19, 2022

[8]:
```
import pandas as pd
from math import *
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import ConfusionMatrixDisplay
from sklearn.metrics import confusion_matrix
from sklearn.metrics import plot_confusion_matrix
from random import sample
import utils
from utils import shuffling
from utils import normalization
from utils import traintestsplit
from utils import feature_extractbonn
import os
```

[9]:
```
#prepare the data for feature engineering and preprocessing
epilepsy_data = pd.read_csv("bonn_epilepsy.csv", sep =",")

epilepsy_data.drop("Unnamed",axis=1,inplace=True)
epilepsy_data.head()
epilepsy_data.y = epilepsy_data.y==1
epilepsy_data.y = epilepsy_data.y.astype(int)

epilepsy_data= epilepsy_data[epilepsy_data.isnull().any(axis=1)==False]

label = epilepsy_data["y"].astype("category").to_numpy()

label2 = epilepsy_data["y"]

epilepsy_data.drop("y",axis=1,inplace=True)
```

[10]:
```
#extract new features
epilepsy_data = feature_extractbonn(epilepsy_data)
epilepsy_data= epilepsy_data.iloc[:,-13:]
epilepsy_data
```

[10]:

	MAX	MIN	MAX2	MIN2	MAX3	MIN3	MxMnDif	MxMnDif2	\
0	229	-281	52441	78961	12008989	12008989	510	260100	

```
1       513 -1716  263169  2944656  135005697  135005697     2229   4968441
2        80  -126    6400    15876     512000     512000      206     42436
3       -22  -105     484    11025     -10648     -10648       83      6889
4        78  -103    6084    10609     474552     474552      181     32761
...     ...   ...     ...      ...        ...        ...      ...       ...
11495    73   -79    5329     6241     389017     389017      152     23104
11496   471  -388  221841   150544  104487111  104487111      859    737881
11497   121   -90   14641     8100    1771561    1771561      211     44521
11498   148  -157   21904    24649    3241792    3241792      305     93025
11499   110  -108   12100    11664    1331000    1331000      218     47524

             MEAN            VAR         STD      SKEW      KURT
0      -16.910112    9212.342157   95.980947 -0.202033  0.103825
1       28.112360  223886.834762  473.166815 -1.523960  1.414839
2      -44.044944    1963.466895   44.311025  0.498697 -0.212826
3      -68.910112     254.997524   15.968642  0.370253  0.252723
4       -6.651685    1505.606805   38.802149 -0.466683 -0.223217
...           ...            ...         ...       ...       ...
11495    5.157303    1472.754777   38.376487 -0.187119 -0.959651
11496    5.674157   26744.864978  163.538573  0.009116  0.572711
11497    6.752809    1961.554371   44.289439  0.092899 -0.387423
11498  -38.842697    4045.884720   63.607269  0.523610  0.314278
11499   -2.112360    2461.580524   49.614318 -0.024918 -0.712424

[11500 rows x 13 columns]
```

[11]:
```
#normalize the data
normalized = normalization(epilepsy_data, label2)

#shuffle the data
shuffled = shuffling(normalized)

#split the data into train, test, and validation sets
X_train, X_test, y_train, y_test = traintestsplit(shuffled,0.15)
```

[12]:
```
np_train_data = X_train
np_test_data = X_test

np_label_train = y_train
np_label_test = y_test
```

[13]:
```
#saving the train and test sets
np.save("X_train", X_train)
np.save("X_test", X_test)

np.save("y_train", y_train)
np.save("y_test", y_test)
```

[14]:
```
#saving the train and test sets for further use
np.save("np_train_data", np_train_data)
np.save("np_test_data", np_test_data)

np.save("np_label_train", np_label_train)
np.save("np_label_test", np_label_test)
```

[ ]:
```

```

train_test_split_val_Bonn

December 19, 2022

[2]:
```
import pandas as pd
from math import *
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import ConfusionMatrixDisplay
from sklearn.metrics import confusion_matrix
from sklearn.metrics import plot_confusion_matrix
from random import sample
import utils
from utils import shuffling
from utils import normalization
from utils import traintestvalsplit
from utils import feature_extractbonn
import os
```

[3]:
```
#prepare the data for feature engineering and preprocessing
epilepsy_data = pd.read_csv("bonn_epilepsy.csv", sep =",")

epilepsy_data.drop("Unnamed",axis=1,inplace=True)
epilepsy_data.head()
epilepsy_data.y = epilepsy_data.y==1
epilepsy_data.y = epilepsy_data.y.astype(int)

epilepsy_data= epilepsy_data[epilepsy_data.isnull().any(axis=1)==False]

label = epilepsy_data["y"].astype("category").to_numpy()

label2 = epilepsy_data["y"]

epilepsy_data.drop("y",axis=1,inplace=True)
```

[4]:
```
#extract new features
epilepsy_data = feature_extractbonn(epilepsy_data)
epilepsy_data= epilepsy_data.iloc[:,-13:]
epilepsy_data
```

[4]:

	MAX	MIN	MAX2	MIN2	MAX3	MIN3	MxMnDif	MxMnDif2	\
0	229	-281	52441	78961	12008989	12008989	510	260100	

```
1       513 -1716  263169  2944656  135005697  135005697     2229   4968441
2        80  -126    6400    15876     512000     512000      206     42436
3       -22  -105     484    11025     -10648     -10648       83      6889
4        78  -103    6084    10609     474552     474552      181     32761
...     ...   ...     ...      ...        ...        ...      ...       ...
11495    73   -79    5329     6241     389017     389017      152     23104
11496   471  -388  221841   150544  104487111  104487111      859    737881
11497   121   -90   14641     8100    1771561    1771561      211     44521
11498   148  -157   21904    24649    3241792    3241792      305     93025
11499   110  -108   12100    11664    1331000    1331000      218     47524

             MEAN            VAR         STD      SKEW      KURT
0      -16.910112    9212.342157   95.980947 -0.202033  0.103825
1       28.112360  223886.834762  473.166815 -1.523960  1.414839
2      -44.044944    1963.466895   44.311025  0.498697 -0.212826
3      -68.910112     254.997524   15.968642  0.370253  0.252723
4       -6.651685    1505.606805   38.802149 -0.466683 -0.223217
...           ...            ...         ...       ...       ...
11495    5.157303    1472.754777   38.376487 -0.187119 -0.959651
11496    5.674157   26744.864978  163.538573  0.009116  0.572711
11497    6.752809    1961.554371   44.289439  0.092899 -0.387423
11498  -38.842697    4045.884720   63.607269  0.523610  0.314278
11499   -2.112360    2461.580524   49.614318 -0.024918 -0.712424

[11500 rows x 13 columns]
```

[5]:
```
#normalize the data
normalized = normalization(epilepsy_data, label2)

#shuffle the data
shuffled = shuffling(normalized)

#split the data into train, test, and validation sets
X_train, X_test, X_val, y_train, y_test, y_val = traintestvalsplit(shuffled,0.15)
```

[6]:
```
np_train_data = X_train
np_test_data = X_test
np_val_data = X_val

np_label_train = y_train
np_label_test = y_test
np_label_val = y_val
```

[7]:
```
#saving the train and test sets
np.save("X_train", X_train)
np.save("X_test", X_test)
np.save("X_val", X_val)
```

```
np.save("y_train", y_train)
np.save("y_test", y_test)
np.save("y_val", y_val)
```

[8]:
```
#saving the train and test sets for further use
np.save("np_train_data", np_train_data)
np.save("np_test_data", np_test_data)
np.save("np_val_data", np_val_data)

np.save("np_label_train", np_label_train)
np.save("np_label_test", np_label_test)
np.save("np_label_val", np_label_val)
```

[ ]:
```

```

train_test_split_Beirut

December 19, 2022

[2]:
```
import pandas as pd
from math import *
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import ConfusionMatrixDisplay
from sklearn.metrics import confusion_matrix
from sklearn.metrics import plot_confusion_matrix
from random import sample
import os
import utils
from utils import shuffling
from utils import maxminnorm
from utils import traintestsplit
from utils import feature_extractbeirut
```

[3]:
```
#prepare the data for feature engineering and preprocessing
epilepsy_data = pd.read_csv("beirut_epilepsy.csv", sep =",")

epilepsy_data.drop("Unnamed",axis=1,inplace=True)

epilepsy_data = epilepsy_data.loc[(epilepsy_data["y"] == 0) |␣
↪(epilepsy_data["y"] == 1)].reset_index()
epilepsy_data.drop("index",axis=1,inplace=True)

epilepsy_data.y = epilepsy_data.y.astype(int)
epilepsy_data= epilepsy_data[epilepsy_data.isnull().any(axis=1)==False]

label = epilepsy_data["y"].astype("category").to_numpy()
label2 = epilepsy_data["y"]
epilepsy_data.drop("y",axis=1,inplace=True)

epilepsy_data = feature_extractbeirut(epilepsy_data)

epilepsy_data= epilepsy_data.iloc[:,-5:]
epilepsy_data
```

[3]:

	MEAN	VAR	STD	SKEW	KURT
0	-0.000664	0.000344	0.018554	0.075787	0.310464
1	-0.008468	0.003020	0.054959	-10.313749	131.780987
2	-0.000017	0.000021	0.004601	0.156762	0.469875
3	-0.000525	0.000014	0.003799	-1.109231	2.896466
4	-0.000650	0.000088	0.009390	-0.219487	0.011856
...	...	...	...	...	...
6924	0.001217	0.000196	0.013999	-0.002263	0.271676
6925	-0.000209	0.000194	0.013939	0.420493	0.942831
6926	0.000088	0.000008	0.002808	-0.069649	0.507662
6927	0.000447	0.000420	0.020483	0.145546	-0.154202
6928	-0.000224	0.000228	0.015085	0.547034	1.033254

[6929 rows x 5 columns]

[4]:
```
#normalize the data
normalized = maxminnorm(epilepsy_data, label2)

#shuffle the data
shuffled = shuffling(normalized)

#split the data into train and test sets
X_train, X_test, y_train, y_test = traintestsplit(shuffled,0.15)
```

[5]:
```
np_train_data = X_train
np_test_data = X_test

np_label_train = y_train
np_label_test = y_test
```

[6]:
```
#saving the train and test sets
np.save("X_train", X_train)
np.save("X_test", X_test)

np.save("y_train", y_train)
np.save("y_test", y_test)
```

[7]:
```
#saving the train and test sets for further use
np.save("np_train_data", np_train_data)
np.save("np_test_data", np_test_data)

np.save("np_label_train", np_label_train)
np.save("np_label_test", np_label_test)
```

[ ]:
```

```

train_test_split_val_Beirut

December 19, 2022

[19]:
```
import pandas as pd
from math import *
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import ConfusionMatrixDisplay
from sklearn.metrics import confusion_matrix
from sklearn.metrics import plot_confusion_matrix
from random import sample
import utils
from utils import shuffling
from utils import maxminnorm
from utils import traintestvalsplit
from utils import feature_extractbeirut
import os
```

[20]:
```
#prepare the data for feature engineering and preprocessing
epilepsy_data = pd.read_csv("beirut_epilepsy.csv", sep =",")

epilepsy_data.drop("Unnamed",axis=1,inplace=True)

epilepsy_data = epilepsy_data.loc[(epilepsy_data["y"] == 0) |␣
↪(epilepsy_data["y"] == 1)].reset_index()
epilepsy_data.drop("index",axis=1,inplace=True)

epilepsy_data.y = epilepsy_data.y.astype(int)
epilepsy_data= epilepsy_data[epilepsy_data.isnull().any(axis=1)==False]

label = epilepsy_data["y"].astype("category").to_numpy()
label2 = epilepsy_data["y"]
epilepsy_data.drop("y",axis=1,inplace=True)

epilepsy_data = feature_extractbeirut(epilepsy_data)

epilepsy_data= epilepsy_data.iloc[:,-5:]
epilepsy_data
```

[20]:

	MEAN	VAR	STD	SKEW	KURT
0	-0.000664	0.000344	0.018554	0.075787	0.310464
1	-0.008468	0.003020	0.054959	-10.313749	131.780987
2	-0.000017	0.000021	0.004601	0.156762	0.469875
3	-0.000525	0.000014	0.003799	-1.109231	2.896466
4	-0.000650	0.000088	0.009390	-0.219487	0.011856
...	...	...	...	...	...
6924	0.001217	0.000196	0.013999	-0.002263	0.271676
6925	-0.000209	0.000194	0.013939	0.420493	0.942831
6926	0.000088	0.000008	0.002808	-0.069649	0.507662
6927	0.000447	0.000420	0.020483	0.145546	-0.154202
6928	-0.000224	0.000228	0.015085	0.547034	1.033254

[6929 rows x 5 columns]

[21]:
```
#normalize the data
normalized = maxminnorm(epilepsy_data, label2)

#shuffle the data
shuffled = shuffling(normalized)

#split the data into train, test, and validation sets
X_train, X_test, X_val, y_train, y_test, y_val = traintestvalsplit(shuffled,0.15)
```

[22]:
```
np_train_data = X_train
np_test_data = X_test
np_val_data = X_val

np_label_train = y_train
np_label_test = y_test
np_label_val = y_val
```

[23]:
```
#saving the train and test sets
np.save("X_train", X_train)
np.save("X_test", X_test)
np.save("X_val", X_val)

np.save("y_train", y_train)
np.save("y_test", y_test)
np.save("y_val", y_val)
```

[24]:
```
#saving the train and test sets for further use
np.save("np_train_data", np_train_data)
np.save("np_test_data", np_test_data)
np.save("np_val_data", np_val_data)

np.save("np_label_train", np_label_train)
```

```
np.save("np_label_test", np_label_test)
np.save("np_label_val", np_label_val)
```

[ ]: