SMOKE DETECTION

Ahmet Mustafa Baraz

Fuat Işıklan

Abstract—**In this paper, K-Nearest Neighbors (KNN), Logistic Regression, and Neural Network algorithms are implemented for a fire (or smoke) detector. Firstly, we represent the structure of the data so that the features that affect the fire alarm probability of the detector can be understood. As usual, we scale the data to get rid of the large numbers that the computer might not be able to deal with and throw out some correlated features along with the ones that are not related with the fire alarm (UTC etc.). After the data is cleaned with these steps, the mathematical background behind these algorithms are given so that the results obtained from the implementation of the algorithms can be interpreted better. Then, these results are given consecutively and analyzed one by one. It can be seen from these results that all of these algorithms works decent for the smoke detection problem in hand. Thus, we believe that we were able to model the fire detector fine enough.**

I. Introduction

Detecting a natural or human-caused disaster before it occurs is crucial since the disaster may cause some serious physical and psychological damage. The types of these disaster might be earthquake, tsunami, fire, etc. In our project, we choose our problem to be detecting the latter, in other words, detecting whether there is a fire or not. In essence, this problem is a binary classification problem, which means that there are only two outcomes the detector might yield: It will give an alarm or not. The procedure of the detection involves measurements of features like the temperature, air humidity, and carbon dioxide (CO_2) level of the environment. Indeed, the detector is not going to be perfectly able to detect the fire all the time because of reasons such as the detection threshold (i.e. at which levels of the features it will give an alarm) and detection capability (i.e. how much it is capable of measuring the true values of the features) of the detector, and the redundant noise that inevitably enters the detector. Our aim is to implement different machine learning algorithms so that the detector learns how to correctly classify the fire, and as a result, the detection error that the detector presents shall be minimum. To achieve this, we use lots of data obtained from Kaggle with the intent that the detector can be safely used to prevent possible future fires. Further information about the data set and implemented algorithms will be provided in the following sections.

II. Data and Method

An exemplary data for our problem can be seen in Table I where the caption explains what the features physically mean. We have approximately 62600 data in total, however, even before we begin implementing the machine learning algorithms, we have noticed that the column #14 was of no use since it was only an index going from 0 to 62600. Thus, we have excluded it from the design matrix. Moreover, we had to split the last column of the data set from the rest since the last column consists of binary alarm values (i.e. 0 or 1). After these modifications, we have randomly shuffled the rows of the data set. The reason why we have conducted this shuffle is that the initial data set was ordered so that 0 and 1 alarm values were clustered, which means that, for example, using the first few thousand of the data would yield no error in the KNN algorithm. This is because KNN algorithm inspects the K nearest neighbors of a data point so that if all of the neighboring points are of the same class, the algorithm would yield no error, which makes no sense. Here, notice that these initial preparations are valid all of the machine learning algorithms that we are implenting, which are **K-Nearest Neighbors (KNN)**, **Logistic Regression**, and **Neural Network**. Since the logic behind these algorithms are almost entirely different, it is best to examine the analytical backgrounds (i.e. theories) behind them separately in 3 subsections.

Please note that we have clearly assigned each group member (Ahmet Mustafa Baraz and Fuat Işıklan) a certain task. We have decided that the **KNN** algorithm would be implemented by Ahmet Mustafa Baraz, **Logistic Regression** algorithm would be implemented by Fuat Işıklan and the **Neural Network** algorithm would be implemented via joint work. The timeline for our work and the tasks we have assigned to ourselves can be summarized by the **Gantt Chart** given in Fig. 1. Moreover, the reason behind choosing these specific algorithm is related to the implementation difficulty and our smoke detection problem. For instance, it is known that implementing **KNN** and **Logistic Regression** algorithms is pretty straightforward. Besides, given the fact that our problem is binary classification, it seemed to us that it would be logical to utilize these algorithms to make our predictions. Similarly, we have chosen the **Neural Network** algorithm as our "advanced" algorithm since it was relatively easy to implement, as well. Moreover, since we can train a Neural Network to achieve a certain task, we thought that the **Neural Network** algorithm would do good for our binary classification problem. Besides these reasoning, we had, naturally, no prior knowledge of which machine learning algorithm would give the best prediction results since this can be achieved after implementing many different algorithms.

	Index		
Features	1	2	3
UTC	1654733363	1654733364	1654733367
Temp.	20.462	20.477	20.519
Hum.	51.77	52.28	53.50
TVOC	0	0	0
eCO_2	440	420	410
Raw H_2	12479	12486	12491
Raw Eth.	19515	19532	19541
Press.	939.762	939.769	939.755
PM1.0	0.12	0.11	0.08
PM2.5	0.42	0.38	0.29
NC0.5	0.1	0.09	0.07
NC1.0	0.487	0.434	0.336
NC2.5	0.291	0.259	0.201
CNT	32	33	36
Fire Al.	0	0	1

TABLE I: UTC: Timestamp UTC seconds, Temp: Temperature, Hum: Air humidity, TVOC: Organic compounds, eCO_2 : Carbon dioxide concentration, Raw H_2: Molecular Hydrogen, Raw Eth: Ethanol, Press: Air Pressure, PMx: Particular matter size smaller than x, NCx: # concentration of particulate matter, CNT: Sample Counter, Fire Al: Binary label vector, it is 0 if the smoke is not detected.

II-A K-Nearest Neighbors

K-Nearest Neighbors is a supervised machine learning algorithm that, in a nutshell, checks the K nearest neighboring data points of a data point. In a classification problem like ours, the KNN algorithm assigns the data point the most common class value of the K nearest neighbors. For instance, if K=3 and the classes of the K nearest neighbors of a data point is Y={1,0,0}, the data point is taken to be of class 0, since 0 is the most common among the Y set. One should notice here that K should not be an even number since $N_{Y=1} \neq N_{Y=0}$ in order for the algorithm to assign a proper class value. Here $N_{Y=i}$, i=0,1 is the number of K nearest neighbors belonging to class i. One should ask here how to detect the K nearest neighbors of a single data point. This is achieved by taking the **Euclidean distance** between a given data point and others. Unsurprisingly, one should take the K nearest neighbors to be the ones that give the first minimum K Euclidean distance values. Here, Euclidean distance between two data points $\mathbf{x}^{(k)}$ and $\mathbf{x}^{(j)}$ is defined by:

$$\|\mathbf{x}^{(k)}\text{-}\mathbf{x}^{(j)}\| = \sqrt{\sum_{i=1}^{p}((x_i^{(k)} - x_i^{(j)})^2}, \; k,j \in \{1,...,N\}; \; k \neq j, \tag{1}$$

where p is the feature size, N is the sample size, and superscripts (k) and (j) denote the kth and jth data points, respectively. Once one determines the K nearest neighbor data points of a given data point, it is trivial to assign a class value to that specific data point. Correspondingly, the **training error** will be equal to the zero-one loss function[1] that is, for the

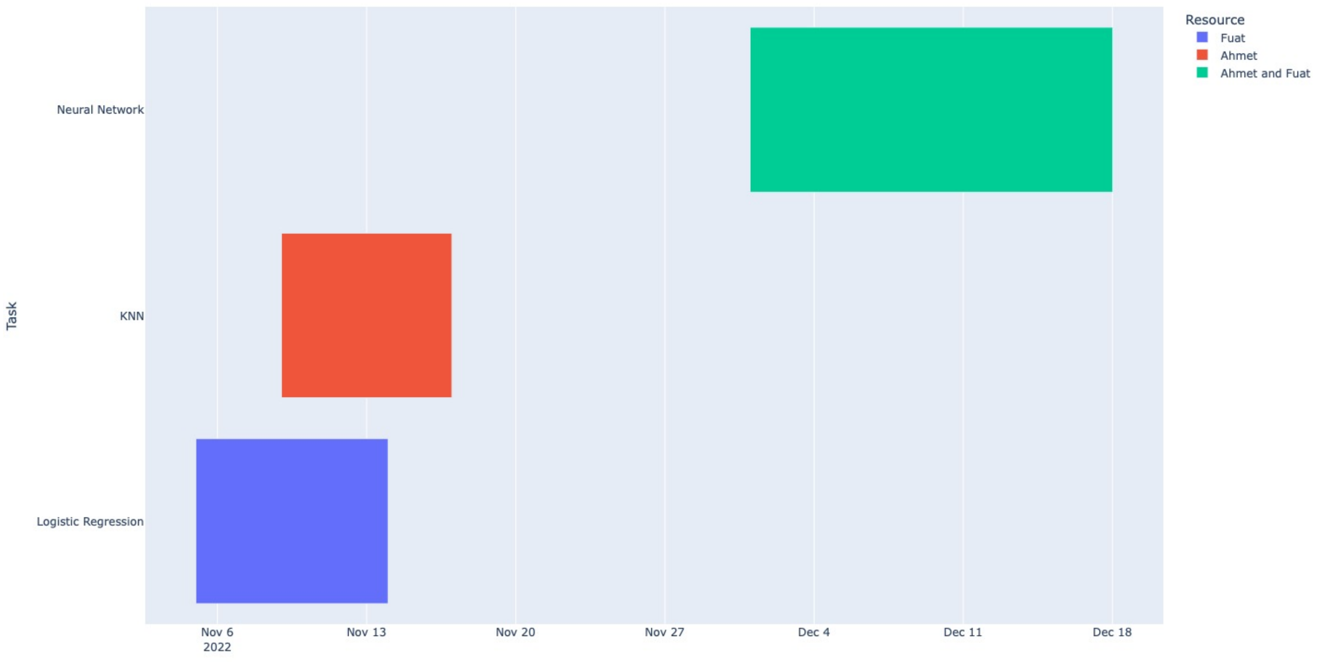

Fig. 1: Gantt Chart that summarizes the process of our project.

training data, given as:

$$\mathcal{L}(\mathbf{y}^{predicted}) = \frac{1}{N}\sum_{i=1}^{N} \delta_{\mathbf{y}_i^{predicted} \neq \mathbf{y}_i^{train}}, \tag{2}$$

where N is the sample size, y_i^{train} are the training scores, $y_i^{predicted}$ are the predicted scores, and $\delta_{i\neq j}$ is the Kronecker delta function which is given as:

$$\delta_{i\neq j} = \begin{cases} 1 & \text{if } i \neq j, \\ 0 & \text{if } i = j. \end{cases} \tag{3}$$

Similarly, the **test error** can be computed by:

$$\mathcal{L}(\mathbf{y}^{predicted}) = \frac{1}{N}\sum_{i=1}^{N} \delta_{\mathbf{y}_i^{predicted} \neq \mathbf{y}_i^{test}}, \tag{4}$$

where $\mathbf{y}_i^{test}$ are the test scores. The optimal value of the K will be computed via the cross-validation method, which will be covered at the end of this section. In the cross-validation method, we will be using the test error definition given in Eq. (16).

II-B Logistic Regression

In the Logistic Regression algorithm, one tries to maximize the log-likelihood function (i.e. the probability that the given design matrix plus coefficient vector generates the outcomes) under a Bernoulli assumption for the outcomes, which in turn amounts to mean minimizing the training error. Using the Bernoulli distribution for the outcomes makes sense because in fact the binary classification problem is rooted in a Bernoulli distribution with only 2 possible outcomes. Moreover, one can use the logistic or sigmoid function to represent the probabilities of this Bernoulli distribution for the outcomes. We have decided to use the sigmoid function for our algorithm, which is given as:[2]

$$\text{S(x)} = \frac{1}{1+e^{-\text{x}}}, \tag{5}$$

where it is not hard to see that the sigmoid function maps S: X $\mapsto$ Y $\in \{0,1\}$, as wanted. Moreover, note also that

$$\text{S}(\mathbf{x}) = \frac{1}{1+e^{\mathbf{x}^\mathsf{T}\underline{\boldsymbol{w}}}} \tag{6}$$

is the form of the sigmoid function for the logistic regression. After one sets the log-likelihood function for the logistic regression, he/she can see that it accepts no closed-form solution for maximum likelihood estimator $\underline{\boldsymbol{w}}$. Thus, one must use an auxiliary algorithm to numerically determine the coefficient vector. In our project, we use the Gradient Descent algorithm (GD), however, one may as well use some other methods such as the Newton-Raphson method. As a side note, the log-likelihood function of the logistic regression is concave, which means that any local maxima represent the global maximum.[3] The proof of this is simple but redundant to share here. The consequence of this is that there may be different local optimum $\underline{\boldsymbol{w}}$ vectors that maximize the same log-likelihood function. This simply means that we should not be worried if the maximizing $\underline{\boldsymbol{w}}$ vector appears to be different each time we run the code for the same data set and the same initial points since they should represent the same global maximum of the log-likelihood function.

As we have stated earlier, maximizing the log-likelihood function is the same as minimizing the corresponding loss function. Thus, the loss function we will be using is the well-known **Cross Entropy Loss** and it is given by:

$$\text{Loss}(\underline{\mathbf{w}}) = \frac{1}{N}\sum_{i=1}^{N}[y^{(i)}\log(S_{\underline{\mathbf{w}}}(x^{(i)})) + (1-y^{(i)}) \times \log(1-S_{\underline{\mathbf{w}}}(x^{(i)}))], \tag{7}$$

where N is the sample size, and y_i and x_i are the ith outcome and feature vectors, respectively. The updating procedure of the **Gradient Descent** (GD) algorithm is given by:

$$\underline{\mathbf{w}}^{(new)} = \underline{\mathbf{w}}^{(old)} - \gamma[\nabla_{\underline{\mathbf{w}}}\text{Loss}], \tag{8}$$

where $\nabla_{\mathbf{w}}$ indicates the gradient with respect to the coefficient vector and γ is the learning rate of the algorithm. Furthermore, the gradient of the loss function can easily be computed as:[4]

$$\nabla_{\underline{\mathbf{w}}}\text{Loss}(\underline{\mathbf{w}}) = \mathbf{X}^\mathsf{T}(S_{\underline{\mathbf{w}}}(\mathbf{X}) - \mathbf{y}), \tag{9}$$

where $\mathbf{X}^\mathsf{T}$ is the transpose of the design matrix. Using Eqs. (8) and (9), we can easily find the optimal coefficient vector $\underline{\mathbf{w}}$ so that we can utilize Eq. (6) to compute the associated probabilities (i.e. scores). After we do this, we can set a threshold value for the score (for instance 0.5) above which the algorithm says that there is a fire. It is good to set the threshold to an average value since one does not want to miss any possible fire alarms as well as cause no false alarms and thus, no panic. In other words, we want to keep the recall and precision values relatively balanced since it is important to detect the fires while also causing not too many false alarms. Moreover, we will use the terms recall and precision for the error analysis in the results section, which are given by:

$$\textbf{Recall} = \frac{\text{TP}}{\text{TP+FN}}, \quad \textbf{Precision} = \frac{\text{TP}}{\text{TP+FP}}, \tag{10}$$

which can be combined to generate an F1 score:

$$\textbf{F1 Score} = 2 * \frac{\text{Precision}\times\text{Recall}}{\text{Precision+Recall}}. \tag{11}$$

Since the F1 score is a harmonic mean in essence and is high only if recall and precision are both high, we can adjust the threshold so that we get a high F1 score. Moreover, we can define the accuracy of the model as:

$$\textbf{Accuracy} = \frac{\text{TP+TN}}{\text{TP+TN+FP+FN}}. \tag{12}$$

The analysis of these values will be done, as mentioned earlier, in the results section.

II-C Neural Network

Neural Networks are multilayer perceptron algorithms in essence. Therefore, in the Neural Network algorithm, there are (more than two) layers that are connected to each other via different weights. These weights can be thought as the weights, for instance, in the Linear Regression model and the layers of Neutral Networks are composed of an **input** layer, **hidden** layers, and an **output** layer. The neurons in these layers may be fully connected (i.e. each neuron is connected to all of the neurons in the adjacent layers) or not. Moreover, one assign an activation function to each neuron in the layers apart from the input layer to produce an output given the weights. It is common to use the **Sigmoid**, **Tanh**, **ReLu**, and **SoftMax** non-linear activation functions to capture the nonlinearities in the models, in other words, to capture difficult tasks.[5] Here, the latter activation function is suitable for one-hot representation in multiclass classification problems. When the outputs (i.e. activations) of the previous layer neurons are used as inputs to the next layer neurons, we get a sequence of outputs. This is called **Forward Propagation**. When the last activation from the output layer is obtained, we get the results of the model. This is where you begin to train to Neural Networks. Since there is a certain task one wants to achieve via Neural Networks, one has to train them to do so. Therefore, there must also be an **error** (or **loss**) function to account for the accuracy of the model. Moreover, this error function can be minimized with respect to the weights of the Neural Network. Since there is no analytical expression for the weights that minimize the error functions in Neural Network algorithms, one uses the **Gradient Descent** method. One can update the weights to minimize the modeling error (i.e. training error) of the algorithm via this method. Pleasingly, the gradient of the error function with respect to the weights of the system can be found through an elegant method called **Backpropagation**. Thence, the Neural Networks can be trained to achieve a certain prediction task such as ours. An example of a Neural Network scheme can be found in Fig. 2 below.

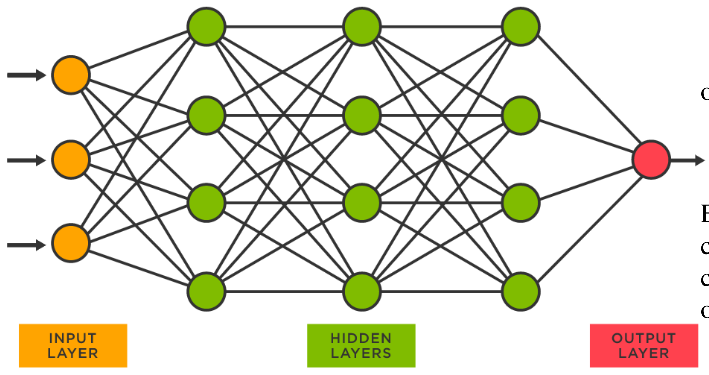

Fig. 2: An exemplary Neural Network with 3 hidden layers. From *What is a Neural Network?*, by Tibco, https://www.tibco.com/reference-center/what-is-a-neural-network.

To understand how Neural Networks work, it is a must to go through the mathematical background. First, we define a loss function:

$$e(\underline{\mathbf{w}}) = \text{Loss}(\mathbf{f}_{\underline{\mathbf{w}}}(\mathbf{x}), \mathbf{y}), \tag{13}$$

and we further define:

$$\delta_j^l = -\frac{\partial e(\underline{\mathbf{w}})}{\partial v_j^l}, \tag{14}$$

where the subscript j and the superscript l denotes the jth neuron in the lth layer. Here, δ can be seen as a measure of error as it will be clear in a moment. If we apply the chain rule to the right-hand side of Eq. (14) with l-1 instead of l, we obtain:

$$\delta_i^{(l-1)} = -\frac{\partial e(\underline{\mathbf{w}})}{\partial v_i^{(l-1)}} = \sum_{j=1}^{d^{(l)}} -\frac{\partial e(\underline{\mathbf{w}})}{\partial v_j^{(l)}} \times \frac{\partial v_j^{(l)}}{\partial x_i^{(l-1)}} \times \frac{\partial x_i^{(l-1)}}{\partial v_i^{(l-1)}}$$
$$= \sum_{j=1}^{d^{(l)}} \delta_j^{(l)} w_{ij}^{(l)} \phi'(v_i^{(l-1)}), \tag{15}$$

where $d^{(l)}$ is the number of neurons in layer l and w_{ij} are the weights between the ith neuron in layer l-1 and jth neuron in layer l. Furthermore, $x_i^{(l)}$ (activations), $v_i^{(l)}$ (activation inputs), and $\phi(x)$ (activation function) can be respectively defined as follows:

$$\phi(x) = \frac{1}{1+e^{-x}}, \tag{16}$$

$$x_i^{(l)} = \phi(v_i^{(l)}), \tag{17}$$

$$v_j^{(l)} = \sum_{i=1}^{d^{(l-1)}} w_{ij} x_i^{(l-1)}. \tag{18}$$

Here, $\phi(x)$ in Eq. (16) is the famous **Sigmoid** function. Additionally, since we are going to work in Python, it is best to express these equations in matrix forms:

$$\boldsymbol{\delta}^{(l-1)} = \underline{\mathbf{w}}\boldsymbol{\delta}^{(l)}\phi'(\mathbf{v}^{(l-1)}), \tag{19}$$

where now $\underline{\mathbf{w}}$ is the weight matrix and bold font indicates a vector. Moreover, the gradients with respect to the weights can be expressed in a very simple manner:

$$\frac{\partial e(\underline{\mathbf{w}})}{\partial w_{ij}^{(l)}} = \frac{\partial e(\underline{\mathbf{w}})}{\partial v_i^{(l)}} \times \frac{\partial v_j^{(l)}}{\partial w_{ij}^{(l)}} = -\delta_j^{(l)} x_i^{(l-1)}, \tag{20}$$

or in matrix-vector form:

$$\frac{\partial \mathbf{e}(\underline{\mathbf{w}})}{\partial \underline{\mathbf{w}}^{(l)}} = -[\boldsymbol{\delta}^{(l)}]^{\mathsf{T}} \mathbf{x}_i^{(l-1)}. \tag{21}$$

Eq.(19) implies that if we can find $\boldsymbol{\delta}^{(L)}$ at the output layer, we can iterate backward to find $\boldsymbol{\delta}$ at the first hidden layer. This is called the **backpropagation** method. We can find $\boldsymbol{\delta}^{(L)}$ at the output layer by using square-loss function:

$$\mathbf{e}(\underline{\mathbf{w}}) = \frac{1}{2}\sum_{i=1}^{d^{(L)}} \mathbf{e}_i^2(\underline{\mathbf{w}}), \text{ where } \mathbf{e}_i(\underline{\mathbf{w}}) = \mathbf{y} - \hat{\mathbf{y}} = \mathbf{y} - \phi(\mathbf{v}^{(L)}). \tag{22}$$

Here, $\mathbf{y}$ and $\hat{\mathbf{y}}$ are the true and predicted labels at the output layer, respectively. As a final step, we can use the definition in the middle of the equality of Eq. (15) to obtain the $\boldsymbol{\delta}^{(L)}$ at the output layer:

$$\boldsymbol{\delta}_{(L)} = -\frac{\partial\left(\frac{1}{2}\sum_{i=1}^{d^{(l}} \mathbf{e}_i^2(\underline{\mathbf{w}})\right)}{\partial \mathbf{v}^{(L)}}, \tag{23}$$

We would like to emphasize that we are using matrix-vector notation for the theory since this is also what we do in the

Python code for the Neural Network algorithm.

Given these equations, one can first **forward propagate** to find the activations at the output layer, and then **backpropagate** to find the gradients with respect to the weights that will be used in the **Gradient Descent** method. This procedure can be visualized in Fig. 3 below.

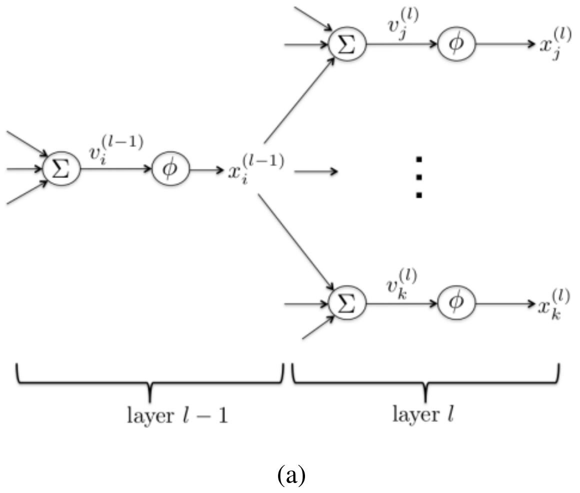
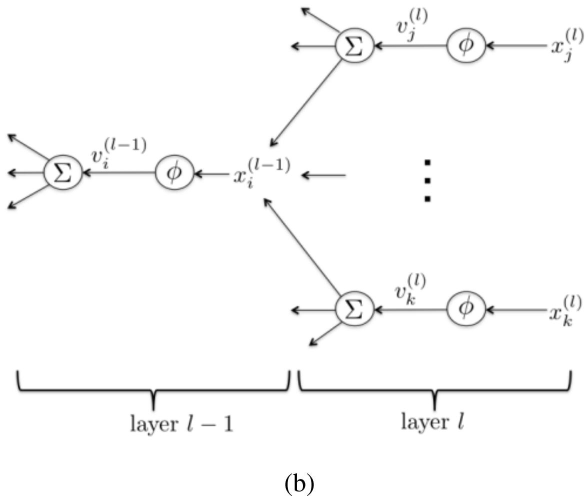

Fig. 3: Forward propagation in (a) and backpropagation in (b).

Lastly, we are in a position to write down the **Gradient Descent** method for updating the weights of the Neural Network and minimizing the model (i.e. training) error:

$$\underline{\mathbf{w}}^{(k+1,l)} = \underline{\mathbf{w}}^{(k,l)} - \gamma[\nabla_{\underline{\mathbf{w}}^{(k,l)}}\mathbf{e}(\underline{\mathbf{w}}^{(k,l)})], \tag{24}$$

where γ is the learning rate of the algorithm, k denotes the iteration number, and l denotes the layer number.

Basically, we have used all of these equations while we were implementing the Neural Network algorithm for our smoke detection problem in Python. Therefore, it must be emphasized that understanding these mathematical models are rather crucial.

II-D Cross-Validation

We will be using K-fold cross-validation to find the optimal K value in the K-Nearest Neighbors (KNN) algorithm. Note that cross-validation helps to find the values of hyper-parameters, which means that it is not much of use to find the optimal value of the $\underline{\mathbf{w}}$ vector for the Logistic Regression. Nonetheless, K-fold cross-validation can still be applied to the Logistic Regression to roughly determine the test error range and mean validation error (i.e. performance of the model). The k-fold cross-validation algorithm can be visualized in Fig. 4 below.

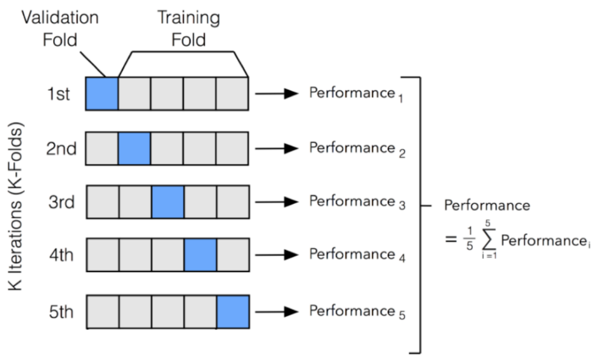

Fig. 4: 5-fold cross-validation. 5 or 10-fold cross-validations are optimal since increasing the fold size increases the variance and computation time of the prediction, which results from the bias-variance trade-off. From "Model Seçimi-K Fold Cross Validation," by G.Öğündür, 2020, (https://medium.com/@gulcanogundur/model-seçimi-k-fold-cross-validation-4635b61f143c). Copyright 2020 by *Medium*.

The logic is that one divides the data set into K sections and uses each time 1 subsection as the validation set, and K-1 subsections as the training set. One may think that increasing fold size will decrease the error and this would be true. However, this would only decrease the training error, which is of no use to us. If the fold size is too high, the computation time will be rather high. Moreover, the model will over-fit the data and the bias will be very low. In turn, the variance in the model will be too high and test error will blow up. This is well-known as the bias-variance trade-off. Thus, it is typically optimal to use 5 or 10-fold cross-validation.[6] We are using 5-fold cross-validation in our project.

The decision procedure for the optimal K value in the K-Nearest Neighbors (KNN) algorithm will be:

1) Divide the dataset into 5,
2) Apply 5-fold CV,
3) Iterate this for different K values,
4) Choose the K value which gives the minimum average CV error.

III. Results

Before yielding any results, we scaled the data so that the Euclidean distance given in Eq. (1) would not be dominated by a large value. In other words, we wanted to make our design matrix unitless. We have used the max-min scaling to scale the data, which is given as:

$$\mathbf{x}^i_{\text{scaled}} = \frac{\mathbf{x}^i - x^i_{\text{min}}}{x^i_{\text{max}} - x^i_{\text{min}}}, \ i = 1, ..., p; \tag{25}$$

where the superscript i denotes the column number, i.e., different features. Correspondingly, x^i_{min} and x^i_{max} are the minimum and maximum values in that column. This scaling is not a must in the logistic regression in general, however, we had to do it anyways since some of the values in our features were rather high, which results in NaN values if we do not do the max-min scaling.

III-A K-Nearest Neighbors

K-Nearest Neighbors (KNN) algorithm works pretty well on our data. We have used 500, 1000, 2000, 4000, and 5000 data for the cross-validation procedure. However, it is best to use the 5000 data among these numbers because the more the data is, the more we will be truly modeling the data (if we disregard the computational cost for now). The error is typically between 0% and 1%. Moreover, the least error is yielded by using **K=1** or **K=3**. In other words, the error **increases** as one increases the K value. It seems that using K=1 is overfitting but there is not exactly a training procedure here. What KNN does is that it takes a test data, checks the surrounding training points, and assigns the test point the most common class among these training points. In other words, we already check the test data to yield the error, which must naturally be the test error. The cross-validation yielded the below results using **5000** data and these results can be summarized in Fig 5 below.

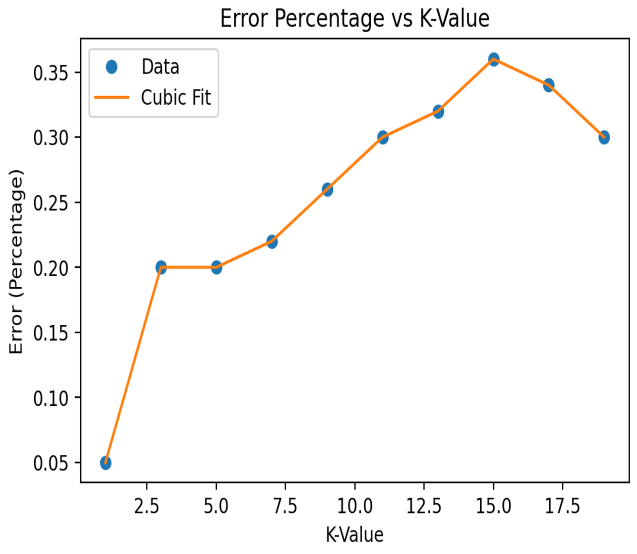

Fig. 5: K-Value of the KNN algorithm vs error percentage. One can roughly say that the error increases as K increases.

From these values, one can say that the KNN algorithm works really well on the data, at least for 5000 data. Using more data is indeed better for accuracy but it takes more time. In other words, the more data you use, the more computational time it costs. This algorithm took more than half an hour to work 5000 data. However, note that increasing the data size increases the computational time exponentially since the code uses many for loops over this data size. Thus, we have decided that using 5000 data is enough. As for the result of the cross-validation, we think that the K=1 case means overfitting the model since the error is almost zero. Thus, we think that **K=3** case is more appropriate for smoke detection. After all, K=3 gives no worse results than the K=1 case, additionally, the cross-validation algorithm sometimes gives K=3 case as the optimal one. This is because the initial shuffle conducted on the data changes the test data at every run of the code. In a nutshell, we choose **K=3** for the KNN algorithm and conclude that it gives excellent results.

III-B Logistic Regression

Logistic Regression algorithm appears to be working well for our data, as well. However, it yields more error than the KNN algorithm. Moreover, Logistic Regression was way more hard than implementing the KNN algorithm since we have faced a few difficulties. These difficulties can be listed below:

- Using the values given in the original data set does not work since some of these values are really high. Thus, we again had to scale the data set, -
- Choosing the initial values of the coefficient vector $\underline{\mathbf{w}}$ affects the convergence of the Gradient Descent algorithm. We picked the initial values from a standard normal distribution since we have scaled the feature values between 0 and 1,
- Setting the learning rate and the number of iterations for the GD algorithm is important. We have picked the learning rate as small as possible and the number of iterations as large as possible while taking the computational time into account,
- After the Gradient Descent algorithm converged, picking the threshold value is up to us. We have picked the probability threshold value as 0.5 since we want to prevent false alarms as well as not miss possible fires, item Lastly, to reduce the test error, we had to reduce the training error. In other words, we have traded some bias in favor of a decrease in the variance.

The optimal hyper-parameter to be estimated for this case is the learning rate. Thus, we conduct the 5-fold **Cross-Validation** procedure to find the most optimal (i.e. that yields the least test error) learning rate. The results of this Cross-Validation procedure are given in Fig 6.

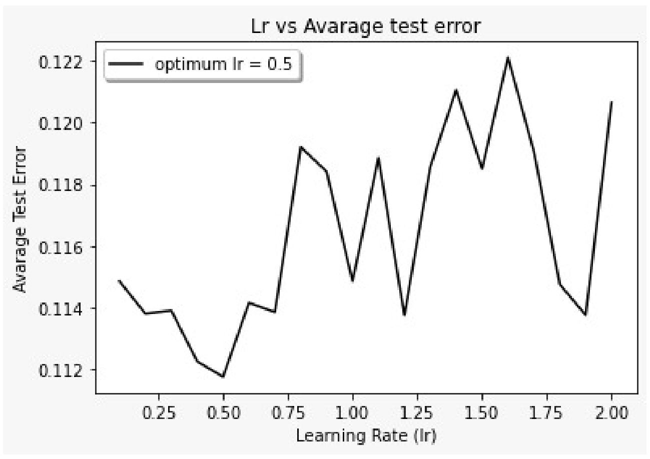

Fig. 6: The 5-fold Cross-Validation procedure yields the optimal learning rate value. Moreover, it also gives us average test errors over a large number of data.

From this figure, one can read the optimal learning rate as **0.5**. Using this value and **n=2000** iterations (epoch) for the Gradient Descent method, we can access the recall precision values, and the F1 scores that are yielded by our model:

Using Optimal Learning Rate Obtained via CV: Training Error= 0.06, Accuracy= 0.89, Recall= 0.92, Precision= 0.86, F1 Score= 0.89.

Notice that we want both the recall and precision values as high as possible since we have set a balanced threshold value for the scores. We can see that we have achieved this by looking at the high F1 scores. Thus, one can conclude that the fire alarm works well with the Logistic Regression model.

One can also put recall precision curves to analyze the behavior of the Logistic Regression model by changing the threshold value. However, we do not want this. We have an a priori demand that the fire alarm should not give false alarms while not missing possible fires. Thus, we have already set our threshold to a perfectly balanced value, which is 0.5.

III-C Neural Network

Neural Network algorithm seems to work well on our data. We have used 20000 data to access the average test error that we obtain through **Cross-Validation** procedure. Since our problem is binary classification, we thought that there was no need to use one-hot representation. When this idea was settled, using **SoftMax** activation function at the output layer seemed to be irrelevant. Thus, we have utilized **Sigmoid** activation function both at the **hidden layers** and at the output layer. As we were informed by the Interim Report feedback to conduct **Cross-Validation** procedure to report the accuracy of our model, we have systematically used **5-fold Cross-Validation** to predict the hyper-parameters such as the learning rate that will be used in the Gradient Descent algorithm and the Neural Network Structure. However, it is important to note that the **Cross-Validation** procedure costs too much computational time, thus, we have decided that we can model our smoke detector using at most 2 hidden layers. In these 2 hidden layers, we have used same number of neurons. For instance, when there was 2 hidden layers, we have iterated the 5-fold Cross-Validation procedure over ”**1 1**”, ”**2 2**”, ”**3 3**”, ”**4 4**”, and ”**5 5**” hidden layer structures. Here, these numbers denote the number of neurons in the hidden layers. Moreover, our data consists of 13 useful features as explained in Section II, naturally, our input layer consists of 13 neurons and the output layer consists of a single neuron that yields a single prediction value.

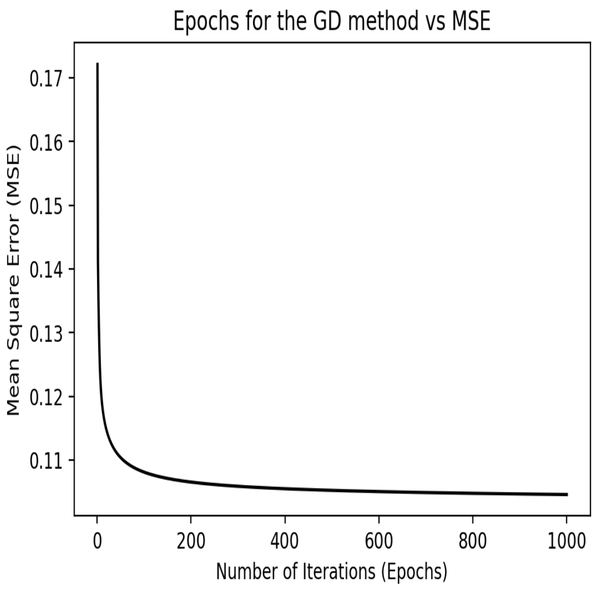

Fig. 7: The training (mean square) error decreases as the number of iterations (epochs) for the Gradient Descent method increases.

While we were training the Neural Network, we constantly monitored the gradient values. By doing this, we observed that the training error began to increase when the gradient value increased to a considerable amount that was usually on the order of 1. This was because our features were already scaled between 0 and 1 so that gradient descent updates with gradient values on the order of 1 were causing the weights to explode. When we were comfortable with the values of the gradients, we observed that the training error was decreasing as the number of iterations (i.e. epochs) for the Gradient Descent method. Additionally, the training error eventually converges to a specific value, which validates the correctness of our implementation of the Gradient Descent method. This can be seen in Fig. 7.

Moreover, the results obtained from the Cross-Validation procedure can be seen in Fig. 8. These results tell us that the optimal test error (i.e. prediction error) is obtained when the number of hidden layers is 1 and there is only a single neuron in the hidden layer. At this optimal value, the learning rate value is also 1.

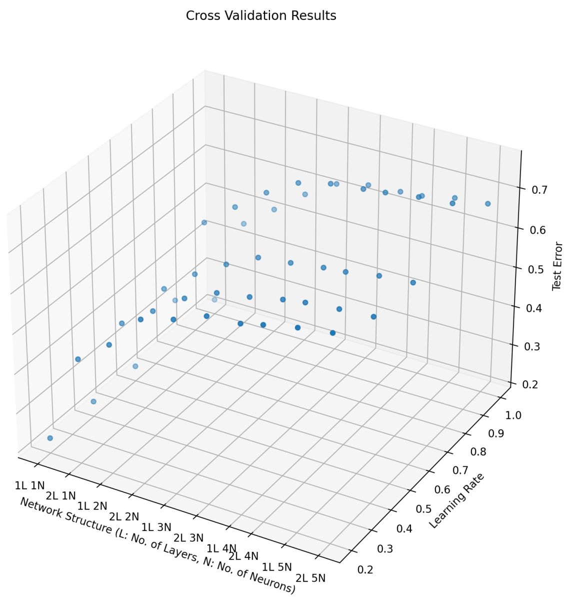

Fig. 8: The results of the 5-fold Cross-Validation procedure as a function of the estimated hyper-parameters, which are the learning rate of the Gradient Descent method and the structure of the Neural Network.

If we set the threshold value for the output decision to be 0.5 just as we did in the other two algorithms, the accuracy, precision, recall, and F1 scores that are yielded by the optimal learning rate and Neural Network structure can be found by Eqs. (10), (11), and (12):

Using Optimal Hyper-Parameters Obtained via CV: Accuracy= 0.795, Recall= 0.98, Precision= 0.80, F1 Score= 0.88.

The accuracy of the model is decent and the F1 score is high since the precision and recall values are high. This means that the Neural Network algorithm works acceptably well for our smoke detection problem.

IV. Conclusion

In this paper, we have covered three machine-learning algorithms that we have implemented for our fire (smoke) detection project. These algorithms are the K-Nearest Neighbors (KNN), Logistic Regression, and Neural Network. We have explained the procedure that we have followed, such as scaling the data set, and the difficulties that we have encountered, such as the convergence of the Gradient Descent method, even before we begin modeling the fire detector upon these algorithms. Furthermore, we have also given several solutions to these problems and as a consequence, the algorithms worked pretty well for our fire detector. According to the results, among these algorithms, the K-Nearest Neighbors (KNN) algorithm seems to work best for our smoke detection problem. There is also a rather important point to emphasize about our data set. We implicitly assume that our data set is correct, which means **y=1** implies there is really a fire, whereas **y=0** implies there is really not a fire. With this assumption, and given the low test errors, we can conclude that we have successfully implemented the KNN, Logistic Regression, and Neural Network algorithms to our fire detector.

References

[1] Sammut, C., & Webb, G. I. (Eds.). (2011). *Encyclopedia of machine learning*. Springer Science Business Media.

[2] Han, J., & Moraga, C. (1995, June). The influence of the sigmoid function parameters on the speed of backpropagation learning. In *International workshop on artificial neural networks* (pp. 195-201). Springer, Berlin, Heidelberg.

[3] Bertsimas, D., & Tsitsiklis, J. N. (1997). *Introduction to linear optimization* (Vol. 6, pp. 479-530). Belmont, MA: Athena Scientific.

[4] Murphy, K. P. (2012). *Machine learning: a probabilistic perspective*. MIT press.

[5] Neter, J., Kutner, M. H., Nachtsheim, C. J., Wasserman, W. (1996). Applied linear statistical models.

[6] Hastie, T., Tibshirani, R., Friedman, J. H., & Friedman, J. H. (2009). *The elements of statistical learning: Data mining, inference, and prediction* (Vol. 2, pp. 1-758). New York: Springer.

V. Appendix

```
−−−−−−−−−−−−KNN Code−−−−−−−−−−−−−−−

import pandas as pd
import numpy as np
from statistics import mode
import matplotlib.pyplot as plt
from scipy import interpolate


df = pd.read_csv('smoke_detection_iot.csv')
df.drop(['Unnamed: 0'], inplace = True, axis = 1)
df.head(n = 20)


# First lets convert data frame to design matrix.
design = df.to_numpy()
# design [:,2]

np.random.shuffle(design) # Shuffle the data once so that it is not "biased". What we mean by biased here is that
                          # original dataset contains full zeros (no smoke) for almost first 3000 data points.
                          # We do not want that since in this case KNN gives no error since a test data point is surrounded by
                               all zeros.


# Finding maximum and minimum values of the feature vector elements.
maximum_values = []
minimum_values=[]
for i in range (len(df.columns.values.tolist())):
    minimum=np.min(design[:,i])
    maximum = np.max(design[:,i])
    maximum_values.append(maximum)
    minimum_values.append(minimum)

# Apply max−min scaling so that Euclidean distance is not dominated by a large term.
for i in range(len(df.columns.values.tolist())):
    design [:, i] = (design [:, i]−minimum_values[i])/(maximum_values[i]−minimum_values[i])

# Define a function that finds the Euclidean distance between test and training data points.
def ed(row0, row1):
    row0 = np.array(row0)
    row1 = np.array(row1)

    dist = np.sqrt(np.sum((row0−row1)**2))

    return dist # Returns the distance.

def KNN(x_train,x_test, y_train, y_test,k):
    error=0
     distance_list_all =[] # This list is used for storing distance lists for all test data.
    for test_row in x_test: # Iteration over test data.
        distance_list =[] # This list is used for storing distance list for each test data.
        for train_row in x_train: # Iteration over train data.
            dist=ed(train_row, test_row) # Euclidean distance between test and training data points.
             distance_list .append(dist)

        distance_list_all .append( distance_list )

     predicted_values_global =[] # This list is used for storing the predicted values for each test data.
    for i in range(len( distance_list_all )): # Note that len( distance_list_all )=len(x_test)


        indices=np.argsort( distance_list_all [i])
        indices=indices[0:k]# Sort by indices and choose the first k of them.
        k_nearest_values=[] # This list is used for storing the classes (0 or 1) of k−nearest neighboors of each test data.
        for j in indices: # Iterate over indices so that you can find which predicted value (0 or 1) that index corresponds to.
```

```
                k_nearest_values .append( y_train [ j ])  # Append the  predicted  value .

            most_common=mode(k_nearest_values) # Choose the most common class (0 or 1) among the k−nearest neighboors.
             predicted_values_global .append(most_common)
            error +=abs( y_test [ i ]−most_common) # Error is found via 0−1  loss  function .

        return  100∗error / len ( y_test )  # Return  the  error  for  k−nearest  neighboors .


def  CrossValidation ( k_fold ) :  # K−fold CV.
        error_global =[]  # This  list  is  used  for  storing  the  mean k−fold error  ( total  error / value  of  k_fold )  for  each K value of
            the  k−nearest  neighboor  algorithm .
        for  i  in  range (1,20,2) :  # Search  the  optimum k value (take  it  as  an odd number).

            error =[]  # This  list  is  for  each  k  value ,  which means it  becomes empty for each k  value .
            for  j  in  range (1, k_fold+1):  # We do the K−fold CV here. We go Test, Train ,  Train ,  Train ,  Train  ;  Train ,  Test ,  Train ,
                Train ,  Train ,   ...;   Train ,  Train ,  Train ,  Train ,  Test .
                print ( j )
                test_data =design [( j−1)∗1000:j∗1000] # Assume 500 test  data .
                train_data =np. delete ( design [0:5000], range (( j−1)∗1000,j∗1000), axis =0)  # Assume 2000 train  data .
                y_train = train_data  [:,  −1]  #  Split  the  predicted  values  (note  they  are  in  the  last  column).
                x_train = train_data  [:,  :−2]  #  Split  the  training  data  from  the  predicted  values .  Note  that  −2  is  there  because  of
                    the  fact  that  column  no.  14  had  nothing  to  do  with  the  data ,  thus  we  have  erased  it .
                y_test = test_data  [:,  −1]  #  Split  the  true  values  (they  are  in  the  last  column).
                x_test = test_data  [:,  :−2]  #  Split  the  test  data  from  the  true  values .  −2  is  there  because  of  the  same argument.
                error .append(KNN(x_train,x_test , y_train , y_test , i ) )  # Run KNN and find the error.
            error_global .append(sum(error) / k_fold )  # Find  the  total  error  via  sum function  and  divide  it  by  the  #  of  folds  ( i .e.
                k_fold )  to  find  average  error ,  then  append  it  into  the  list  .

        sorted_error =np. argsort ( error_global )  # Sort  the  average  CV errors  for  each  k  value  by  their  indices  so  that  we can  easily
            access  the  index  of  the  optimum k  value.

        return  range (1,20,2) [ sorted_error  [0]],  error_global # Return  the  optimum k  value.

print ( CrossValidation (5) )  # Run CV.

# Below can be uncommented to produce the figure  in  the  interim  report .

# data =[0.05,  0.2,  0.2,  0.22000000000000003, 0.26,  0.3,  0.31999999999999995, 0.36, 0.33999999999999997, 0.3]
# x=range (1,20,2)
# xnew=range(1,20,2)
# f= interpolate  . interp1d (x, data , kind=”cubic”)
#  plt . plot (x, data ,’ o ’, xnew,f(xnew),’−’)
#  plt . legend ([’ Data ’,  ’Cubic  Fit ’],  loc  =  ’ best ’)
#  plt . xlabel (’ K−Value’)
#  plt . ylabel (’ Error  (Percentage ) ’)
#  plt .  title  (’ Error  Percentage  vs  K−Value’)

−−−−−−Logistic Regression Code−−−−−−−


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

df  = pd.read_csv( ’ smoke_detection_iot .csv ’ )  # Read  the  data .


# Aranging data ,  deleting  unwanted columns,  split  data  in  X and Y

def  arange_data(df,  n):    # df  =  data ,  n  =  sample  size .

        df  =  df . sample( frac =1). reset_index (drop=True)   #  Shuffle  the  data  frame.
        Y = df[ ’ Fire  Alarm’ ].to_numpy()[0:n]   #  Split  the  labels .
        # Drop  the  unwanted columns:
        df . drop ([ ’CNT’],  inplace=True,  axis=1)
        df . drop ([ ’Unnamed: 0’],  inplace =True,  axis =1)
        df . drop ([ ’ Fire  Alarm’ ],  inplace =True,  axis =1)
```

```
    # df.drop([' UTC'],axis=1,inplace=True)
    # Convert the data to array and apply min−max scaling.
    design = df.to_numpy()
    maximum_values = []
    minimum_values = []
    for i in range(len(df.columns.values. tolist ())):
        minimum = np.min(design[:, i])
        maximum = np.max(design[:, i])
        maximum_values.append(maximum)
        minimum_values.append(minimum)
    for i in range(len(df.columns.values. tolist ())):
        design [:, i] = (design [:, i] − minimum_values[i]) / (maximum_values[i] − minimum_values[i])

    design_1 = np.c_[np.ones((np.shape(design) [0], 1)), design]  # Add the DC terms.

    X = design_1[0:n].reshape((n, 14))  # Declare the design matrix.

    Beta = np.random.normal(0, 1, size=(14, 1)).reshape((14, 1))  # Initialize the parameters for GD.

    # Beta = np.zeros ((14,1) )

    return X, Y, Beta  # Return X set, Y set, and parameters.

# Initialize the train and parameters.
n_train = 20000
l = arange_data(df, n_train )
X_train = l[0]
Y_train = l[1]
param = l[2]


# Define the sigmoid function.
def sigmoid(x):
    return 1 / (1 + np.exp(−x))


# Defining the cross entropy loss function.

# Take the dot product of each row^T with the parameter vector, insert it into the sigmoid function, and sum the
# loss function over all rows.

def cross_entropy(weigth, des, y):
    liste = []

    for i in range(len(des)):
        k = des[i].T.dot(weigth)

        loss = −(y[i] ∗ np.log(sigmoid(k)) + (1 − y[i]) ∗ (1 − np.log(sigmoid(k)))) / np.shape(des)[0]

        liste .append(loss)

    sol = sum(np.array( liste ))

    return sol


# Define the Gradient Descent (GD) algorithm. Use the fact that d(loss)/d(beta)=X^T(sigmoid(x)−y).
def GD(X, Y, Beta, lr, n_iter):
    loss_list = []
    # Betha = []
    for i in range( n_iter ):
        z = np.dot(X.T, (sigmoid(np.dot(X, Beta)) − Y.reshape(len(Y), 1)))
        Beta = Beta − lr ∗ z
        # Beta = Beta − lr ∗ np.dot(X.T, sigmoid(np.dot(X, Beta))−Y)
        # loss = cross_entropy(Beta, X, Y)
        loss_list .append(z)
        # Betha.append(Beta)
    # idx = np.where(abs(np.array( loss_list )) == abs(np.array( loss_list )).min())
    # loss = loss_list [idx [0][0]]

    return Beta, X, z, loss_list   # [Beta,X ,loss , loss_list ]

# Train the model by using Gradient Descent algorithm with learning rate = 0.001 and # of iterations = 1000.
sol = GD(X_train, Y_train, param, 0.001, 2000)

optimum_beta=sol[0]


# Predict the labels (y) usng finding the optimal parameter vector beta.
# Set the threshold to 0.5 and assign class 1 above the threshold and 0 below the threshold.

def score(X, Beta):
    y = []
    s = sigmoid(np.dot(X, Beta))

    for i in range(len(s)):
        if s[i] >= 0.5:
            y.append(1)
        else:
            y.append(0)
    return np.array(y)

predicted_labels_for_train = score(X_train, optimum_beta)


# Error function. False predicitons /sample size
def error(y_t, y):
    n = len(y)
    k = 0
    for i in range(n):
        if y[i] != y_t[i]:
            k = k + 1
    error = k / n

    return error

train_error = error( predicted_labels_for_train ,Y_train)

X_test = arange_data(df,1000)[0]
Y_test = arange_data(df,1000)[1]

predicted_labels_for_test = score(X_test, optimum_beta)
test_error = error( predicted_labels_for_test , Y_test)


# Declare precision and recall.

def precision_recall (X_test, Y_test, Beta):
    s = score(X_test, Beta)
    tp = 0
    fp = 0
    fn = 0
    for i in range(len(s)):
        if s[i] == 1 and Y_test[i] == s[i]:
            tp = tp + 1

        elif s[i] == 1 and Y_test[i] != s[i]:
            fp = fp + 1
        elif s[i] == 0 and Y_test[i] != s[i]:
            fn = fn + 1

    p = tp / (tp + fp)
    r = tp / (tp + fn)

    return p, r, s

# Calculate precision and recall
```

```
pr = precision_recall (X_test, Y_test, optimum_beta)

# CrossValidation for learnng rate lr.

def CrossValidation ( lr_list , X_train, Y_train, k_fold, param):  # K-fold CV.

    error_global = []  # For storing the errors of each ( lr_list , structure_list ) pair.

    for i in lr_list :
        error_list = []
        for k in range(1, k_fold):
            x_test = X_train[(k - 1) * int(len(X_train) / k_fold):k * int(len(X_train) / k_fold)]
            x_train = np.delete(X_train[0: int(len(X_train))],
                                range((k - 1) * int(len(X_train) / k_fold), k * int(len(X_train) / k_fold)),
                                axis=0)  # Assume 2000 train data.

            y_test = Y_train[(k - 1) * int(len(Y_train) / k_fold):k * int(len(Y_train) / k_fold)]
            y_train = np.delete(Y_train[0: int(len(Y_train))],
                                range((k - 1) * int(len(Y_train) / k_fold), k * int(len(Y_train) / k_fold)), axis=0)

            sol = GD(x_train, y_train, param, i, 2000)

            pred = score(x_test, sol[0])

            error_1 = error(pred, y_test)

            error_list .append(error_1)  #

        error_global .append(
            sum( error_list ) / k_fold)  # Find the total error via sum function and divide it by the # of folds (i.e. k_fold)
                to find average error, then append it into the list.

    sorted_error = np.argsort(
        error_global )  # Sort the average CV errors for each ( lr_list , structure_list ) pair by their indices so that we can
            easily access the index of the optimum k value.

    min_index = sorted_error [0]

    min_lr = lr_list [min_index]

    return min_lr, error_global[min_index], error_global

lr_list = [i*0.05 for i in range(1,21)]
k_fold = 5
Validate = CrossValidation ( lr_list , X_train, Y_train, k_fold, param)

fig, ax = plt.subplots()
plt.plot( lr_list , Validate [2], 'k-', label='optimum lr = {}'.format( Validate [0]))
ax. set_title ('Lr vs Avarage test error')
ax. set_xlabel('Learning Rate (lr)')
ax. set_ylabel('Avarage Test Error')
ax.legend(shadow=True, fancybox=True)



---------- Neural Network Code------

import numpy as np
from random import random
from untitled2 import arange_data

import pandas as pd

df = pd.read_csv('smoke_detection_iot.csv')

""" This code implements NN algorithm for smoke detection"""

#################
#################
```

```
#################

# IGNORE BELOW PART #

# def  NeuralNet( input_layer , hidden_layers , output_layer ):

#       # We have to express  the  neural  network  in  terms  of  the  parameters  of  the  function .

#       NN=[] # This  list  will  be something  like  NN=[3,4,5,2],  for  example,  for  input_layer =3,
#             #  hidden_layers =[4,5],  and  output_layer =2.

#       NN.append(input_layer)
#       NN.extend(hidden_layers)  # The elements  of  ”hidden_layers”  will  be  stored  in  ”NN” list .
#       NN.append(output_layer)

#        return  NN

# # Note  that  above  function   constructs   the  Neural  Network.

# #  Start   the  neural  network.

# NN=NeuralNet (14,[7,7,7,7],1)

# Define  the   initial   values  of  the  weights  between the  layers ,   derivatives   of  the  loss   function  w.r.t
# weights,  and the   activation   in  the  input  layer ,  i .e.  the  feature   vectors   in  the   dataset .

###############
###############
###############

# Define  the  forward  propogation .


def  forward_propogation(x_i,weight_matrix,NN): # x_i’s  are  the   activations  .
      global   v_list  # To monitor  the  v_i’s .         # in  the  input  layer .
       activations =[x_i]  # Note  that  the   activations   in  the  input  layers  are  just  the  feature   vectors .
       v_list =[]

      for  i  in  range(len(NN)−1): # Forward propogation  is  done  in  matrix  form via  NN equations.

            v_i= np.dot(x_i,weight_matrix[ i ])

             v_list .append(v_i)



            x_i=sigmoid(v_i)  # These  are  the   activations  .


             activations .append(x_i)  # Save  the   activation   values  at  each  layer .

      return    activations

#   Initialize   the  function  above  to  get  the   activation   values  up  to  the  output  layer .

# Define  the  back  propogation .

# Note  that  ”loss”  input   is  the  error  (y−y_hat)  in  the  output  layer .

def  back_propogation( loss ,  activations  ,weight_matrix,NN):
      global    loss_list   # For  monitoring .
      global    derivative_list   # For  monitoring  the   gradients ,  which  is   crucial  .
       derivative_list  =[]
       loss_list  =[]
      for  i  in  reversed(range(len(NN)−1)): # List   is   reversed   since  we  start   from  the   last   layer .

            # ”Delta”s  are  the  deltas  in  the  NN equations.

            delta=loss∗  first_derivative_of_sigmoid  ( v_list [ i ])
            delta_reshaped  =  delta . reshape( delta . shape [0],  −1). T # We reshape  it  to  have  a  column vector  structure .

            activation = activations [ i ]  # ” activation ”  is  the  activation  value  in  the  i+2’th  layer .

            activations_reshaped = activation . reshape(  activation  . shape [0],  −1) # We reshape  it  to  have  a  column vector  structure .

            derivative_list  . append(np.dot(  activations_reshaped  , delta_reshaped ) )

            loss =np.dot( delta , weight_matrix [ i ]. T)

            loss_list  . append( loss )


    return   derivative_list  [::−1]  # We return  the  reversed   derivative_list   to  match the  shape  with  weight_matrix  in  the
        Gradient  Descent  method.

# Define  the  Gradient  Descent  method.

def GD(n,weight_matrix,  derivative_list  ):  # n  is  the  learning  rate .
    global  a, b # For  monitoring  weight  matrix  and  derivative_list  inside  the  Gradient  Descent.
    for  i  in  range(len(weight_matrix)):
        a=weight_matrix

        b=  derivative_list

        weight_matrix [ i ]+=  derivative_list  [ i ]∗n # Gradient  Descent  is  done here  via  the  gradients  found by backpropogation.

    return  weight_matrix # Return  the  updated  weights.




# Use Sigmoid function  both  in  the  hidden  and output  layers .


def  sigmoid(x):

    return  1/(1+np.exp(−x))

def   first_derivative_of_sigmoid  (x):

    return  (1−sigmoid(x))∗sigmoid(x)


# Define  the  mean square  error  (for  n=1 obviously).
def  mse(y,y_hat):
    return  (y−y_hat)∗∗2

# Define  the  training  function .

def  train ( x_train , y_train ,epoch,weight_matrix ,n,NN): # Epoch  is  the  number of  steps  and n  is  the  learning  rate .


    for  i  in  range(epoch):

        sum_errors  = 0 # For  finding  the  total  training  error .

        for  j ,  inputs  in  enumerate( x_train ): #  Iterate  over  training  data .


            y_j=y_train [ j ]  # Get the  fire  alarm  values  of  the  training  data .

            activations  =forward_propogation( inputs , weight_matrix ,NN) # Forward propogate the  training  data .

            loss =y_j− activations [−1]  # Calculate  the  error  yielded  in  the  output  layer  for  initial  weights .

            grad=back_propogation( loss ,  activations  , weight_matrix ,NN) # Back propogate the  training  data .
```

```
                weight_matrix=GD(n,weight_matrix,grad) # Update the weights.

                sum_errors +=mse(y_j, activations [−1]) # Find the mean square training error .

            # Inform us with at which epoch value we are.
            if i%50==0:
                print ("Mean Square Error: {} at epoch {}".format(sum_errors / len( x_train ), i+1))

        print ("Training is complete !!!! ")
        print ("−−−−−−−−−−−−−−−−−−")
        return weight_matrix # Return the updated weights for the trained model.



def predict ( x_test ,weight_matrix,NN):
        predicted_labels = []
        for i, inputs in enumerate( x_test ):
            act = forward_propogation( inputs , weight_matrix,NN)
            prediction = act[−1] #Final Prediction is the activation in the output layer .

            # Set a threshold in the output_layer above which you label the predicted label as 1 and below which you label 0.
            if prediction > 0.5:
                predicted_labels .append(1)
            else:
                predicted_labels .append(0)
        return predicted_labels # Return the predicted labels .

# Below function computes the test error .

def calculate_error_test (y, y_pred):

        error = 0
        for i in range (len(y)):

            if y[i] != y_pred[i]: # if the predicted label does not match the true label , add error a value of 1.
                error = error + 1
        total_error = error / len(y)
        return total_error

# Below function is for Cross Validation to estimate the optimal NN structure and learning rate .

def CrossValidation ( lr_list , structure_list , data ,epoch,k_fold): # K−fold CV.

        error_global =[] # For storing the errors of each ( lr_list , structure_list ) pair .
        i_j_list =[]

        for i in lr_list :
            error_list =[]

            for j in structure_list :
                for k in range(1,k_fold):


                    weight_matrix=[]
                    for _ in range(len(j)−1):
                        weights=np.random.normal(0,1,size=(j[_], j[_+1]))
                        weight_matrix.append(weights)


                    x=data[0]
                    y=data[1]

                    x_test=x[(k−1)∗int(len(x)/k_fold):k∗int(len(x)/k_fold)] # Assume 500 test data.
                    x_train=np.delete (x[0: int (len(x)) ], range((k−1)∗int(len(x)/k_fold),k∗int(len(x)/k_fold)), axis=0) # Assume 2000
                        train data.

                    y_test=y[(k−1)∗int(len(y)/k_fold):k∗int(len(y)/k_fold)] # Assume 500 test data.
                    y_train=np.delete (y[0: int (len(y)) ], range((k−1)∗int(len(y)/k_fold),k∗int(len(y)/k_fold)), axis=0)
```

```
                        sol= train ( x_train , y_train ,epoch,weight_matrix , i , j )

                        pred = predict ( x_test , sol , j )

                        error = calculate_error_test ( y_test , pred)

                        error_list .append( error ) #

                  i_j_list .append([ i , j ])
                  error_global .append(sum( error_list ) / k_fold) # Find the total error via sum function and divide it by the # of
                      folds (i.e. k_fold) to find average error , then append it into the list .

        sorted_error =np. argsort ( error_global ) # Sort the average CV errors for each ( lr_list , structure_list ) pair by their indices
            so that we can easily access the index of the optimum k value.

        min_index=sorted_error [0]

        min_i_j= i_j_list [min_index]

        return min_i_j, error_global [min_index], error_global , i_j_list

# Define the learning rates that we will iterate over in cross validation .

lr_list =[i*0.2 for i in range (1,6) ]

# Define the NN structures (for instance structure_list =[14,5,5,1] means there are 2 hidden layers with 5 neurons
# and 14 neurons in the input layer , 1 neuron in the output layer ).
structure_list =[]
for i in range (1,6) :
        liste = [13, i , 1]
        liste_1 = [13, i , i , 1]
        structure_list .append( liste )
        structure_list .append( liste_1 )

# Take a slice of the data .

data=arange_data( df,20000)

# We apply 5−fold CV with 100 epochs to reduce the computational cost .
epoch=200
k_fold=5

# Get the results of the NN algorithm.

result =CrossValidation ( lr_list , structure_list , data ,epoch,k_fold)

−−−−−−This Part calculates , precision recall and plots epoch, loss relationship −−−−−−−

import numpy as np
from random import random
from untitled8 import arange_data
import matplotlib . pyplot as plt

import pandas as pd

df = pd.read_csv(’ smoke_detection_iot .csv’ )

”””” This code implements NN algorithm for smoke detection””””

#################
#################
#################

# IGNORE BELOW PART #

# def NeuralNet( input_layer , hidden_layers , output_layer ) :

#       # We have to express the neural network in terms of the parameters of the function .
#       NN=[] # This list will be something like NN=[3,4,5,2], for example, for input_layer =3,
#               # hidden_layers =[4,5], and output_layer =2.

#       NN.append(input_layer)
#       NN.extend(hidden_layers) # The elements of "hidden_layers" will be stored in "NN" list.
#       NN.append(output_layer)

#        return NN

# # Note that above function constructs the Neural Network.

# #  Start the neural network.

# NN=NeuralNet(14,[7,7,7,7],1)

# Define the initial values of the weights between the layers, derivatives of the loss function w.r.t
# weights, and the activation in the input layer, i.e. the feature vectors in the dataset.

###############
###############
###############

# Define the forward propogation.


def forward_propogation(x_i,weight_matrix,NN): # x_i's are the activations .
    global v_list # To monitor the v_i's.        # in the input layer.
     activations =[x_i] # Note that the activations in the input layers are just the feature vectors.
     v_list =[]

    for i in range(len(NN)−1): # Forward propogation is done in matrix form via NN equations.

        v_i= np.dot(x_i,weight_matrix[i])

         v_list.append(v_i)



        x_i=sigmoid(v_i) # These are the activations .


         activations .append(x_i) # Save the activation values at each layer.

    return activations

#  Initialize the function above to get the activation values up to the output layer.

# Define the back propogation.

# Note that "loss" input is the error (y−y_hat) in the output layer.

def back_propogation(loss, activations ,weight_matrix,NN):
    global loss_list # For monitoring.
    global derivative_list # For monitoring the gradients, which is crucial .
     derivative_list =[]
     loss_list =[]
    for i in reversed(range(len(NN)−1)): # List is reversed since we start from the last layer.

        # "Delta"s are the deltas in the NN equations.

        delta=loss∗ first_derivative_of_sigmoid ( v_list [i])

        delta_reshaped = delta.reshape(delta.shape[0], −1).T # We reshape it to have a column vector structure .

         activation = activations [i] # " activation " is the activation value in the i+2'th layer.

         activations_reshaped = activation .reshape( activation .shape[0], −1) # We reshape it to have a column vector structure .
        derivative_list .append(np.dot( activations_reshaped , delta_reshaped ))

        loss=np.dot( delta ,weight_matrix[ i ]. T)

        loss_list .append(loss )


    return  derivative_list [::−1] # We return the reversed derivative_list to match the shape with weight_matrix in the
        Gradient Descent method.

# Define the Gradient Descent method.

def GD(n,weight_matrix, derivative_list ): # n is the learning rate .
    global a,b # For monitoring weight matrix and derivative_list inside the Gradient Descent.
    for i in range(len(weight_matrix)):
        a=weight_matrix

        b= derivative_list

        weight_matrix[i]+= derivative_list [i]∗n # Gradient Descent is done here via the gradients found by backpropogation.

    return weight_matrix # Return the updated weights.




# Use Sigmoid function both in the hidden and output layers .


def sigmoid(x):

    return 1/(1+np.exp(−x))

def  first_derivative_of_sigmoid (x):

    return (1−sigmoid(x))∗sigmoid(x)


# Define the mean square error (for n=1 obviously).
def mse(y,y_hat):
    return (y−y_hat)∗∗2

# Define the training function .

def train ( x_train , y_train ,epoch,weight_matrix,n,NN): # Epoch is the number of steps and n is the learning rate .

    loss_list =[]
    for i in range(epoch):

        sum_errors = 0 # For finding the total training error .

        for j, inputs in enumerate( x_train ): # Iterate over training data .


            y_j=y_train [j] # Get the fire alarm values of the training data .

            activations =forward_propogation( inputs ,weight_matrix,NN) # Forward propogate the training data .

            loss=y_j− activations [−1] # Calculate the error yielded in the output layer for initial weights.

            grad=back_propogation( loss , activations ,weight_matrix,NN) # Back propogate the training data .

            weight_matrix=GD(n,weight_matrix,grad) # Update the weights.

            sum_errors +=mse(y_j, activations [−1]) # Find the mean square training error .


        dummy=sum_errors/len(x_train)
```

```
            loss_list .append(dummy)
            # Inform us with at which epoch value we are.
            if i%50==0:
                print ("Mean Square Error: {} at epoch {}".format(sum_errors / len( x_train ), i+1))

    print ("Training is complete !!!! ")
    print ("−−−−−−−−−−−−−−−−−−")
    return weight_matrix,  loss_list # Return the updated weights for the trained model.



def predict ( x_test ,weight_matrix,NN):
    predicted_labels = []
    for i, inputs in enumerate( x_test ):
        act = forward_propogation( inputs , weight_matrix,NN)
        prediction = act[−1] #Final Prediction is the activation in the output layer .

        # Set a threshold in the output_layer above which you label the predicted label as 1 and below which you label 0.
        if prediction > 0.5:
            predicted_labels .append(1)
        else :
            predicted_labels .append(0)
    return predicted_labels # Return the predicted labels .

# Below function computes the test error .

def calculate_error_test (y, y_pred):

    error = 0
    for i in range (len(y)):

        if y[i] != y_pred[i ]: # if the predicted label does not match the true label , add error a value of 1.
            error = error + 1
    total_error = error / len(y)
    return total_error

# Below function is for Cross Validation to estimate the optimal NN structure and learning rate .


# Take a slice of the data.

data=arange_data(df,20000)
x_train =(data [0]) [0:10000]
x_test =(data [0]) [10000:10200]

y_train =(data [1]) [0:10000]
y_test =(data [1]) [10000:10200]

# We apply 5−fold CV with 100 epochs to reduce the computational cost .
epoch=1000
k_fold=5

# Get the results of the NN algorithm.

NN=[13,1,1]
n=1

weight_matrix=[]
for _ in range(len(NN)−1):
    weights=np.random.normal(0,1, size=(NN[_],NN[_+1]))
    weight_matrix.append(weights)

t=train ( x_train , y_train ,epoch,weight_matrix ,n,NN)
optimum_weight=t[0]

predicted_labels =predict ( x_test ,optimum_weight,NN)

def PrecRecall( y_test ,pred):
    tp=0
    fp=0
    fn=0

    for i in range(len( y_test )):
        if pred[ i]==1 and y_test [ i]==pred[ i ]:
            tp+=1

        elif pred[ i]==1 and y_test [ i ]!=pred[ i ]:
            fp+=1

        elif pred[ i]==0 and y_test [ i ]!=pred[ i ]:
            fn+=1

    p=tp/( tp+fp)
    r=tp/( tp+fn)

    return p,r

 prec_recall =PrecRecall( y_test , predicted_labels )


 test_error = calculate_error_test ( y_test , predicted_labels )

epochh=[i+1 for i in range(0,1000) ]

fig ,ax=plt . subplots ()

plt . plot (epochh, t [1], 'k')

ax. set_xlabel ('Number of Iterations (Epochs)')
ax. set_ylabel ('Mean Square Error (MSE)')
ax. set_title ('Epochs for the GD method vs MSE')


----This part of the code calculates 3-d plots for structers of NN and learning rate with test error-----


import numpy as np
import matplotlib . pyplot as plt


x_axis =[1,4,7,10,13,16,19,21,24,27]

y_axis =[0.2,0.4,0.6,0.8,1]


Z=np.zeros(shape=(5,10) )


X, Y=np.meshgrid(x_axis,y_axis )



row1=[0.22595,0.4519,0.51185,0.5979000000000001,0.6214000000000002,0.6531000000000001,
0.6589500000000001,0.6718000000000001,0.6889500000000001,0.7004500000000000]

row2=[0.22595,0.4519,0.5063500000000001,0.5613,0.59805,0.6113500000000001,0.6284500000000002,0.63675,0.6444500000000001,0.64965]

row3=[0.22595,0.4519,0.5123,0.55975,0.5994,0.6089000000000001,0.6199000000000001,0.6246000000000003,0.6379500000000001,
0.6443500000000001]

row4=[0.31145,0.5374,0.59875,0.6566000000000001,0.7022000000000002,0.7215500000000002,0.7299500000000003,0.7356500000000002,
0.7461500000000002,0.7525000000000002]

row5 =[0.22569999999999996,0.45165,0.5108,0.57155,0.61965,0.63805,0.6428,0.6466,0.663,0.67025]


Z[0]=row1
```

```
Z[1]=row2

Z[2]=row3

Z[3]=row4

Z[4]=row5

fig = plt.figure( figsize =(10,10))

ax = fig.add_subplot(111, projection='3d')

ax. scatter (X,Y,Z)

plt.setp(ax, xticks =[1,4,7,10,13,16,19,21,24,27], xticklabels =['1L 1N', '2L 1N', '1L 2N','2L 2N', '1L 3N', '2L 3N','1L 4N',
    '2L 4N', '1L 5N','2L 5N'])

ax. set_xlabel ('Network Structure (L: No. of Layers, N: No. of Neurons)')
ax. set_ylabel ('Learning Rate')
ax. set_zlabel ('Test Error')

ax. set_title ('Cross Validation Results')
```