EEE-485: Statistical Learning and Data Analytics
The Term Project:
Fire Alarm Prediction

Göktuğ Tolga Ekiz

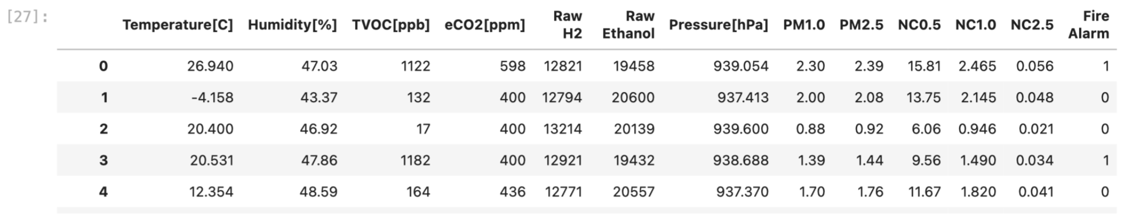
Student Number:
22002648
Group:
62

December 13, 2023

Contents

1 Introduction

The aim of this project is to use the data from a variety of smoke detection sensors and combined it with an AI model to predict the presence of fire. The data is taken from the Smoke Detection dataset[1]. In this project, 3 main algorithms are going to be focused on: PCA, Logistic Regression and Neural Networks. There are going to be some challenges in this project. Firstly, the data set is fairly big (62630 total observations with 14 features). To help alleviate this problem PCA will be implemented. The size issue does not really pose a problem for Logistic Regression due the low number of parameters we have to train. However, for neural networks this is a big issue. This is why batch gradient descent combined with PCA will be implemented for the training of the Neural Network. Since the response variable is a binary variable, Logistic Regression and Neural Networks will be a good fit for this classification problem. It should be noted that even if it was a multi class classification problem, Logistic Regression and Neural Networks would still be a good fit. One other problem is that the predictor variables takes values from 10^{-3} to 10^{5} as seen in the figure below:



[27]:

	Temperature[C]	Humidity[%]	TVOC[ppb]	eCO2[ppm]	Raw H2	Raw Ethanol	Pressure[hPa]	PM1.0	PM2.5	NC0.5	NC1.0	NC2.5	Fire Alarm
0	26.940	47.03	1122	598	12821	19458	939.054	2.30	2.39	15.81	2.465	0.056	1
1	-4.158	43.37	132	400	12794	20600	937.413	2.00	2.08	13.75	2.145	0.048	0
2	20.400	46.92	17	400	13214	20139	939.600	0.88	0.92	6.06	0.946	0.021	0
3	20.531	47.86	1182	400	12921	19432	938.688	1.39	1.44	9.56	1.490	0.034	1
4	12.354	48.59	164	436	12771	20557	937.370	1.70	1.76	11.67	1.820	0.041	0

Figure 1: Example data

However, this problem could easily solved with standardizing the data. Note that this figure is made after some minor pre-processing that is why there are 12 predictor variables.

2 Dataset Description

This data set is important because a Fire Alarm System warns people when smoke, fire, carbon monoxide or other fire-related emergencies are detected. Fire alarm systems saves many lives every year. The following is a description of the variables included in the dataset:

- **UTC:** Timestamp UTC seconds.
- **Temperature[°C]:** Temperature, with °C units.
- **Humidity[%]:** Air humidity during the experiment
- **TVOC[ppb]:** Total Volatile Organic Compounds, measured in ppb where ppb is parts per billion.
- **eCO2[ppm]:** CO2 equivalent concentration, measured in ppm where ppm is parts per million
- **Raw H2:** The amount of Raw Hydrogen in the air.
- **Raw Ethanol:** The amount of Raw Ethanol in the air
- **Pressure[hPa]:** Air pressure in hPa units.
- **PM1.0:** Particulate matter of diameter less than 1.0 micrometer
- **PM2.5:** Particulate matter of diameter less than 2.5 micrometer
- **NC0.5:** Concentration of particulate matter of diameter less than 0.5 micrometer
- **NC1.0:** Concentration of particulate matter of diameter less than 1.0 micrometer
- **NC2.5:** Concentration of particulate matter of diameter less than 2.5 micrometer
- **CNT:** Sample Count

- **Fire Alarm:** 1 if there is a fire, 0 if not.

PM and NC variables might seem similar however, they are not because PM measures the mass concentration while NC measures molecule concentration.

3 Data Preprocessing

Firstly, the response variable was separated from the data. Then, UTC and CNT predictor variables were removed. This was done because these variables did not contain any information about the presence of fire. CNT was a simple counter. For UTC, the original data collectors described UTC as a tool to keep track of the data where a UTC timestamp added to every sensor reading[2]. With UTC and CNT removed, 12 variables are left as predictors. However, since 12 predictors are still a relatively large number, PCA will be used to further pre-process the data. Finally, the data was split into 3 pieces with the ratios of 60:20:20 with the labels training data, validation data, test data respectively.

3.1 PCA Algorithm

PCA is an algorithm that is used to reduce data size while preserving as much of the variance in the dataset as possible for faster and more efficient data analysis. [3] For PCA, data standardization is needed as all of the predictor variables' variances are different and this can cause a few of the large valued variables to dominate the total variance explained metric. To standardize the following must be done: Let X be our data matrix with dimensions nxm. Where n is number of observations and m the number of predictor variables.

$$\mu_i = \sum_{j=1}^{n} \frac{X_{j,i}}{n}$$

and let σ_i= Standard deviation of i th column of X

$$\mu = [\mu_1, ..., \mu_m]^T and \sigma = [\sigma_1, ..., \sigma_m]^T$$

if we let the vector v be all ones with dimensions nx1,

$$X_{0mean} = X - v\mu^T$$

Note that $v\mu^T$ is also called the outer product of v and μ

Finally to standardize X, we should perform an element by element division to X_{0mean} with $v\sigma^T$ which will make all of the columns 0 mean and unit variance. Thus after all these steps we will end up with $\hat{X}$ which is the standardized version of X. After this step, the directions of the major axes of variance should be found. Turns out, these are directions of the eigenvectors of Σ which is the sample covariance matrix. Where:

$$\Sigma = \frac{1}{n}\hat{X}^T\hat{X}$$

When we sort the eigenvectors according to their eigenvalues the first k eigenvectors we pick, are the major axes of variance. The value k is decided by looking at the total variance explained by the first k principal components. It should be stated that the eigenvalues of these vectors are the variances that the specific eigenvector explains. For our case, we can look at the figure below:

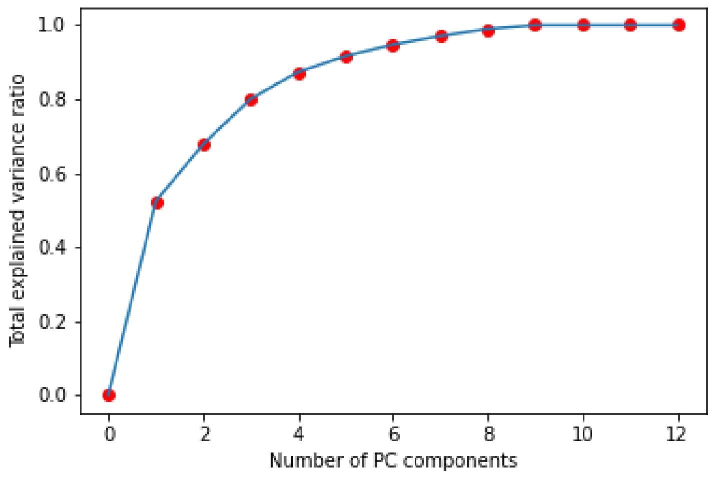


Figure 2 : Total variance explained

```
If we use the first 0 Principal Components, the total explained variance is:0.0
If we use the first 1 Principal Components, the total explained variance is:0.5233308417468325
If we use the first 2 Principal Components, the total explained variance is:0.6777496570809234
If we use the first 3 Principal Components, the total explained variance is:0.8004172346663404
If we use the first 4 Principal Components, the total explained variance is:0.8730621987632288
If we use the first 5 Principal Components, the total explained variance is:0.9161051464600123
If we use the first 6 Principal Components, the total explained variance is:0.9471111945814091
If we use the first 7 Principal Components, the total explained variance is:0.9705926851577317
If we use the first 8 Principal Components, the total explained variance is:0.9894975349176769
If we use the first 9 Principal Components, the total explained variance is:0.9999999999989929
If we use the first 10 Principal Components, the total explained variance is:0.9999999999997932
If we use the first 11 Principal Components, the total explained variance is:0.9999999999999631
If we use the first 12 Principal Components, the total explained variance is:1.0
```

Picking k=6 would be a decent choice as %95 total variance explained is a common practice. Hence, our new projected matrix will explain %94.71 of the variance in our original data. To calculate the projected matrix which will be nxk where k=6, let :

$$E = [e_1, ...e_k]$$

As stated before k=6, Then:

$$X_{projection} = \hat{X}E$$

This matrix multiplication projects $\hat{X}$ to the first 6 eigenvectors of Σ because it is mathematically equivalent to taking each row of $\hat{X}$ and dotting with each column of E. A better visual for this would be the following figure:

$$A \bullet B = \begin{array}{c} \vec{r_1} \rightarrow \\ \vec{r_2} \rightarrow \end{array} \begin{bmatrix} a & b \\ c & d \end{bmatrix} \bullet \begin{array}{c} \begin{array}{cc} \vec{c_1} & \vec{c_2} \\ \downarrow & \downarrow \end{array} \\ \begin{bmatrix} e & f \\ g & h \end{bmatrix} \end{array} = \begin{bmatrix} \vec{r_1} \bullet \vec{c_1} & \vec{r_1} \bullet \vec{c_2} \\ \vec{r_2} \bullet \vec{c_1} & \vec{r_2} \bullet \vec{c_2} \end{bmatrix}$$

Since each eigenvector e_i is unit length, the dot product between the eigenvector and an observation row will project the observation the eigenvector. Hence, PCA algorithm was successfully applied to the data matrix X.

4 Logistic Regression

Logistic regression is a supervised machine learning model which is used for classification. In our project, logistic regression is going to be used for a binary classification. Some assumptions are made for logistic regression. Firstly, it is assumed that the response variable for each observation are independent from each other. Secondly, we assume that the response variable Y, follows a Bernoulli distribution.

Let,

$$g(z) = \frac{1}{1+e^{-z}}$$

And our hypothesis be:

$$h_\beta(x) = g(\beta^T x) = \frac{1}{1+e^{-\beta^T x}}$$

With these given, we finally assume that:

$$P(y=1 \mid x;\beta) = h_\beta(x)$$

$$P(y=0 \mid x;\beta) = 1 - h_\beta(x)$$

These two lines could be summarized to one with the following:

$$p(y \mid x;\beta) = (h_\beta(x))^y(1-h_\beta(x))^{1-y}$$

Since we have n training examples the likelihood function could be written as :

$$L(\beta) = \prod_{i=1}^{n}(h_\beta(x^i))^{y^{(i)}}(1-h_\beta(x^i))^{1-y^i}$$

Where i is the ith training example. Our objective for Logistic regression is to maximize Likelihood. An easier option is to maximize log likelihood as log is a monotonic function. Log likelihood is given as:

$$l(\beta) = \sum_{i=1}^{n} y^i log(h_\beta(x^i)) + (1-y^i)log(1-h_\beta(x^i))$$

It should be restated that the superscript i just indicates the number of the training example. For gradient descent we need to define a cost function which will be:

$$J(\beta) = \frac{-1}{n}\sum_{i=1}^{n} y^i log(h_\beta(x^i)) + (1-y^i)log(1-h_\beta(x^i))$$

Since minimizing this cost function is equivalent to maximizing the likelihood. Unfortunately there is no closed form solution to minimize this cost function. Note that the $J(\beta)$ is also refereed as cross entropy loss.

4.1 Training using Gradient Descent

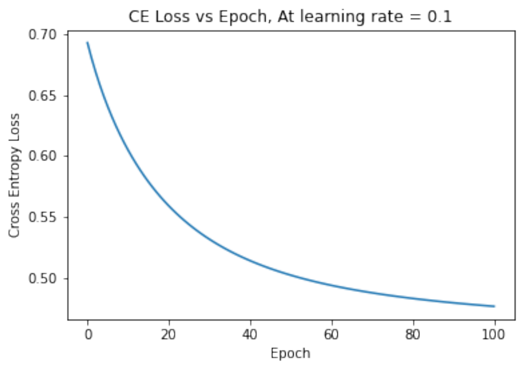
As stated above there is no closed form solution for logistic regression such that the cost function is minimized. Thus, we need to apply an iterative algorithm such as gradient descent to find the β values such that the minimum cost, or in other words, the highest likelihood is achieved. Gradient descent is applied the following way:

$$\beta_j := \beta_j - \alpha\frac{\partial}{\partial\beta_j}J(\beta_0,...\beta_k)$$

We simultaneously update β_j for j= 0,1,..k and α is the training rate hyperparameter.

$$\beta_j := \beta_j - \alpha\frac{1}{n}\sum_{i=1}^{n}(h_\beta(x^{(i)}) - y^{(i)})x_j^{(i)}$$

Since logistic regression converged both with and without PCA in reasonable times, mini batch gradient descent was not used. However, it is very unlikely that this will be the case in Neural Networks. To determine the learning rate, some values of α was used. Here are some figures:



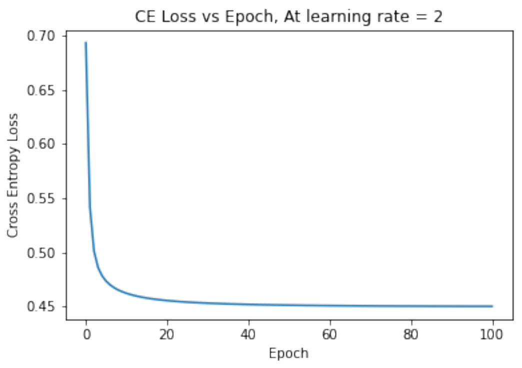


Figure 3: Loss at learning rate   and   Figure 4: Loss at learning rate 2

As seen above it could be seen that learning rate 2 makes the loss converge faster. Since the loss constantly decreases, it could be deduced that the learning rate is not too high. For our case, we picked α=2. Figure 3 & 4 are from Logistic regression with PCA, however, the same holds for Logistic regression without PCA.

5 Neural Network

In this project another machine learning algorithm that was used was a Neural Network or a feed-forward Neural Network (FNN) to be exact. Feed-forward Neural Network, also known as multi layer perception (MLP), is a network where information only flows forward, meaning there is no feedback loops. The general structure is the following :

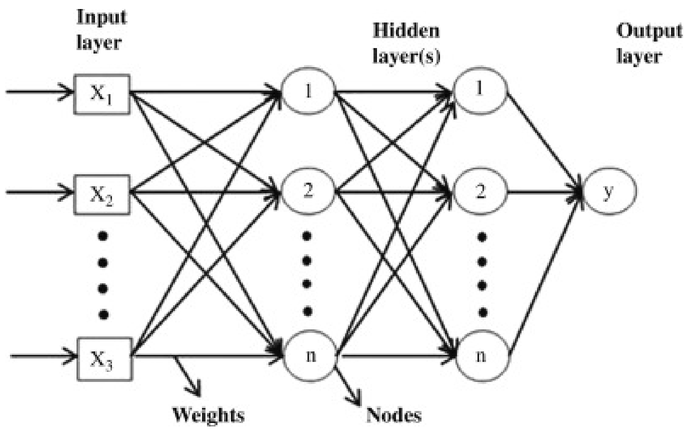


Figure 5: General structure of an FNN[4]

For the FNN used in this project, the architecture is the following: 6 input neurons, 2 hidden layers with 18 neurons each, 1 output neuron as we deal with a binary classification problem. The activation function that was used for the hidden layers was RelU and for the output activation function, sigmoid was used. Some benefits of RelU as a hidden layer activation are: it introduces non-linearity, it is efficient and finally RelU and it's derivative are very simple functions as shown below:

$$RelU(x)\begin{cases} x & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases} \tag{1}$$

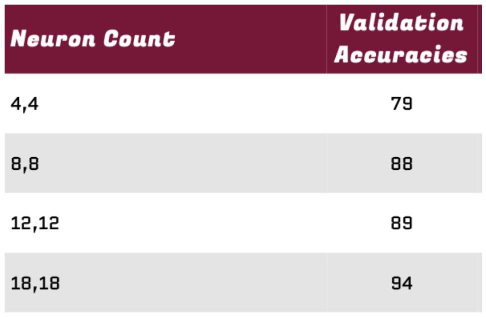
$$f'(x)\begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases} \tag{2}$$

Sigmoid was used as the output activation as it outputs a "probability" where the function is defined as the following:

$$g(z) = \frac{1}{1+e^{-z}}$$

The 6 inputs that were used are the ones that were generated by using PCA in Section 3.1. PCA gains more importance in this section because a neural network is very computationally expensive to train and dimensionality reduction alleviates this problem.

Additionally, the 18 neurons on the hidden layers were not selected randomly. Since there is no ”right” neuron count in the hidden layers, trial and error method was used. Under 12 neurons resulted in some under fitting where the training and validation accuracy were relatively low.



Neuron Count	Validation Accuracies
4,4	79
8,8	88
12,12	89
18,18	94

Figure 6: Neuron Count vs Accuracy

Figure 6 accuracies were found using Stochastic Gradient Descent with 15000 epoch at learning rate 0.001. On the other hand, when 20 neurons were exceeded some of the inputs to the sigmoid activation became too large. This caused math library to give out an ” OverflowError: math range error”. Thus, the neuron count hyper-parameter was set to 18 for both hidden layers as it was a healthy medium between these issues.

Loss function is selected as Cross- Entropy(CE) loss because it works well with a binary calcification problem. CE loss is defined as the following for the Neural Network.

$$J(o,y) = ylog(o) + (1-y)log(1-o)$$

The neural network is trained using stochastic gradient descent(SGD) and mini-batch gradient descent. Batch gradient descent was avoided since the training data set contains 37000 data. This would make the learning process extremely difficult since the entire training data is considered before taking a step in the direction of gradient. However, SGD considers only one training example to calculate the gradient. This causes very a noisy convergence. Mini batch gradient descent is best of both worlds where it considers a chunk of the data, this chunk is 100 in our case which makes the convergence less noisy. An illustration is shown below:

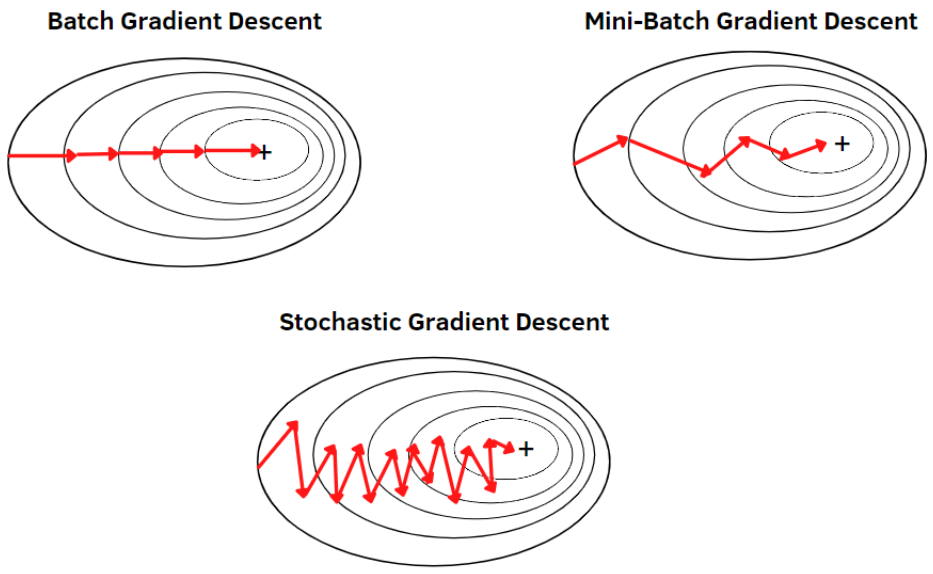


Figure 7: Versions of Gradient descen[5]]

The training process with gradient descent for FNN is very similar to the one that was used in Logistic Regression(Section 4). Only this time, instead of taking the gradient with respect to beta parameters, the weights and biases will be used.

$$W := W - \alpha \frac{\partial}{\partial W} J$$

And,

$$B := B - \alpha \frac{\partial}{\partial B} J$$

Where α is the learning rate.

$$\delta_n^L = \frac{\partial}{\partial z_n^L} J = h'(z_n^L) \sum_m \delta_m^{L+1} w_{nm}^{L+1}$$

Where h' is the activation function derivative and,

$$z^L = w^L a^{L-1} + b^L$$

Firstly, Stochastic Gradient Descent was used.

$$\frac{\partial J}{\partial w^L} = a^{L-1} \cdot (\delta^L)^t$$

$$\frac{\partial J}{\partial b^L} = \delta^L$$

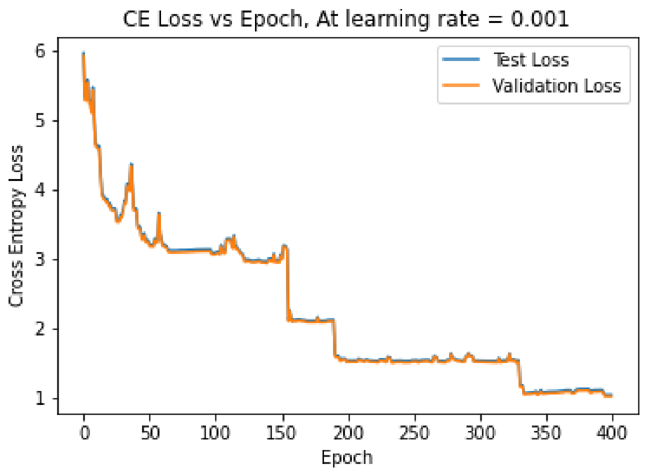


Figure 8: SGD with 0.001 learning rate

Notice that the training process is noisy , very slow and actually did not converge at 500 epoch. A solution to the slowness could be increasing the learning rate, turning the training process to Mini Batch gradient descent or implementing momentum. Firstly, increasing the learning rate was implemented. As seen above, the training loss and validation loss are almost identical, this means that the training set represents the validation set very well. When closely looked, it could be seen that there are very minor differences between the two losses where usually training and validation loss differs from each other by ± 0.1 This meant that the neural network was not over fitting the training data. Thus, algorithms such as weight decay does not need to be implemented.

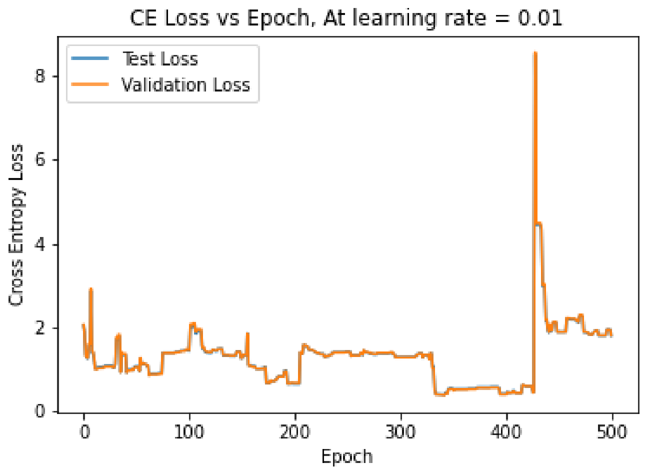


Figure 9: SGD with 0.01 learning rate

After implementing a increase of the learning rate, the result turned out even worse. The noise became extremely high. This was due to the large fluctuations in the weights and biases and since the network is trained on one training data for each iteration, sometimes it caused massive error as seen above. To mitigate this, Mini batches were used for the following training example:

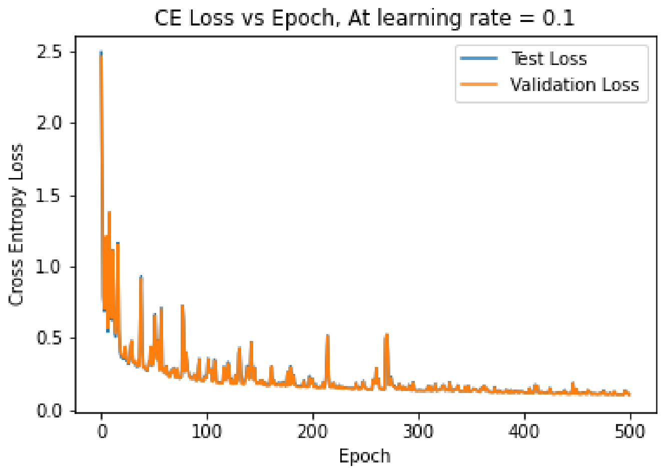


Figure 10: Mini-Batch Gradient Descent with 0.1 learning rate

As seen above, the noise was severely reduced and the loss converged much faster compared to SGD algorithm. Since it converged relatively fast, momentum was not needed.

6 Performance Figures

6.1 Logistic Regression: With and without PCA

6.1.1 With PCA

- The accuarcy for the traing data was: 0.8129756772579702, and the accuarcy for the validation data was: 0.8105540475810314. The training and validation accuracy was extremely close. This meant that the model was not over fitting the training data so we did not have to add regularization.
- It took 5 minutes 22 seconds for gradient descent to converge.

- It took 200 epoch to converge however it could also be considered converged at 100 iterations as well we ran it for 200 because it didn't take too much time. (Refer to Figure 4)

6.1.2 Without PCA

- The accuracy for the training data is: 0.8974666027995104, and the accuracy for the validation data is: 0.894938527862047. The training and validation accuracy was extremely close. This meant that the model was not over fitting the training data so we did not have to add regularization to this model either.
- However due to using 12 predictor variables instead of 6, this model took 9 min 17 seconds to converge which was 200 epoch as well.

6.2 Neural Network

6.2.1 SGD

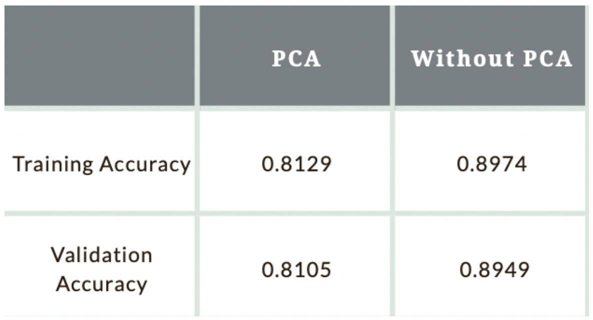
- SGD did not fully converge in 500 iterations and it was extremely noisy. The train loss was: 1.0362 and validation loss: 1.0157. Accuracy of training data was: 0.83056, Validation Accuracy :0.82915 with a run time of 8min 22s.
- The training and validation accuracy was extremely close. This meant the model was not over fitting the training data and there was no need for regularization.
- It should be noted that the reason training took this long is because training and validation loss as well as accuracy was calculated for the entire data set (around 60.000 data) in each iteration. These calculations were the main ones that extended the time because they were not optimized and used a lot of for loops. However, the SGD and the Neural Network algorithms were very optimized as they were written about 6 times until they worked. Thus, they were made better along the way. In reality, the actual training, without loss or accuracy calculations, took about 1 -2 seconds.

6.2.2 Mini Batch Gradient Descent

- Mini batch Gradient Descent converged in 500 epoch with relatively low noise as seen in Figure 10.
- Training loss: 0.10796 and Validation loss was: 0.10576. Train accuracy was :0.96359 and Validation accuracy was :0.96399 where it took 10 min 39 s.
- Again, the majority of this time was used to calculate loss and accuracy.

7 Conclusion

7.1 For Logistic Regression



	PCA	Without PCA
Training Accuracy	0.8129	0.8974
Validation Accuracy	0.8105	0.8949

Figure 5: Logistic Regression accuracy with and without PCA

We can notice that the accuracy on this one was better than the PCA applied one. This information is not surprising as we lost nearly %5 of the variation when we applied PCA and it turns out it was not just noise. As mentioned above the model with PCA applied, converged significantly faster than the other one. Since our hyper-parameter tuning was over, we calculated the test accuracy for both models. PCA one was better with time and computational usage with a test accuracy of: 0.81079. Whereas the one without PCA was better in accuracy by around %8 percent with a test accuracy of 0.89414.

7.2 For Neural Network

The Feed-forward neural network, worked best with 18,18 neurons in the two hidden layers. The mini-batch gradient descent, converged fast and with low noise. Since it converged fast, momentum or Adam optimizer was not used. Additionally, since validation and training accuracy was nearly the same, no regularization methods were used. After tuning all the hyper-parameters as described above, the final test accuracy for the FNN is : 0.96128. In conclusion, compared to Logistic regression, it could be seen that neural networks are far superior for this dataset.

8 Appendix

8.1 Data Preprocessing

[1]:
```python
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import math
```

[ ]:
```python
"""
df = pd.read_csv ('smoke_detection_iot.csv')
df= df.sample(frac=1)         #suffles the data, this is because they were sorted by Fire alarm

df=df.drop(['Unnamed: 0','UTC','CNT'],axis=1)


df.head(37578).to_csv('train.csv')
df.tail(25052).to_csv('test_and_validation.csv')  #first half of test_and_validation is for test, the second half
                                                  #60-20-20 train,test,validation split.
df.to_csv('suffeled.csv')

#So that if I accidentally run the program, I won't accidentally train on test or validation data
"""
```

[2]:
```python
df = pd.read_csv ('train.csv')
df=df.drop(['Unnamed: 0'], axis=1)
X_d=df.drop(['Fire Alarm'], axis=1) #_d means dataframe version of the matrix
Y_d=df['Fire Alarm']
```

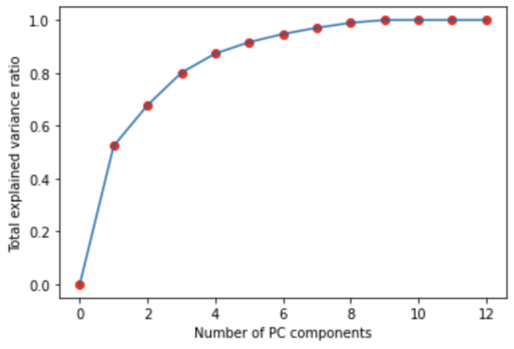
[3]:
```python
X =X_d.to_numpy() #Normalize X
Y=Y_d.to_numpy()
def normalize(X):
    mean=[]
    std=[]
    for i in range(12):
        mean.append(np.mean(X[:,i]))
        std.append(np.std(X[:,i]))
    mean=np.array(mean)
    std=np.array(std)
    a=np.ones((len(X[:,1])))
    X_std=np.divide(X-np.outer(a,mean),np.outer(a,std))
    return X_std
X_std=normalize(X)
```

8.2 PCA

[37]:
```
#PCA
cov_X= (np.matmul(X_std.T,X_std))/len(X_std[:,1]) #Same result with np.cov(X_std,rowvar=False)
eigval_X ,eigvec_X =np.linalg.eig(cov_X)
index=eigval_X.argsort()[::-1] #to sort eigenvalues and vectors (max to min)
eigval_X=eigval_X[index]
eigvec_X=eigvec_X[:,index]
total_var_explained=sum(eigval_X)

cum_var_explained=[0]
for i in range(12):
    cum_var_explained.append(sum(eigval_X[:i+1]))
total_var_explained_ratio=cum_var_explained/total_var_explained
plt.plot([x for x in range(13)],total_var_explained_ratio , 'ro')
plt.plot([x for x in range(13)],total_var_explained_ratio )    #I picked 6 PC because the total var explained was
plt.xlabel('Number of PC components')
plt.ylabel('Total explained variance ratio')
for i in range(13):
    print(f'If we use the first {i} Principal Components, the total explained variance is:{total_var_explained_rat
#plt.savefig('total_var_explained.png')
```

```
If we use the first 0 Principal Components, the total explained variance is:0.0
If we use the first 1 Principal Components, the total explained variance is:0.5233308417468325
If we use the first 2 Principal Components, the total explained variance is:0.6777496570809234
If we use the first 3 Principal Components, the total explained variance is:0.8004172346663404
If we use the first 4 Principal Components, the total explained variance is:0.8730621987632288
If we use the first 5 Principal Components, the total explained variance is:0.9161051464600123
If we use the first 6 Principal Components, the total explained variance is:0.9471111945814091
If we use the first 7 Principal Components, the total explained variance is:0.9705926851577317
If we use the first 8 Principal Components, the total explained variance is:0.9894975349176769
If we use the first 9 Principal Components, the total explained variance is:0.9999999999989929
If we use the first 10 Principal Components, the total explained variance is:0.9999999999997932
If we use the first 11 Principal Components, the total explained variance is:0.9999999999999631
If we use the first 12 Principal Components, the total explained variance is:1.0
```



[5]:
```
X_prjct=np.dot(X_std,eigvec_X[:,:6])#data projected to 6 different eigenvectors where the eigen vectors are #len=1
#the new 6 columns represent the length of the projection to the PC where it goes PC1,PC2...PC6
#(to go back to the original data, we can use np.matmul(X_prjct,eigvec_X[:,:6].T)) Note that there will be lost in
X_dsgn=np.c_[ np.ones(len(X_prjct[:,1])),X_prjct]
```

8.3 Logistic Regression

•[6]:

```python
#Logistic regression
#useful functions
def sigmoid(x):
    return 1/(1 + math.exp(-x))

def cross_entropy_loss_total(X,Y,b):#Y-> correct classification, X-> Data with length of features + 1(design), b->
    total=0
    for i in range(len(Y)):
        total+=Y[i]*(math.log(sigmoid(np.dot(b,X[i,:])))) + (1-Y[i])*(math.log(1-sigmoid(np.dot(b,X[i,:]))))
    return (-1/len(X[:,0]))*total

def gradient(X,Y,b,b_j): #b_j is the subscript of beta that the loss is differentiated to. (i.e. 0,1,2,3..)
    total=0
    for i in range(len(Y)):
        total+= (sigmoid(np.dot(b,X[i,:]))-Y[i])*X[i,b_j]
    return (1/len(Y))*total

def gradient_descent(learning_rate,X,Y,b):
    c=b.copy() #This line is because it gradient function uses b, and we don't want it changeing during a single e
    for j in range(len(b)):
        b[j]=b[j]-learning_rate*gradient(X,Y,c,j)
    return b
def prediction(X_dsgn,b,threshold):
    y_prediction=[]
    for i in range(len(X_dsgn[:,1])):
        if sigmoid(np.matmul(b,X_dsgn[i,:]))> threshold:
            y_prediction.append(1)
        else:
            y_prediction.append(0)
    return y_prediction
def accuracy(Y,X_dsgn,threshold,b):
    total=0
    for i in range(len(Y)):
        pred=0
        if sigmoid(np.dot(b,X_dsgn[i,:]))> threshold:
            pred=1
        else:
            pred=0
        total+=abs(pred-Y[i])
    return (1-total/len(Y))
```

[7]:

```python
df_t = pd.read_csv ('test_and_validation.csv').tail(12526) #test data, _t indicates test, _v indicates validation
df_t=df_t.drop(['Unnamed: 0'], axis=1)
X_dt=df_t.drop(['Fire Alarm'], axis=1).to_numpy() #_d means dataframe version of the matrix
Y_dt=df_t['Fire Alarm'].to_numpy()
X_std_t=normalize(X_dt) #standardized test data
X_prjct_t=np.dot(X_std_t,eigvec_X[:,:6]) #standardized then projected test data to PC components
X_dsgn_t=np.c_[ np.ones(len(X_prjct_t[:,1])),X_prjct_t]  #test design matrix with the first coulmn equaling to 1
```

[8]:

```python
df_v = pd.read_csv ('test_and_validation.csv').head(12526) #validation data , _t indicates test, _v indicates vali
df_v=df_v.drop(['Unnamed: 0'], axis=1)
X_dv=df_v.drop(['Fire Alarm'], axis=1).to_numpy() #_d means dataframe version of the matrix
Y_dv=df_v['Fire Alarm'].to_numpy()
X_std_v=normalize(X_dv)   #standardized validation data
X_prjct_v=np.dot(X_std_v,eigvec_X[:,:6]) #standardized then projected validation data to PC components
X_dsgn_v=np.c_[ np.ones(len(X_prjct_v[:,1])),X_prjct_v]  #validation design matrix with the first column equaling
```

[22]:

```python
b=[0,0,0,0,0,0,0]#starting at all beta 0
```

[ ]:

```python
%%time
#train_loss=[cross_entropy_loss_total(X_dsgn,Y,b)]
epoch=200
learing_rate=2
print(f'The coeficients are :{b} \t The cost is: {cross_entropy_loss_total(X_dsgn,Y,b)} ')
for t in range(epoch):
    b=gradient_descent(learing_rate,X_dsgn,Y,b)
    #train_loss.append(cross_entropy_loss_total(X_dsgn,Y,b))
    print(f'The coeficients are :{b} \t The cost is: {cross_entropy_loss_total(X_dsgn,Y,b)} ')
```

[24]:

```python
print(f'The accuarcy for the traing data is: {accuracy(Y,X_dsgn,0.5,b)}, and the accuarcy for the validation data
#0.5 is the threshold
```

```
The accuarcy for the traing data is: 0.8129756772579702, and the accuarcy for the validation data is: 0.81055404758103
14
```

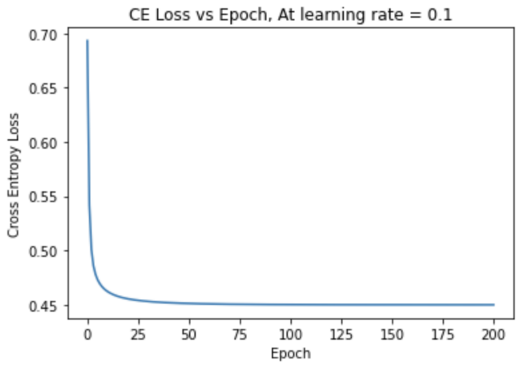
[12]:

```python
#Thus, we can see that there is no overfit so we can use the described model above. Final metrics are:
print(f'The accuarcy for the traing data is: {accuracy(Y,X_dsgn,0.5,b)}, and the accuarcy for the test data is: {a
```

```
The accuarcy for the traing data is: 0.8129756772579702, and the accuarcy for the test data is: 0.8107935494172123
```

```
[13]: plt.plot([x for x in range(len(train_loss))],train_loss)
      plt.xlabel('Epoch')
      plt.ylabel('Cross Entropy Loss')
      plt.title('CE Loss vs Epoch, At learning rate = 0.1')
      #plt.savefig('lr0.1.png')
```

```
[13]: Text(0.5, 1.0, 'CE Loss vs Epoch, At learning rate = 0.1')
```



```
[14]: b_original=[0,0,0,0,0,0,0,0,0,0,0,0,0,0]
```

```
[ ]: sion without PCA
     np.ones(len(X_std[:,1])),X_std] #Used standardized version because without it we can see extremely big numbers whi

     _descent(2,X_dsgn_original,Y,b_original)
     ents are :{b_original} \t The cost is: {cross_entropy_loss_total(X_dsgn_original,Y,b_original)} ')
```

```
[ ]: 
```

```
[39]: np.ones(len(X_std_v[:,1])),X_std_v],0.5,b_original)} test accuaracy is : {accuracy(Y_dt,np.c_[ np.ones(len(X_std_t
```

```
The accuarcy for the traing data is: 0.8974666027995104, and the accuarcy for the validation data is: 0.89493852786204
7 test accuaracy is : 0.8941401884081112
```

8.4 Neural Networks

[2611…

```python
#Neural Network
def RelU(x):
    return np.maximum(0, x)

def derRelU(z):
    return (z>0)

def initialize(neurons): #where neurons is a list where all of neuron count info is stored, layer by layer
    B1=[np.random.randn(i) for i in neurons[1:]][0]
    B2=[np.random.randn(i) for i in neurons[1:]][1]
    B3=[np.random.randn(i) for i in neurons[1:]][2]
    W1=[np.random.randn(j, i) for i,j  in zip(neurons[:-1], neurons[1:])][0]
    W2=[np.random.randn(j, i) for i,j  in zip(neurons[:-1], neurons[1:])][1]
    W3=[np.random.randn(j,i) for i,j  in zip(neurons[:-1], neurons[1:])][2]
    return [W1,W2,W3], [B1,B2,B3]

def feed_forward(x,W,B): #inputs a single example, outputs Activations ,  z and the output of the network
    z1=(np.matmul(W[0],np.array(x).T)+B[0])
    a1=RelU(z1)
    z2=(np.matmul(W[1], a1)+B[1])
    a2=RelU(z2)
    z3=(np.matmul(W[2], a2)+B[2])
    a3=sigmoid(z3)
    output=a3
    return [x,a1,a2],[z1,z2,z3], output

def back_prop(y,W,B,A,Z,output): #for a sinlge example
    #output delta
    delta_out=[np.array([output-y])]
    #hidden delta
    for w, b, a in zip( W[::-1],B[::-1],A[::-1]):
        delta_out.append(np.multiply(np.matmul(w.T,delta_out[-1]), derRelU(a)))
    delta_out=delta_out[::-1][1:]   #remove dC/dinput
    d_weights=[]
    for i ,a in zip(delta_out,A):
        d_weights.append(np.matmul(np.reshape(i,(len(i),1)), np.reshape(a,(1,len(a)) )))
    d_bias=delta_out
    return d_weights, d_bias

def update_network(dw,db,W,B,lr):
    for i in range(len(W)):
        W[i]=np.subtract(W[i],dw[i]*lr)
        B[i]=np.subtract(B[i],db[i]*lr)
    return W,B

def cross_entropy_neural(o,y): #single
    return  -(y*math.log(o)+ (1-y)*math.log(1.0001- o)) #causes domain error if it was 1

def loss_and_accuracy(X_t,Y_t,X_v,Y_v,W,B):
    total_loss_t=0
    total_loss_v=0
    accuracy_t=0
    accuracy_v=0
    for xx_t,yy_t in zip(X_t,Y_t):
            total_loss_t+= cross_entropy_neural(feed_forward(np.array(xx_t),W,B)[-1],yy_t)
            accuracy_t+= abs(yy_t- (feed_forward(np.array(xx_t),W,B)[-1]>0.5))
    for xx_v,yy_v in zip(X_v,Y_v):
        total_loss_v+= cross_entropy_neural(feed_forward(np.array(xx_v),W,B)[-1],yy_v)
        accuracy_v+= abs(yy_v- (feed_forward(np.array(xx_v),W,B)[-1]>0.5))
    #return f'train loss ={total_loss_t/len(Y_t)} val: {total_loss_v/len(Y_v)} accuracy: {1-accuracy_t/len(Y_t)} val:{1-
    return total_loss_t/len(Y_t), total_loss_v/len(Y_v), 1-accuracy_t/len(Y_t), 1-accuracy_v/len(Y_v)
def SGD(X_t,Y_t,X_v,Y_v,W,B,epoch,lr):
    loss_t=[]
    loss_v=[]
    for x_t,y_t in zip(X_t[:epoch],Y_t[:epoch]):
        total_loss_t=0
        total_loss_v=0
        accuracy_t=0
        accuracy_v=0
        A,Z,output= feed_forward(np.array(x_t),W,B)

        dw,db = back_prop(y_t,W,B,A,Z,output)
        W,B=update_network(dw,db,W,B,lr)
    return W,B,loss_t, loss_v

def MBSGD(X_t,Y_t,X_v,Y_v,W,B,epoch,lr,batchsize):
    loss_t=[]
    loss_v=[]
    for i in range(epoch):
        c=list(zip(X_t,Y_t))
        random.shuffle(c)
        c=c[:batchsize]
        x_one,y_one = zip(*c)
        total_w=0
        total_b=0
        count=0
        for x_t,y_t in zip(x_one,y_one): #one batch
            A,Z,output= feed_forward(np.array(x_t),W,B)
            dw,db = back_prop(y_t,W,B,A,Z,output)
            if count==0:
                total_w=dw
                total_b=db
                count+=1
            else:
                for i in range(len(dw)): #inside W
                    total_w[i]=np.add(total_w[i],dw[i])
                    total_b[i]=np.add(total_b[i],db[i])
        M=loss_and_accuracy(X_t,Y_t,X_v,Y_v,W,B)
        loss_t.append(M[0])
        loss_v.append(M[1])
        print(f'train loss= {M[0]} val loss: {M[1]} train accuracy{M[2]} val accuracy:{M[3]}')
        W,B=update_network(total_w,total_b,W,B,lr/batchsize)
    return W,B,loss_t, loss_v
```

[2640…
```
W,B=initialize([6,4,4,1])
loss_and_accuracy(X_prjct,Y,X_prjct_v,Y_dv,W,B)
```

[ ]:
```
%%time
W_new,B_new,loss_t,loss_v = MBSGD(X_prjct,Y,X_prjct_v,Y_dv,W,B,500,0.1,100)
```

A,Z,output= feed_forward(np.array([1,0.4,-1,-1,1,2]),W,B) dw, db= back_prop(0,W,B,A,Z,output) W,B=update_network(dw,db,W,B,0.01)

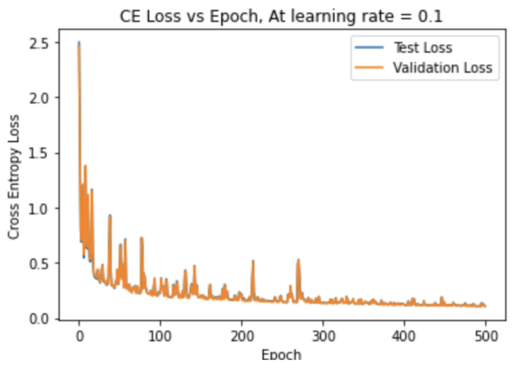
W,B=update_network(dw,db,W,B,0.01)

[ ]:
```
%%time
W_new,B_new,loss_t,loss_v=SGD(X_prjct,Y,X_prjct_v,Y_dv,W,B,15000,0.001)

```

[ ]:
```
loss_and_accuracy(X_prjct,Y,X_prjct_v,Y_dv,W,B)
```

[2603…
```
plt.plot([i for i in range(len(loss_t))],loss_t,label="Test Loss")
plt.plot([i for i in range(len(loss_v))],loss_v,label="Validation Loss")
plt.legend(['Test Loss','Validation Loss'])
plt.xlabel('Epoch')
plt.ylabel('Cross Entropy Loss')
plt.title('CE Loss vs Epoch, At learning rate = 0.1')
plt.savefig('MBSGD0.1.png')
```



References

[1] D. Contractor, “Smoke detection dataset,” Aug 2022. [Online]. Available: https://www.kaggle.com/datasets/deepcontractor/smoke-detection-dataset

[2] S. Blattmann, “Real-time smoke detection with ai-based sensor fusion,” Aug 2022. [Online]. Available: https://www.hackster.io/stefanblattmann/real-time-smoke-detection-with-ai-based-sensor-fusion-1086e6

[3] M. Vafakhah and S. Janizadeh, “Chapter 6 - application of artificial neural network and adaptive neuro-fuzzy inference system in streamflow forecasting,” in *Advances in Streamflow Forecasting*, P. Sharma and D. Machiwal, Eds. Elsevier, 2021, pp. 171–191. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780128206737000020

[4] N. Sairamya, L. Susmitha, S. Thomas George, and M. Subathra, “Chapter 12 - hybrid approach for classification of electroencephalographic signals using time–frequency images with wavelets and texture features,” in *Intelligent Data Analysis for Biomedical Applications*, ser. Intelligent Data-Centric Systems, D. J. Hemanth, D. Gupta, and V. Emilia Balas, Eds. Academic Press, 2019, pp. 253–273. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780128155530000136

[5] N. Naveen, “What is gradient descent, batch gradient descent, stochastic gradient descent, mini-batch gradient descent?” Jan 2023. [Online]. Available: https://www.nomidl.com/machine-learning/what-is-gradient-descent-batch-gradient-descent-stochastic-gradient-descent-mini-batch-gradient-descent/