In [ ]:
```
#import libraries
import numpy as np
import pandas as pd
import keras

from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn import metrics
from sklearn.metrics import accuracy_score, f1_score, confusion_matrix, precision_score, recall_score

from keras.models import Sequential
from keras.layers import Dense, Activation
from keras.optimizers import Adam
```

In [ ]:
```
import seaborn as sns
import matplotlib.pyplot as plt
```

In [ ]:
```
#read the train data
train_data=pd.read_csv(r'C:\Users\96891\OneDrive\Documents\sonia\smoke_detection_iot.csv')
```

In [ ]:
```
#print the data
train_data.head()
```

In [ ]:
```
#dropping unnecessary columns
train_data.drop(['Unnamed: 0', 'UTC', 'CNT'], axis=1, inplace=True)
```

In [ ]:
```
#check updated data
train_data.head()
```

In [ ]:
```
#data pre-proccessing
#splitting the dependent and independent variable
x=train_data.drop('Fire Alarm', axis=1)
y=train_data['Fire Alarm']
```

In [ ]:
```
#splitting the data into training and testing sets
X_train, X_test, Y_train, Y_test= train_test_split(x,y,test_size=0.3,random_state=0)
#random_state=0, we get the same train and test sets accross different executions
```

In [ ]:
```
#print the dimensions of the train and test data
print(X_train.shape)
print(Y_train.shape)
print(X_test.shape)
print(Y_test.shape)
```

In [ ]:
```
# the scale of each feature is very different, so we need to bring all of them to the same scale.
ss= StandardScaler()
X_train=ss.fit_transform(X_train)
X_test= ss.transform(X_test)
```

In [ ]:
```
#class distribution
#check if the target classes are balanced
sns.countplot(x = Y_train)
plt.text(x = 0 - 0.1, y = Y_train.value_counts()[0] + 500, s = Y_train.value_counts()[0])
plt.text(x = 1 - 0.1, y = Y_train.value_counts()[1] + 500, s = Y_train.value_counts()[1])
plt.xticks([0, 1], ['No Alarm', 'Alarm'])
plt.ylabel('Count')
plt.tight_layout(pad = -1)
plt.title('Class Imbalance', fontsize = 15)
plt.show()
```

In [ ]:
```
pip install imblearn
```

In [ ]:
```
#data is highly biased, will result in a biased model
#solution:Synthetic Minority Over-sampling Technique
from imblearn.over_sampling import SMOTE

smote = SMOTE(random_state = 10)
X_train, Y_train = smote.fit_resample(X_train, Y_train)
```

In [ ]:
```
#check classes again
sns.countplot(x = Y_train)
plt.text(x = 0 - 0.1, y = Y_train.value_counts()[0] + 500, s = Y_train.value_counts()[0])
plt.text(x = 1 - 0.1, y = Y_train.value_counts()[1] + 500, s = Y_train.value_counts()[1])
plt.xticks([0, 1], ['No Alarm', 'Alarm'])
plt.ylabel('Count')
plt.tight_layout(pad = -1)
plt.title('Class Imbalance', fontsize = 15)
plt.show()
```

In [ ]:
```
#now that the data is balanced, we can build the model
#Dense Neural Network
#Model Architecture
model=Sequential([
    Dense(units=32, activation='relu',input_shape=(12,),name="Layer1"),
    Dense(units=64,activation='relu',name="Layer2"),
    Dense(units=128, activation='relu', name="Layer3"),
    Dense(units=1, activation='sigmoid', name="Output")
])
#relu activation function is used in the hidden layers and sigmoid activation function is used in the output layer
```

In [ ]:
```
#before training, we must compile the model
model.compile(loss='binary_crossentropy', optimizer= 'Adam', metrics=['accuracy'])
```

In [ ]:
```
#fit the model
model.fit(X_train, Y_train, validation_split=0.1, batch_size=10, epochs=10, shuffle=True, verbose=2)
```

In [ ]:
```
#evaluate the model on testing data
model.evaluate(X_test,Y_test)
```

In [ ]:
```
Y_true,Y_pred=Y_test, np.round(model.predict(X_test))
```

In [ ]:
```
#calculate
f1=f1_score(Y_true, Y_pred)
acc=accuracy_score(Y_true, Y_pred)
precision=precision_score(Y_true, Y_pred)
recall=recall_score(Y_true, Y_pred)
cm=confusion_matrix(Y_true, Y_pred)
```

In [ ]:
```
#print
print(f"F1 Score : {f1}\n")
print(f"Accuracy : {acc}\n")
print(f"Precision : {precision}\n")
print(f"Recall : {recall}\n")
print(f"Confusion Matrix : {cm}\n")
```

In [ ]:
```
TN=cm[0][0]
FN=cm[1][0]
FP=cm[0][1]
TP=cm[1][1]
```

In [ ]:
```
print ("True Positive= ", TP)
print ("True Negative= ", TN)
print ("False Positive= ", FP)
print ("False Negative= ", FN)
```

In [ ]:
```
#specificity
print ("Specifity=", TN/(TN+FP))
```

In [ ]:
```
#sensitivity
print ("Sensitivity=", TP/(TP+FN))
```

In [ ]:
```
#print classification report
from sklearn.metrics import classification_report
print (classification_report (Y_true, Y_pred))
```

In [ ]: