Supervised Machine Learning: Classification

IBM Skills Network

Project Report

By:

Fahd Seddik

Table of Contents

Main Objective

A smoke detector is a device that senses smoke, typically as an indicator of fire. Smoke detectors are usually housed in plastic enclosures, typically shaped like a disk about 150 millimetres (6 in) in diameter and 25 millimetres (1 in) thick, but shape and size vary.

In this data set (shown next section), we are given some sensor data and based on which we want to try and predict if there is fire or not.

Brief Description

The data set has many useful information that would help us identify and classify our target. Shown below is a snippet of data.head() which shows us a brief about the values of our columns. The data set has **62,630 examples**, the target column is **'Fire Alarm'** being either **0** or **1** and features include:

- UTC
- Temperature[C]
- 'Humidity[%]
- TVOC[ppb]
- eCO2[ppm]
- Raw H2
- Raw Ethanol
- Pressure[hPa]
- PM1.0
- PM2.5
- NC0.5
- NC1.0
- NC2.5
- CNT'

```
data.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 62630 entries, 0 to 62629
Data columns (total 16 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   Unnamed: 0      62630 non-null  int64
 1   UTC             62630 non-null  int64
 2   Temperature[C]  62630 non-null  float64
 3   Humidity[%]     62630 non-null  float64
 4   TVOC[ppb]       62630 non-null  int64
 5   eCO2[ppm]       62630 non-null  int64
 6   Raw H2          62630 non-null  int64
 7   Raw Ethanol     62630 non-null  int64
 8   Pressure[hPa]   62630 non-null  float64
 9   PM1.0           62630 non-null  float64
 10  PM2.5           62630 non-null  float64
 11  NC0.5           62630 non-null  float64
 12  NC1.0           62630 non-null  float64
 13  NC2.5           62630 non-null  float64
 14  CNT             62630 non-null  int64
 15  Fire Alarm      62630 non-null  int64
dtypes: float64(8), int64(8)
memory usage: 7.6 MB
```

```
data.shape

(62630, 16)
```

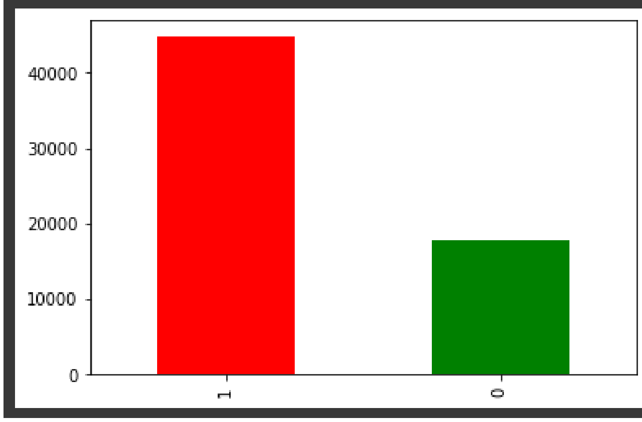
```
data.head()
```

	Unnamed: 0	UTC	Temperature[C]	Humidity[%]	TVOC[ppb]	eCO2[ppm]	Raw H2	Raw Ethanol	Pressure[hPa]	PM1.0	PM2.5	NC0.5	NC1.0	NC2.5	CNT	Fire Alarm
0	0	1654733331	20.000	57.36	0	400	12306	18520	939.735	0.0	0.0	0.0	0.0	0.0	0	0
1	1	1654733332	20.015	56.67	0	400	12345	18651	939.744	0.0	0.0	0.0	0.0	0.0	1	0
2	2	1654733333	20.029	55.96	0	400	12374	18764	939.738	0.0	0.0	0.0	0.0	0.0	2	0
3	3	1654733334	20.044	55.28	0	400	12390	18849	939.736	0.0	0.0	0.0	0.0	0.0	3	0
4	4	1654733335	20.059	54.69	0	400	12403	18921	939.744	0.0	0.0	0.0	0.0	0.0	4	0

Data cleaning & Feature engineering

We might first want to examine if there exists any imbalance in our target column. We will do that by using a bar plot for our ‘Fire Alarm’ target column. As shown below, we can clearly see that there is in fact imbalance in our target column which might hint about using either over-sampling or under-sampling techniques.

```
y_col='Fire Alarm'
data[y_col].value_counts().plot.bar(color=['red','green']);
```



We can check for both missing values or duplicates and handle them right away as shown below by just using the isnull(). It is clear that we did not find any missing values. We can also see that there are no duplicates in our data set which indicates a good quality data set. When it comes to feature engineering, we might use it in our models later in the next section.

```
data.isnull().sum().sort_values()
```

```
Unnamed: 0        0
UTC               0
Temperature[C]    0
Humidity[%]       0
TVOC[ppb]         0
eCO2[ppm]         0
Raw H2            0
Raw Ethanol       0
Pressure[hPa]     0
PM1.0             0
PM2.5             0
NC0.5             0
NC1.0             0
NC2.5             0
CNT               0
Fire Alarm        0
dtype: int64
```

```
data.nunique()
```

```
Unnamed: 0        62630
UTC               62630
Temperature[C]    21672
Humidity[%]        3890
TVOC[ppb]          1966
eCO2[ppm]          1713
Raw H2             1830
Raw Ethanol        2659
Pressure[hPa]      2213
PM1.0              1337
PM2.5              1351
NC0.5              3093
NC1.0              4113
NC2.5              1161
CNT               24994
Fire Alarm            2
dtype: int64
```

We also will use **StratifiedShuffleSplit** to ensure good distribution between our train and test sets respectively. We managed to split into 71.4% **1**s and 28.5% **0** for both our train and test target variables.

```
strat_shuf_split = StratifiedShuffleSplit(n_splits=1,test_size=0.3,random_state=42)
train_idx,test_idx = next(strat_shuf_split.split(data.drop(y_col,axis=1),data[y_col]))
X_train = data.drop(y_col,axis=1).iloc[train_idx,:]
y_train = data[y_col].iloc[train_idx]
X_test= data.drop(y_col,axis=1).iloc[test_idx,:]
y_test = data[y_col].iloc[test_idx]
```

```
X_train.shape,y_train.shape,X_test.shape,y_test.shape
```

```
((43841, 15), (43841,), (18789, 15), (18789,))
```

```
y_train.value_counts(normalize=True)
```

```
1    0.714628
0    0.285372
Name: Fire Alarm, dtype: float64
```

```
y_test.value_counts(normalize=True)
```

```
1    0.71462
0    0.28538
Name: Fire Alarm, dtype: float64
```

Classifier Models

We will start with **LogisticRegression** as our base-line model so we can compare the rest to base-line. We will use class weighting too. Our Logistic Regression model did good enough with statistics shown below.

```
class_weight = {}

# Assign weight of class 0 to be 0.29
class_weight[0] = 0.29
# Assign weight of class 1 to be 0.71
class_weight[1] = 0.71
model_lr = LogisticRegression(random_state=123,
                              max_iter = 1000,
                              class_weight=class_weight)
```

```
print(classification_report(preds_lr, y_test))
```

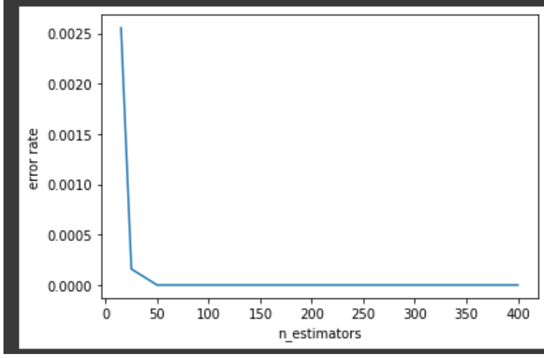
```
              precision    recall  f1-score   support

           0       0.00      0.00      0.00         0
           1       1.00      0.71      0.83     18789

    accuracy                           0.71     18789
   macro avg       0.50      0.36      0.42     18789
weighted avg       1.00      0.71      0.83     18789
```

We will now move on to more complex models. For our second model we will use **GradientBoostingClassifier**. We tried fitting our model to several number of estimators to find the best out of all of them. Results hinted towards n_estimators=50 being the best number of trees for the GradientBoostingClassifier.

```
plt.plot(error_df);
plt.xlabel('n_estimators');
plt.ylabel('error rate');
```




```
print(classification_report(preds_gbc, y_test))

              precision    recall  f1-score   support

           0       1.00      1.00      1.00      5362
           1       1.00      1.00      1.00     13427

    accuracy                           1.00     18789
   macro avg       1.00      1.00      1.00     18789
weighted avg       1.00      1.00      1.00     18789
```

We can see from the error plot above that it seems to plateau after 50 estimators so we will be using that. Our model produced outstanding results. For last model we will be using a **StackingClassifier** this would incroporate SVC, KNN, and DecisionTreeClassifier and will use a final_estimator of a LogisticRegression model.

```
estimators = [('SVM',SVC(random_state=42)),('KNN',KNeighborsClassifier()),('dt',DecisionTreeClassifier())]
clf = StackingClassifier( estimators=estimators, final_estimator= LogisticRegression())
clf.fit(X_train, y_train)

StackingClassifier(estimators=[('SVM', SVC(random_state=42)),
                               ('KNN', KNeighborsClassifier()),
                               ('dt', DecisionTreeClassifier())],
                   final_estimator=LogisticRegression())

preds_clf = clf.predict(X_test)
```

```
print(classification_report(preds_clf, y_test))

              precision    recall  f1-score   support

           0       1.00      1.00      1.00      5361
           1       1.00      1.00      1.00     13428

    accuracy                           1.00     18789
   macro avg       1.00      1.00      1.00     18789
weighted avg       1.00      1.00      1.00     18789
```

Recommended Model

Based on the statistics shown above we can clearly state that the recommended model is **GradientBoostingClassifier** and this is due to the extraordinary performance demonstrated in the previous section. The model has performed well on all scores with number of estimators of only 50. We choose that model because it is not as complex and still does the needed job and has good scores.

Key Findings and Insights

We can see that our StackingClassifier did perform well. However, it is prone to over-fitting as the more complex a model gets the more it is likely to over-fit. Thus, even though it performed good on the test set but it would be preferred to choose the less complex model for better interpretability (**next section for improvements**). When it comes to LogisticRegression the model's performance was moderate. GradientBoostingClassifier obviously performed really well on our train set.

Suggestions

It is certain that further analysis and more models could be applied to this data set and maybe we can have better results. However, some of the suggestions I see for our next steps include having a **GridSearchCV** to try to eliminate over-fitting while further enhancing our models to choose the best hyperparameters for each of them. We can also use these models and save them using the **pickle** library for later use in more sophisticated models or act as a "teacher model" for data distillation.