KNN - MAGIC Gamma Telescope

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, precision_score,
recall_score, f1_score, confusion_matrix
from sklearn.neighbors import KNeighborsClassifier
from imblearn.under_sampling import RandomUnderSampler
from sklearn.preprocessing import MinMaxScaler
import matplotlib.pyplot as plt
```

The next cell the the dataset column names and their adjustment and length

```
column_names = [
    "fLength", "fWidth", "fSize", "fConc", "fConc1",
    "fAsym", "fM3Long", "fM3Trans", "fAlpha", "fDist", "class"
]

dataset = pd.read_csv(r'D:\term 7\machine learning\ML- Assignment 1\
ML- Assignment 1\magic04.data', header=None, names=column_names)
print(len(dataset))
dataset.head()

19020

    fLength    fWidth   fSize   fConc  fConc1     fAsym  fM3Long  
fM3Trans  \
0   28.7967   16.0021  2.6449  0.3918  0.1982   27.7004  22.0110   -
8.2027
1   31.6036   11.7235  2.5185  0.5303  0.3773   26.2722  23.8238   -
9.9574
2  162.0520  136.0310  4.0612  0.0374  0.0187  116.7410 -64.8580  -
45.2160
3   23.8172    9.5728  2.3385  0.6147  0.3922   27.2107  -6.4633   -
7.1513
4   75.1362   30.9205  3.1611  0.3168  0.1832   -5.5277  28.5525   
21.8393

    fAlpha     fDist class
0  40.0920   81.8828     g
1   6.3609  205.2610     g
2  76.9600  256.7880     g
3  10.4490  116.7370     g
4   4.6480  356.4620     g
```

[6]Here we are using pandas library to read our data from the magic04.data file [1] We defined column names and I added for the column that classify whether g or h name class to classify upon this column

Separate dataset into features and labels

```
x=dataset.drop(['class'],axis=1)
y=dataset['class']
```

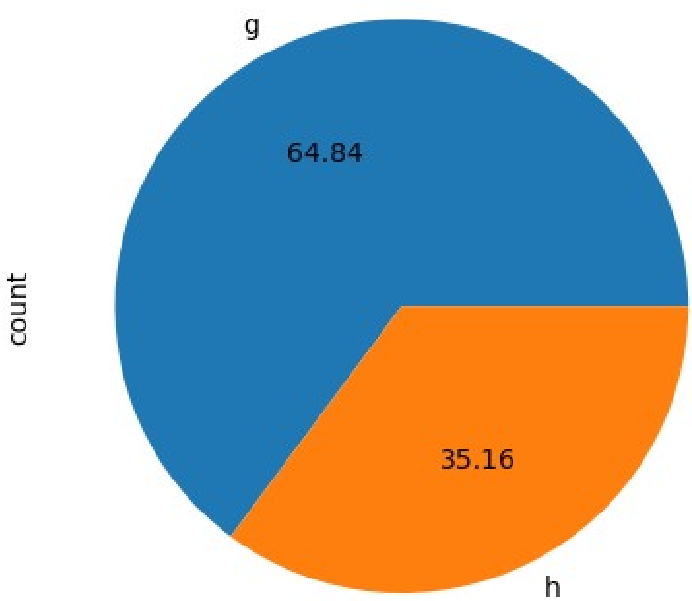
see data distribution

```
y.value_counts()
```

```
class
g    12332
h     6688
Name: count, dtype: int64
```

```
y.value_counts().plot.pie(autopct='%.2f')
```

```
<Axes: ylabel='count'>
```



The data diagram is unbalaced and we have to start balancing the dataset with equal samples for each class so no biasing happens

```
#rus=RandomUnderSampler(sampling_strategy=1)
#x_res,y_res=rus.fit_resample(x,y)

gamma_data = dataset[dataset['class'] == 'g']
hadron_data = dataset[dataset['class'] == 'h']
gamma_sampled = gamma_data.sample(n=len(hadron_data), random_state=42)
```

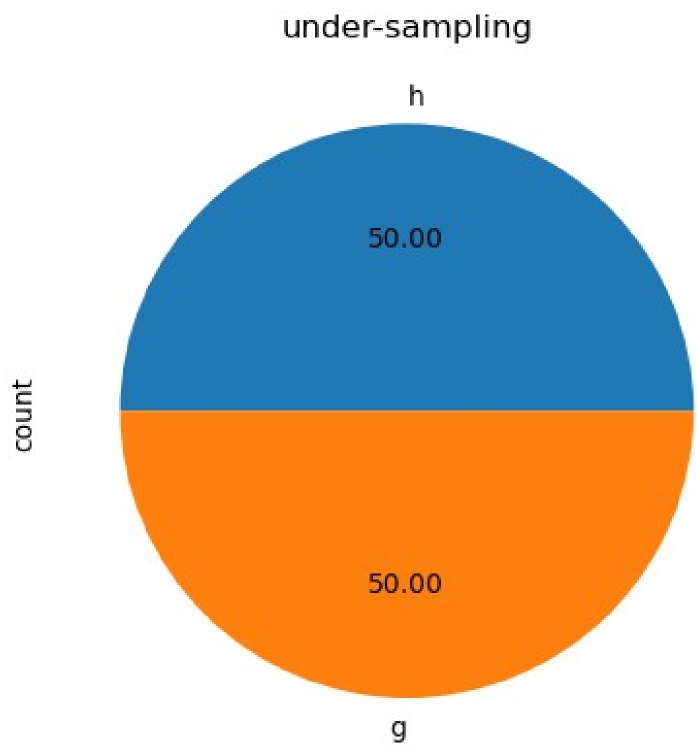
```
balanced_data = pd.concat([gamma_sampled, hadron_data])
balanced_data = balanced_data.sample(frac=1,
random_state=42).reset_index(drop=True)
x_res=balanced_data.drop(['class'],axis=1)
y_res=balanced_data['class']

# Display the balanced class counts to verify
print(balanced_data['class'].value_counts())
```

```
class
h    6688
g    6688
Name: count, dtype: int64
```

[4] & [5] Separate gamma and hadron classes based on the column called class It is the column where it specifies whether g or h [6]Here take from the gamma class part of it which has the same size of the hadron class To make sure that our data is balanced and wouldn't cause bias Random state insures that you are taking your random data from specific point To make sure that everytime we run the code we select our data from the same point [7] We are merging the 2 classes together the balanced gamma class and the hadron class see the data distribution after balancing

```
ax=balanced_data['class'].value_counts().plot.pie(autopct='%.2f')
_=ax.set_title("under-sampling")
balanced_data['class'].value_counts()
```

```
class
h    6688
g    6688
Name: count, dtype: int64
```



Split data and see the dataset numbers The data is now balanced

```
# First, split off 70% for training, leaving 30% in the temporary set
(for validation and testing)
x_train, x_temp, y_train, y_temp = train_test_split(x_res, y_res,
test_size=0.3, random_state=42, stratify=y_res)

# Second, split the temporary set into 50% validation and 50% test
sets
x_val, x_test, y_val, y_test = train_test_split(x_temp, y_temp,
test_size=0.5, random_state=42, stratify=y_temp)

print("Training set size:", x_train.shape[0])
print("Validation set size:", x_val.shape[0])
print("Testing set size:", x_test.shape[0])

Training set size: 9363
Validation set size: 2006
Testing set size: 2007
```

This train_test_split data 70% train data and 30% temp data that will be split This split temp data to two equal parts part for val and other for testing Normalize dataset and view the train dataset

```
scaler = MinMaxScaler()
x_train = scaler.fit_transform(x_train)
```

```
x_test = scaler.transform(x_test)
x_val = scaler.transform(x_val)

x_train_normalized = pd.DataFrame(x_train, columns=x.columns)
x_test_normalized = pd.DataFrame(x_test, columns=x.columns)
x_train_normalized.head()
```

```
    fLength    fWidth     fSize     fConc    fConc1     fAsym
fM3Long  \
0  0.064315  0.094802  0.248906  0.389249  0.411545  0.449520
0.599677
1  0.058032  0.067472  0.220934  0.353108  0.268866  0.393148
0.627575
2  0.039526  0.042897  0.108900  0.648028  0.529641  0.444344
0.561134
3  0.208079  0.113263  0.435186  0.347653  0.261242  0.411335
0.646232
4  0.353522  0.424884  0.611354  0.137857  0.096157  0.457531
0.496167

   fM3Trans    fAlpha     fDist
0  0.503828  0.319064  0.258249
1  0.591757  0.208913  0.542122
2  0.564002  0.354243  0.529708
3  0.517860  0.043420  0.756806
4  0.308644  0.985414  0.723230
```

show the test dataset after normalizing After splitting the data we took each data group and started to normalize it so that all our values are between 0 and 1 to avoid outliers

```
x_test_normalized.head()
```

```
    fLength    fWidth     fSize     fConc    fConc1     fAsym
fM3Long  \
0  0.067323  0.042402  0.135275  0.588476  0.536798  0.450788
0.602953
1  0.052257  0.046628  0.163690  0.599727  0.489964  0.468789
0.587452
2  0.214614  0.067135  0.349172  0.260484  0.203205  0.442485
0.660487
3  0.077003  0.025549  0.074453  0.582453  0.423370  0.406165
0.549749
4  0.362937  0.217648  0.732200  0.047505  0.043411  0.502036
0.748352

   fM3Trans    fAlpha     fDist
0  0.520040  0.491377  0.279852
1  0.561223  0.052498  0.475856
2  0.577174  0.392265  0.572997
```

```
3  0.542968  0.410744  0.289972
4  0.651040  0.086825  0.555269
```

show the validate dataset after normalizing

```
x_val_normalized = pd.DataFrame(x_val, columns=x.columns)
x_val_normalized.head()

    fLength    fWidth     fSize     fConc    fConc1     fAsym
fM3Long  \
0  0.066722  0.040568  0.195683  0.622798  0.505835  0.472812
0.602608
1  0.128692  0.065193  0.325517  0.300716  0.241637  0.440124
0.645570
2  0.065363  0.043380  0.222472  0.560291  0.479851  0.446413
0.605631
3  0.062601  0.066149  0.188971  0.442096  0.328769  0.447561
0.544885
4  0.308699  0.111024  0.433678  0.155700  0.138789  0.323270
0.718523

   fM3Trans    fAlpha     fDist
0  0.522099  0.246854  0.521813
1  0.571047  0.048879  0.442226
2  0.565169  0.040109  0.396978
3  0.523452  0.175032  0.295328
4  0.596289  0.056205  0.510953
```

Define the model : Init K-NN Here we return the data back in the form of data frame

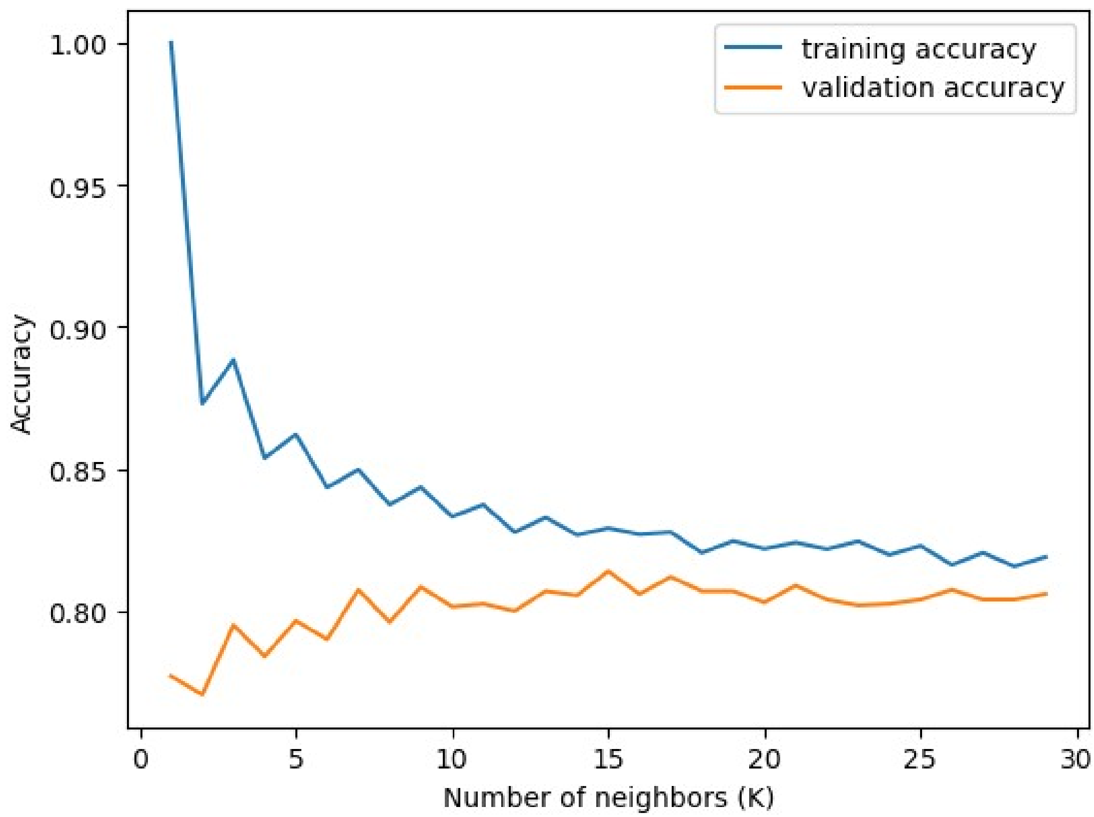
```
k_values = range(1,30)  # Trying a wider range of k values from 1 to
30
training_accuracy=[]
validation_accuracy = []

for k in k_values:
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(x_train_normalized, y_train)
    training_accuracy.append(knn.score(x_train_normalized, y_train))
    validation_accuracy.append(knn.score(x_val_normalized, y_val))
```

we will start trying different k values [7]we train the model using KNN drop to drop the target column then fit data I give it the column containing the correct answers and the data The function knn.score calculates the portion of the correct answers

```
plt.plot(k_values,training_accuracy,label="training accuracy")
plt.plot(k_values,validation_accuracy,label="validation accuracy")
plt.ylabel("Accuracy")
```

```
plt.xlabel("Number of neighbors (K)")
plt.legend()
```

```
<matplotlib.legend.Legend at 0x1bba5ef9a00>
```




```
best_k_index = validation_accuracy.index(max(validation_accuracy))
best_k = k_values[best_k_index]

final_knn = KNeighborsClassifier(n_neighbors=best_k)
final_knn.fit(x_train_normalized, y_train)
y_test_pred = final_knn.predict(x_test_normalized)

test_accuracy1 = final_knn.score(x_test_normalized, y_test)
test_accuracy2 = accuracy_score(y_test, y_test_pred)
test_precision = precision_score(y_test, y_test_pred, pos_label='g',
zero_division=1)
test_recall = recall_score(y_test, y_test_pred, pos_label='g',
zero_division=1)
test_f1 = f1_score(y_test, y_test_pred, pos_label='g',
zero_division=1)
test_confusion_matrix = confusion_matrix(y_test, y_test_pred)

print("Evaluation on Test Set:")
print(f"Best k value is: {best_k}")
```

```python
print(f"Test Accuracy with best k: {test_accuracy1 * 100:.2f}%")
print(f"Test Accuracy with best k: {test_accuracy2 * 100:.2f}%")
print(f"Precision: {test_precision * 100:.2f}%")
print(f"Recall: {test_recall * 100:.2f}%")
print(f"F1 Score: {test_f1 * 100:.2f}%")
print("Confusion Matrix:")
print(test_confusion_matrix)
```

```
Evaluation on Test Set:
Best k value is: 15
Test Accuracy with best k: 81.42%
Test Accuracy with best k: 81.42%
Precision: 77.99%
Recall: 87.55%
F1 Score: 82.50%
Confusion Matrix:
[[879 125]
 [248 755]]
```

Here we calculated the best k that got the best accuracy in the validation data we used this k for our knn model we entered the test data and calculated accuracy, precision, recall, f1 and the cofusion matrix The function accuracy_score calculates the portion of the correct answers The function precision_score calculates the ratio of the correct answers to the total predicted answers The function recall_score calculates the ratio of the results saying it is glass to the actual g glass The function f1_score balance correctly identified samples and minimizing false outputs The confusion matrix is Matrix containing true g positives, true h negatives, false g positives, false h negatives