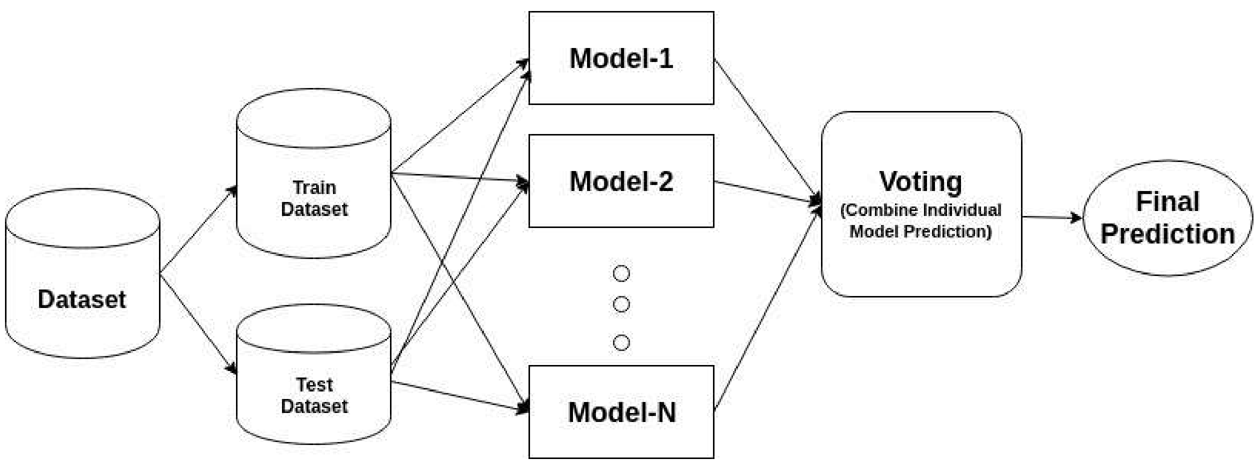
AdaBoost Classifier

1. Intro to Ensemble Machine Learning

- An ensemble model is a composite model which combines a series of low performing or weak classifiers with the aim of creating a strong classifier.
- Here, individual classifiers vote and final prediction label returned that performs majority voting.
- Now, these individual classifiers are combined according to some specific criterion to create an ensemble model.
- These ensemble models offer greater accuracy than individual or base classifiers.
- These models can parallelize by allocating each base learner to different mechanisms.
- So, we can say that ensemble learning methods are meta-algorithms that combine several machine learning algorithms into a single predictive model to increase performance.
- Ensemble models are created according to some specific criterion as stated below:-
  - **Bagging -** They can be created to decrease model variance using bagging approach.
  - **Boosting -** They can be created to decrease model bias using a boosting approach.
  - **Stacking -** They can be created to improve model predictions using stacking approach.
- It can be depicted with the help of following diagram.



1.1 Bagging

- **Bagging** stands for **bootstrap aggregation**.
- It combines multiple learners in a way to reduce the variance of estimates.
- For example, random forest trains N Decision Trees where we will train N different trees on different random subsets of the data and perform voting for final prediction.
- **Bagging ensembles** methods are **Random Forest** and **Extra Trees**.

1.2 Boosting

- **Boosting** algorithms are a set of the weak classifiers to create a strong classifier.
- Strong classifiers offer error rate close to 0.
- Boosting algorithm can track the model who failed the accurate prediction.
- Boosting algorithms are less affected by the overfitting problem.

- The following three algorithms have gained massive popularity in data science competitions.
  - AdaBoost (Adaptive Boosting)
  - Gradient Tree Boosting (GBM)
  - XGBoost
- We will discuss AdaBoost in this kernel and GBM and XGBoost in future kernels.

1.3 Stacking

- **Stacking** (or stacked generalization) is an ensemble learning technique that combines multiple base classification models predictions into a new data set.
- This new data are treated as the input data for another classifier.
- This classifier employed to solve this problem. Stacking is often referred to as blending.

2. How are base-learners classified

- Base-learners are classified into two types.
- On the basis of the arrangement of base learners, ensemble methods can be divided into two groups.
  - In parallel ensemble methods, base learners are generated in parallel for example - Random Forest.
  - In sequential ensemble methods, base learners are generated sequentially for example AdaBoost.
- On the basis of the type of base learners, ensemble methods can be divided into two groups.
  - homogenous ensemble method uses the same type of base learner in each iteration.
  - heterogeneous ensemble method uses the different type of base learner in each iteration.

3. AdaBoost Classifier

- **AdaBoost or Adaptive Boosting** is one of the ensemble boosting classifier proposed by Yoav Freund and Robert Schapire in 1996.
- It combines multiple weak classifiers to increase the accuracy of classifiers.
- AdaBoost is an iterative ensemble method. AdaBoost classifier builds a strong classifier by combining multiple poorly performing classifiers so that you will get high accuracy strong classifier.
- The basic concept behind Adaboost is to set the weights of classifiers and training the data sample in each iteration such that it ensures the accurate predictions of unusual observations.
- Any machine learning algorithm can be used as base classifier if it accepts weights on the training set.
- **AdaBoost** should meet two conditions:

  1. The classifier should be trained interactively on various weighed training examples.
  2. In each iteration, it tries to provide an excellent fit for these examples by minimizing training error.

- To build a AdaBoost classifier, imagine that as a first base classifier we train a Decision Tree algorithm to make predictions on our training data.
- Now, following the methodology of AdaBoost, the weight of the misclassified training instances is increased.
- The second classifier is trained and acknowledges the updated weights and it repeats the procedure over and over again.
- At the end of every model prediction we end up boosting the weights of the misclassified instances so that the next model does a better job on them, and so on.

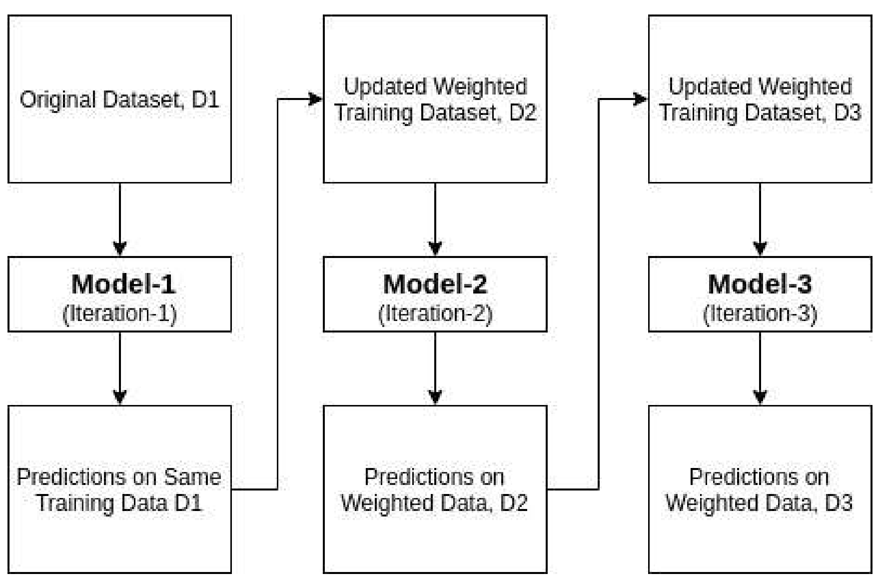
- AdaBoost adds predictors to the ensemble gradually making it better. The great disadvantage of this algorithm is that the model cannot be parallelized since each predictor can only be trained after the previous one has been trained and evaluated.
- Below are the steps for performing the AdaBoost algorithm:

  1. Initially, all observations are given equal weights.
  2. A model is built on a subset of data.
  3. Using this model, predictions are made on the whole dataset.
  4. Errors are calculated by comparing the predictions and actual values.
  5. While creating the next model, higher weights are given to the data points which were predicted incorrectly.
  6. Weights can be determined using the error value. For instance,the higher the error the more is the weight assigned to the observation.
  7. This process is repeated until the error function does not change, or the maximum limit of the

4. AdaBoost algorithm intuition

- It works in the following steps:

  1. Initially, Adaboost selects a training subset randomly.
  2. It iteratively trains the AdaBoost machine learning model by selecting the training set based on the accurate prediction of the last training.
  3. It assigns the higher weight to wrong classified observations so that in the next iteration these observations will get the high probability for classification.
  4. Also, It assigns the weight to the trained classifier in each iteration according to the accuracy of the classifier. The more accurate classifier will get high weight.
  5. This process iterate until the complete training data fits without any error or until reached to the specified maximum number of estimators.
  6. To classify, perform a "vote" across all of the learning algorithms you built.
- The intuition can be depicted with the following diagram:



5. Difference between AdaBoost and Gradient Boosting

- **AdaBoost** stands for **Adaptive Boosting**. It works on sequential ensemble machine learning technique. The general idea of boosting algorithms is to try predictors sequentially, where each subsequent model attempts to fix the errors of its predecessor.
- **GBM or Gradient Boosting** also works on sequential model. Gradient boosting calculates the gradient (derivative) of the Loss Function with respect to the prediction (instead of the features). Gradient boosting increases the accuracy by minimizing the Loss Function (error which is difference of actual and predicted value) and having this loss as target for the next iteration.
- Gradient boosting algorithm builds first weak learner and calculates the Loss Function. It then builds a second learner to predict the loss after the first step. The step continues for third learner and then for fourth learner and so on until a certain threshold is reached.
- So, the question arises in mind that how AdaBoost is different than Gradient Boosting algorithm since both of them works on Boosting technique.
- Both AdaBoost and Gradient Boosting build weak learners in a sequential fashion. Originally, AdaBoost was designed in such a way that at every step the sample distribution was adapted to put more weight on misclassified samples and less weight on correctly classified samples. The final prediction is a weighted average of all the weak learners, where more weight is placed on stronger learners.
- Later, it was discovered that AdaBoost can also be expressed as in terms of the more general framework of additive models with a particular loss function (the exponential loss).
- So, the main differences between AdaBoost and GBM are as follows:-

1. The main difference therefore is that Gradient Boosting is a generic algorithm to find approximate solutions to the additive modeling problem, while AdaBoost can be seen as a special case with a particular loss function (Exponential loss function). Hence, gradient boosting is much more flexible.
2. AdaBoost can be interepted from a much more intuitive perspective and can be implemented without the reference to gradients by reweighting the training samples based on classifications from previous learners.
3. In Adaboost, shortcomings are identified by high-weight data points while in Gradient Boosting, shortcomings of existing weak learners are identified by gradients.
4. Adaboost is more about 'voting weights' and Gradient boosting is more about 'adding gradient optimization'.
5. Adaboost increases the accuracy by giving more weightage to the target which is misclassified by the model. At each iteration, Adaptive boosting algorithm changes the sample distribution by modifying the weights attached to each of the instances. It increases the weights of the wrongly predicted instances and decreases the ones of the correctly predicted instances.

6. AdaBoost implementation in Python

- Now, we come to the implementation part of AdaBoost algorithm in Python.
- The first step is to load the required libraries.

6.1 Import libraries

In [1]:
```
# This Python 3 environment comes with many helpful analytics libraries installed
# It is defined by the kaggle/python docker image: https://github.com/kaggle/docker-pytho
# For example, here's several helpful packages to load in

import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)

# Input data files are available in the "../input/" directory.
# For example, running this (by clicking run or pressing Shift+Enter) will list all files

import os
for dirname, _, filenames in os.walk('/kaggle/input'):
    for filename in filenames:
        print(os.path.join(dirname, filename))

# Any results you write to the current directory are saved as output.
```

6.2 Load dataset

In [2]:
```
iris = pd.read_csv('Iris.csv')
```

6.3 EDA

Preview dataset

In [3]:
```
iris.head()
```

Out[3]:

	Id	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm	Species
0	1	5.1	3.5	1.4	0.2	Iris-setosa
1	2	4.9	3.0	1.4	0.2	Iris-setosa
2	3	4.7	3.2	1.3	0.2	Iris-setosa
3	4	4.6	3.1	1.5	0.2	Iris-setosa
4	5	5.0	3.6	1.4	0.2	Iris-setosa

View summary of dataframe

In [4]:
```
iris.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 150 entries, 0 to 149
Data columns (total 6 columns):
 #   Column         Non-Null Count  Dtype  
---  ------         --------------  -----  
 0   Id             150 non-null    int64  
 1   SepalLengthCm  150 non-null    float64
 2   SepalWidthCm   150 non-null    float64
 3   PetalLengthCm  150 non-null    float64
 4   PetalWidthCm   150 non-null    float64
 5   Species        150 non-null    object 
dtypes: float64(4), int64(1), object(1)
memory usage: 7.2+ KB
```

We can see that there are no missing values in the dataset.

Declare feature vector and target variable

In [5]:
```
X = iris[['SepalLengthCm','SepalWidthCm','PetalLengthCm','PetalWidthCm']]

X.head()
```

Out[5]:

	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm
0	5.1	3.5	1.4	0.2
1	4.9	3.0	1.4	0.2
2	4.7	3.2	1.3	0.2
3	4.6	3.1	1.5	0.2
4	5.0	3.6	1.4	0.2

In [6]:
```
y = iris['Species']

y.head()
```

Out[6]:
```
0    Iris-setosa
1    Iris-setosa
2    Iris-setosa
3    Iris-setosa
4    Iris-setosa
Name: Species, dtype: object
```

In [7]:
```
from sklearn.preprocessing import LabelEncoder

le=LabelEncoder()

y=le.fit_transform(y)
```

6.4 Split dataset into training set and test set

In [8]:
```
# Import train_test_split function
from sklearn.model_selection import train_test_split

# Split dataset into training set and test set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)
```

6.5 Build the AdaBoost model

In [9]:
```
# Import the AdaBoost classifier
from sklearn.ensemble import AdaBoostClassifier

# Create AdaBoostClassifier with SAMME algorithm
abc = AdaBoostClassifier(n_estimators=50, learning_rate=1, random_state=0, algorithm='SAI

# Train Adaboost Classifier
model1 = abc.fit(X_train, y_train)

# Predict the response for test dataset
y_pred = model1.predict(X_test)
```

Create Adaboost Classifier

- The most important parameters are `base_estimator` , `n_estimators` and `learning_rate` .
- **base_estimator** is the learning algorithm to use to train the weak models. This will almost always not needed to be changed because by far the most common learner to use with AdaBoost is a decision tree – this parameter's default argument.
- **n_estimators** is the number of models to iteratively train.
- **learning_rate** is the contribution of each model to the weights and defaults to 1. Reducing the learning rate will mean the weights will be increased or decreased to a small degree, forcing the model train slower (but sometimes resulting in better performance scores).
- **loss** is exclusive to AdaBoostRegressor and sets the loss function to use when updating weights. This defaults to a linear loss function however can be changed to square or exponential.

6.6 Evaluate Model

Let's estimate, how accurately the classifier or model can predict the type of cultivars.

In [10]:
```
#import scikit-learn metrics module for accuracy calculation
from sklearn.metrics import accuracy_score


# calculate and print model accuracy
print("AdaBoost Classifier Model Accuracy:", accuracy_score(y_test, y_pred))
```

```
AdaBoost Classifier Model Accuracy: 1.0
```

- In this case, we got an accuracy of 86.67%, which will be considered as a good accuracy.

7. Advantages and disadvantages of AdaBoost

- The advantages are as follows:

  1. AdaBoost is easy to implement.
  2. It iteratively corrects the mistakes of the weak classifier and improves accuracy by combining weak learners.
  3. We can use many base classifiers with AdaBoost.
  4. AdaBoost is not prone to overfitting.

- The disadvantages are as follows:

  1. AdaBoost is sensitive to noise data.
  2. It is highly affected by outliers because it tries to fit each point perfectly.
  3. AdaBoost is slower compared to XGBoost.

Lungs Disease Prediction

In [11]:
```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
%matplotlib inline
```

In [12]:
```
df=pd.read_csv('cancer patient data sets.csv')
```

In [13]:
```
df.head()
```

Out[13]:

	index	Patient Id	Age	Gender	Air Pollution	Alcohol use	Dust Allergy	OccuPational Hazards	Genetic Risk	chronic Lung Disease	...	Fatigue	Wei L
0	0	P1	33	1	2	4	5	4	3	2	...	3	
1	1	P10	17	1	3	1	5	3	4	2	...	1	
2	2	P100	35	1	4	5	6	5	5	4	...	8	
3	3	P1000	37	1	7	7	7	7	6	7	...	4	
4	4	P101	46	1	6	8	7	7	7	6	...	3	

5 rows × 26 columns

In [14]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000 entries, 0 to 999
Data columns (total 26 columns):
 #   Column                    Non-Null Count  Dtype 
---  ------                    --------------  ----- 
 0   index                     1000 non-null   int64 
 1   Patient Id                1000 non-null   object
 2   Age                       1000 non-null   int64 
 3   Gender                    1000 non-null   int64 
 4   Air Pollution             1000 non-null   int64 
 5   Alcohol use               1000 non-null   int64 
 6   Dust Allergy              1000 non-null   int64 
 7   OccuPational Hazards      1000 non-null   int64 
 8   Genetic Risk              1000 non-null   int64 
 9   chronic Lung Disease      1000 non-null   int64 
 10  Balanced Diet             1000 non-null   int64 
 11  Obesity                   1000 non-null   int64 
 12  Smoking                   1000 non-null   int64 
 13  Passive Smoker            1000 non-null   int64 
 14  Chest Pain                1000 non-null   int64 
 15  Coughing of Blood         1000 non-null   int64 
 16  Fatigue                   1000 non-null   int64 
 17  Weight Loss               1000 non-null   int64 
 18  Shortness of Breath       1000 non-null   int64 
 19  Wheezing                  1000 non-null   int64 
 20  Swallowing Difficulty     1000 non-null   int64 
 21  Clubbing of Finger Nails  1000 non-null   int64 
 22  Frequent Cold             1000 non-null   int64 
 23  Dry Cough                 1000 non-null   int64 
 24  Snoring                   1000 non-null   int64 
 25  Level                     1000 non-null   object
dtypes: int64(24), object(2)
memory usage: 203.3+ KB
```

In [15]:
```
df.describe()
```

Out[15]:

	index	Age	Gender	Air Pollution	Alcohol use	Dust Allergy	OccuPational Hazards	Geneti Ris
count	1000.000000	1000.000000	1000.000000	1000.0000	1000.000000	1000.000000	1000.000000	1000.00000
mean	499.500000	37.174000	1.402000	3.8400	4.563000	5.165000	4.840000	4.58000
std	288.819436	12.005493	0.490547	2.0304	2.620477	1.980833	2.107805	2.12699
min	0.000000	14.000000	1.000000	1.0000	1.000000	1.000000	1.000000	1.00000
25%	249.750000	27.750000	1.000000	2.0000	2.000000	4.000000	3.000000	2.00000
50%	499.500000	36.000000	1.000000	3.0000	5.000000	6.000000	5.000000	5.00000
75%	749.250000	45.000000	2.000000	6.0000	7.000000	7.000000	7.000000	7.00000
max	999.000000	73.000000	2.000000	8.0000	8.000000	8.000000	8.000000	7.00000

8 rows × 24 columns

In [16]:
```
df.shape
```

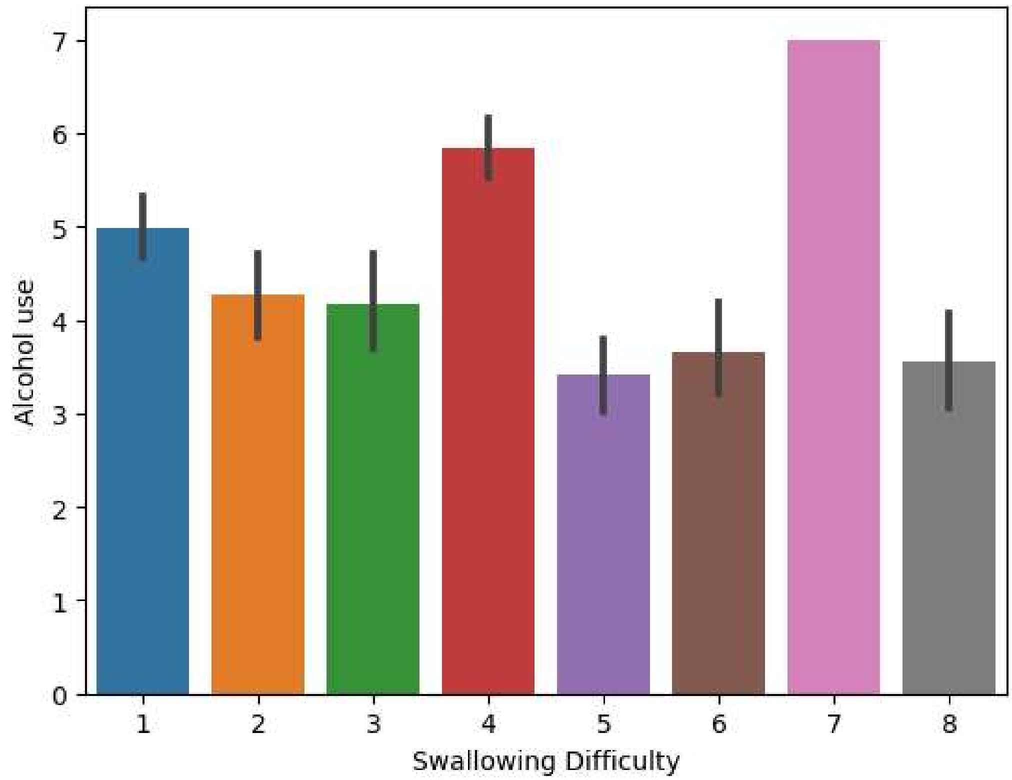
Out[16]:
```
(1000, 26)
```

In [17]:
```
df.isnull().sum()
```

Out[17]:
```
index                       0
Patient Id                  0
Age                         0
Gender                      0
Air Pollution               0
Alcohol use                 0
Dust Allergy                0
OccuPational Hazards        0
Genetic Risk                0
chronic Lung Disease        0
Balanced Diet               0
Obesity                     0
Smoking                     0
Passive Smoker              0
Chest Pain                  0
Coughing of Blood           0
Fatigue                     0
Weight Loss                 0
Shortness of Breath         0
Wheezing                    0
Swallowing Difficulty       0
Clubbing of Finger Nails    0
Frequent Cold               0
Dry Cough                   0
Snoring                     0
Level                       0
dtype: int64
```

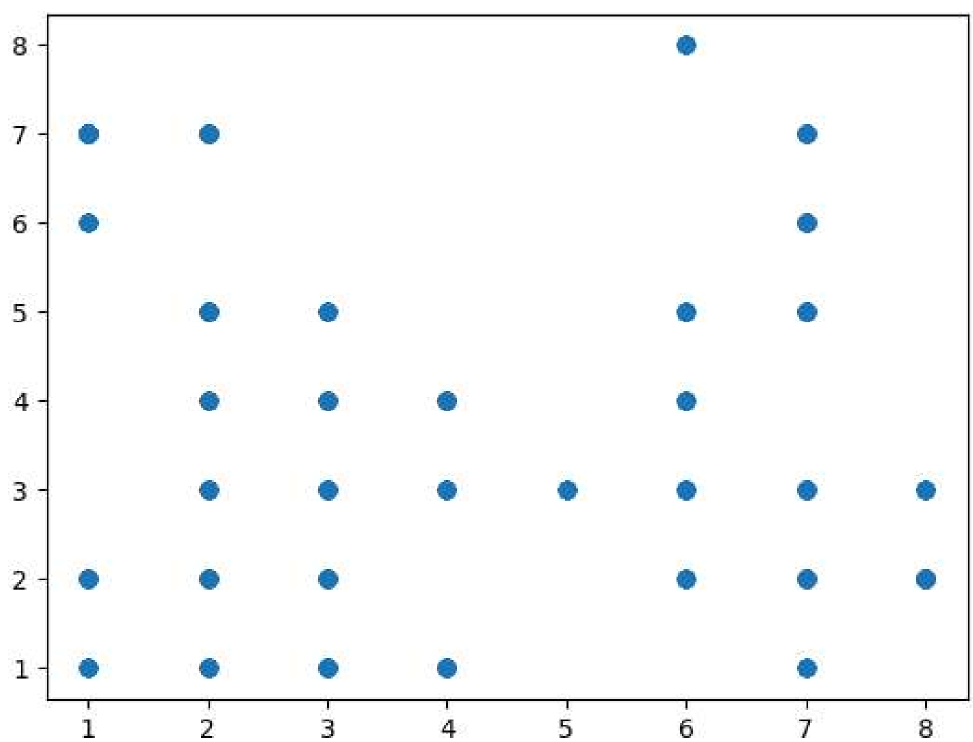
In [18]:
```
sns.barplot(y='Alcohol use', x ='Swallowing Difficulty', data=df)
```

Out[18]: <Axes: xlabel='Swallowing Difficulty', ylabel='Alcohol use'>



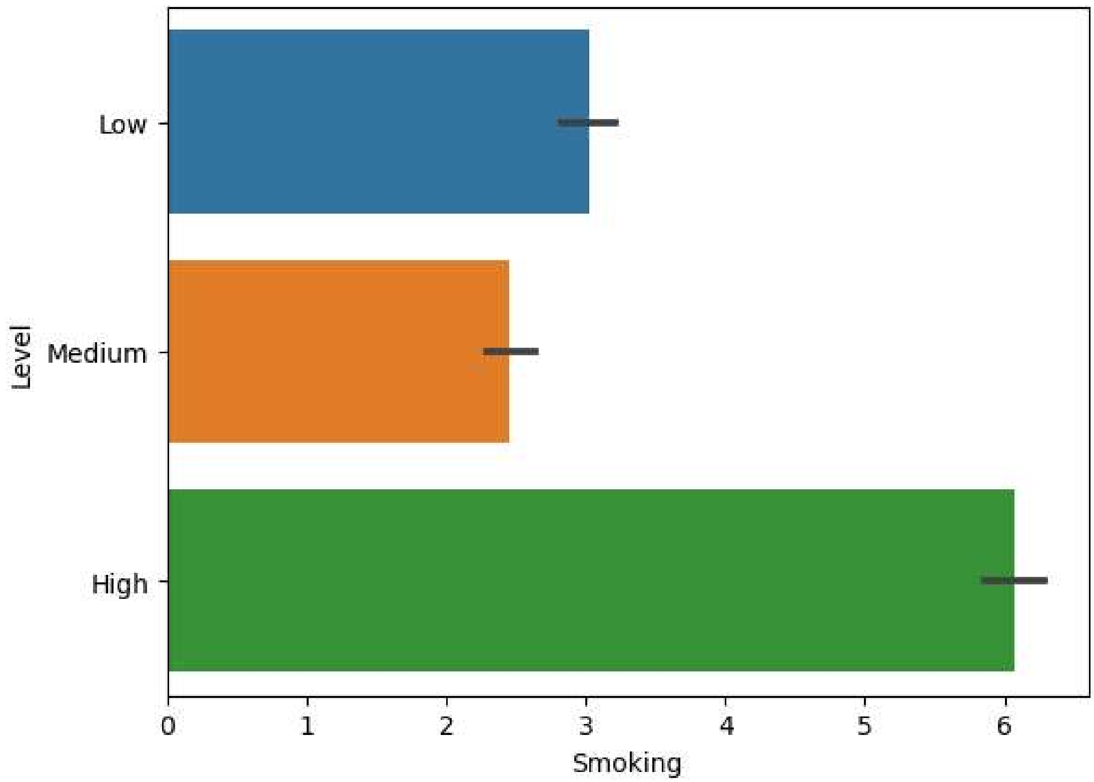
In [19]:
```
plt.scatter(y='Weight Loss', x ='Smoking', data=df)
```

Out[19]: <matplotlib.collections.PathCollection at 0x1ce4d97ec50>



In [20]:
```
sns.barplot(y='Level', x ='Smoking', data=df)
```

Out[20]: <Axes: xlabel='Smoking', ylabel='Level'>

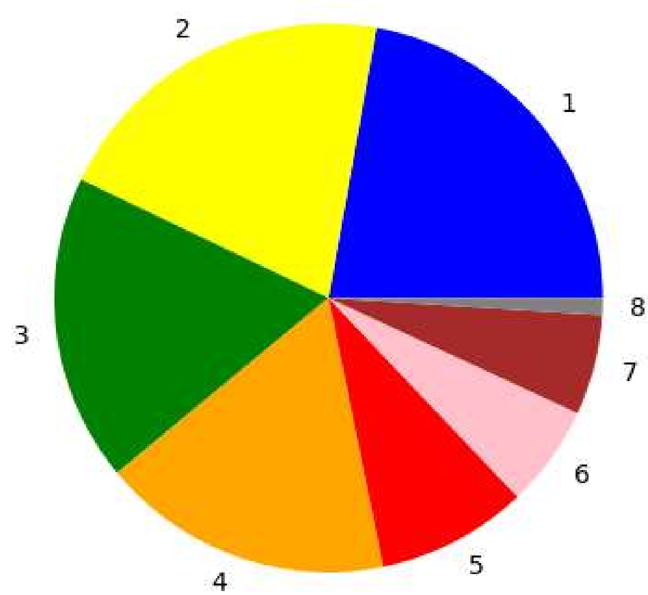


In [21]:
```python
labels= ['1','2','3','4','5','6','7','8']

colors=['blue', 'yellow', 'green', 'orange','red','pink','brown','grey']

plt.pie(x=df['Smoking'].value_counts() ,labels=labels, colors=colors)
```

Out[21]:
```
([<matplotlib.patches.Wedge at 0x1ce4d96ad10>,
  <matplotlib.patches.Wedge at 0x1ce4d96f010>,
  <matplotlib.patches.Wedge at 0x1ce4e0f0150>,
  <matplotlib.patches.Wedge at 0x1ce4e0f1350>,
  <matplotlib.patches.Wedge at 0x1ce4e0f2a90>,
  <matplotlib.patches.Wedge at 0x1ce4e0f3f10>,
  <matplotlib.patches.Wedge at 0x1ce4e101210>,
  <matplotlib.patches.Wedge at 0x1ce4e102610>],
 [Text(0.8431423011403422, 0.7064779260725482, '1'),
  Text(-0.5024662023210625, 0.9785334514083048, '2'),
  Text(-1.0917539039082917, -0.1344373954709198, '3'),
  Text(-0.366101483824868, -1.0372895948293466, '4'),
  Text(0.5086045630381768, -0.9753570620325387, '5'),
  Text(0.8939636055409079, -0.6409594932351812, '6'),
  Text(1.0662956382069828, -0.27021031057449163, '7'),
  Text(1.0994572176396824, -0.034551795611921995, '8')])
```



In [22]:
```python
from sklearn.preprocessing import LabelEncoder
encoder=LabelEncoder()
df['Level'] = encoder.fit_transform(df['Level'])
```

In [23]:
```python
df=df.drop('Patient Id',axis=1)
```

In [24]:
```python
X=df.drop('Level',axis=1)
y=df['Level']
```

In [25]:
```
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report,confusion_matrix
```

In [26]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42
```

In [27]:
```
import xgboost as xgb
```

In [28]:
```
# Convert the data into DMatrix format
dtrain = xgb.DMatrix(X_train, label=y_train)
dtest = xgb.DMatrix(X_test, label=y_test)
```

In [29]:
```
# Set the parameters for XGBoost
params = {
    'objective': 'multi:softmax',
    'num_class': 3,
    'max_depth': 3,
    'eta': 0.1,
    'eval_metric': 'merror'
}
```

In [30]:
```
# Train the XGBoost model
num_rounds = 10
model = xgb.train(params, dtrain, num_rounds)
```

In [31]:
```
# Make predictions on the test set
y_pred = model.predict(dtest)
```

In [32]:
```
# Calculate the accuracy of the model
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)
```

```
Accuracy: 1.0
```

In [33]:
```
print(classification_report(y_test,y_pred))
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        82
           1       1.00      1.00      1.00        55
           2       1.00      1.00      1.00        63

    accuracy                           1.00       200
   macro avg       1.00      1.00      1.00       200
weighted avg       1.00      1.00      1.00       200
```

In [34]:
```
from sklearn.ensemble import AdaBoostClassifier
```

In [35]:

```
abc = AdaBoostClassifier(n_estimators=10, learning_rate=1.0, algorithm='SAMME', random_s
```

In [36]:

```
abc.fit(X_train,y_train)
```

Out[36]:

```
AdaBoostClassifier
AdaBoostClassifier(algorithm='SAMME', n_estimators=10)
```

(https://scikit-learn.org/1.4/modules/generated/sklearn.ens

In [37]:

```
y_pred1 = abc.predict(X_test)
```

In [38]:

```
print(classification_report(y_test,y_pred1))
```

```
              precision    recall  f1-score   support

           0       0.99      1.00      0.99        82
           1       0.98      1.00      0.99        55
           2       1.00      0.97      0.98        63

    accuracy                           0.99       200
   macro avg       0.99      0.99      0.99       200
weighted avg       0.99      0.99      0.99       200
```

In [ ]: