Medical System

* **train\_test\_split** is a utility function in the **sklearn.model\_selection** module of the scikit-learn library in Python. It is used to split a dataset into two subsets: one for training a machine learning model and one for testing the model. This function is essential for evaluating the performance of a machine learning model.
* Label encoding is a process of converting categorical data into numerical data. Many machine learning algorithms cannot work directly with categorical data and require numerical input. The **LabelEncoder** helps to transform these categorical labels into a numeric format.

Example:

from sklearn.preprocessing import LabelEncoder

# Sample categorical data

categories = ["apple", "banana", "cherry", "apple", "cherry", "banana"]

# Create an instance of LabelEncoder

le = LabelEncoder()

# Fit the encoder

le.fit(categories)

# The encoder learns the following mapping:

# "apple" -> 0

# "banana" -> 1

# "cherry" -> 2

# Transform the data

encoded\_categories = le.transform(categories)

print("Encoded categories:", encoded\_categories)

# Output: Encoded categories: [0 1 2 0 2 1]

# Inverse transform the data

decoded\_categories = le.inverse\_transform(encoded\_categories)

print("Decoded categories:", decoded\_categories)

# Output: Decoded categories: ['apple' 'banana' 'cherry' 'apple' 'cherry' 'banana']

* X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=20)

X This represents the feature set. It could be a numpy array, pandas DataFrame, or any array-like structure containing the input features.

y This represents the target labels or outcomes you want to predict. It should have the same length as X.

 **test\_size=0.3**:

* This parameter specifies the proportion of the dataset to include in the test split.
* 0.3 means 30% of the data will be used for testing, and the remaining 70% will be used for training.
* Alternatively, you can specify an integer to directly indicate the number of samples to include in the test set.

 **random\_state=20**:

* This parameter ensures reproducibility of the split. By setting random\_state to a fixed value (e.g., 20), you ensure that the same split will be generated every time you run the code.

 The fit() method takes the target labels (in this case, y) and learns the unique classes. It assigns each unique class a numeric value.

 For example, if y contains the values ['cat', 'dog', 'fish'], le.fit(y) will map these values to numeric codes such as {cat: 0, dog: 1, fish: 2}

Support Vector Machine (SVC):

Example: Imagine one judge (SVC) likes to draw a straight line between the best and worst projects. This judge is very precise and tries to separate the projects into two groups perfectly.

Random Forest Classifier:

Example: Another judge (Random Forest) is like a wise old owl who looks at the results from many different angles (trees). This judge makes a decision based on the majority vote from all these angles.

Gradient Boosting Classifier:

Example: This judge (Gradient Boosting) starts with a basic opinion and then keeps correcting themselves by focusing on the mistakes they made previously, trying to get better with each new opinion.

K-Neighbors Classifier (KNN):

Example: Imagine another judge (KNN) who asks the opinions of the nearest friends (neighbors) to decide which project is best. This judge believes that the projects close to each other are likely to be similar in quality.

Naive Bayes Classifier:

Example: The last judge (Naive Bayes) makes decisions based on past experiences and probabilities. This judge thinks about how likely each project is to be good based on some features they notice.

Accuracy Score:

Example: Imagine you have a score sheet where you note how many projects each judge correctly identifies as good or bad. The accuracy score is like the percentage of correct answers each judge gives. If a judge correctly identifies 8 out of 10 projects, their accuracy score is 80%.

Confusion Matrix:

Example: A confusion matrix is like a big table where you keep track of how often each judge is correct and how often they make mistakes. It shows how many good projects were correctly identified as good (True Positives), how many bad projects were incorrectly identified as good (False Positives), how many bad projects were correctly identified as bad (True Negatives), and how many good projects were incorrectly identified as bad (False Negatives).

A diagram of values

Description automatically generated

Make\_classification

Imagine You’re Setting Up a Treasure Hunt

Let's say you and your friends are setting up a treasure hunt. You have a big field, and you want to place some treasures (let’s call them "good spots") and some decoys (let’s call them "bad spots"). You also want to make a map with clues to find these spots. The make\_classification function is like setting up this treasure hunt.

Setting Up the Treasure Hunt (make\_classification)

When you use make\_classification, you're deciding where to place the treasures and decoys and how to make the clues (features) for finding them. Here’s how it works:

Number of Spots (n\_samples):

Example: Let’s say you decide to place 100 spots in the field (both good and bad). This is like setting n\_samples=100.

Number of Clues (n\_features):

Example: You decide that each spot will have two clues to help find it, like “It’s near a tree” and “It’s close to a river.” This is like setting n\_features=2.

Number of Important Clues (n\_informative):

Example: Only some clues actually help you find the good spots. Maybe both clues are helpful, so n\_informative=2.

Extra Clues (n\_redundant and n\_repeated):

Example: Sometimes, you might have extra clues that repeat the important ones or don’t add new information. If you don’t have any extra or repeated clues, you set n\_redundant=0 and n\_repeated=0.

Number of Classes (n\_classes):

Example: You have two types of spots: good spots (where the treasure is) and bad spots (decoys). This is like setting n\_classes=2.

Clusters per Class (n\_clusters\_per\_class):

Example: If you group the good spots in one part of the field and the bad spots in another, each group is a cluster. If you have one cluster per type, you set n\_clusters\_per\_class=1.

Randomness (random\_state):

Example: If you want to make sure the spots are placed the same way every time you play, you use a specific random seed. Setting random\_state=42 makes the treasure hunt the same each time.

A screenshot of a computer

Description automatically generated

A screenshot of a computer program

Description automatically generated

A screenshot of a computer code

Description automatically generated

pickle.dump(svc,open("svc.pkl",'wb'))

This line of code is using Python's pickle module to serialize the trained Support Vector Classifier (SVC) model (svc) and save it to a file named "svc.pkl" in binary mode ('wb').

reshape(1,-1)

Imagine you have a bunch of toys, and each toy has a different shape and size. Now, you want to organize them neatly in a box so that they fit perfectly. But, you want to make sure that no matter how many toys you have, they all fit in one row in the box.

That's what .reshape(1, -1) does in programming. It's like telling a magical organizer to arrange all your toys in one line, no matter how many you have. The -1 part tells the organizer to figure out how many toys there are and adjust the box's size accordingly. And the 1 part tells the organizer that you want all the toys to be in one row.

So, if you do [col for col in pre.values], you are creating a new list that contains all the rows from pre.values.

A screenshot of a computer

Description automatically generated