Project Report

Xinan Wang

802-829-7764

wang.xina@northeastern.edu

Percentage of Effort Contributed by Student:_____100%_____

Signature of Student:_____Xinan Wang_____

Submission Date:_____12/17/2022_____

IE7300 Project Report

Xinan Wang

December 15 2022

1 Project Overview

The goal of this project is to develop a successful classification model to capture the motions made by the user and predict the motions for the future users. By analyzing the detecting data of each marker position on each axis, we predicted the type of the hand postures.

We started this project by doing the exploratory data analysis, to better understand the contents of the dataset and the relationships of each features. Next, I did the data pre-processing. I normalized the numerical features to decrease the variance within dataset. Also, I split the dataset into training data and testing data by 80:20. Next step, I imported a few of classification models and then trained them by using the training dataset. I also tested the models after training by using the testing dataset. Finally, I evaluated the model performance by checking the training dataset accuracy, testing dataset accuracy, and confusion matrix. By comparing the performance, variance/bias trade-off, model running time and many other aspects of the models, I finally picked up an optimal one as my final model to analyze the hand posture motions dataset.

The entire process of the above description will be showed in the next pages of this report. Also, in the end of the report, I will attach my source code. The dataset of this project is Motion Capture Hand Postures Data Set(https://archive.ics.uci.edu/ml/datasets/Motion+Capture+Hand+Postures) from UCI Machine Learning Repository. The dataset is used for posture recognition via classification.

2 Problem Statement

For the hospitals, they want to quickly get to know the motion of the hand postures that a user is doing, to analyze the user's potential diseases or make the decision of next step medical plan. Therefore, they put 11 marker positions on each of the x, y, z axis, to collect the source data. Then, by using the machine learning algorithms, the model will quickly let them know which hand posture that the user is doing.

The advantage of using machine learning models are that they are accurate and efficient. They are also automatically run so that they need less human supervisions. This will reduce the cost of the hospitals. Also, the machine learning models also have the ability to deal with large size dataset or Big Data.

Therefore, we applied the machine learning classification models on our project, to help us analyze the type of the hand postures that the user is doing.

3 Data Used

The dataset has following features:

1. class: Integer. The class ID of the given record. Ranges from 1 to 5 with 1 = Fist(with thumb out), 2 = Stop(hand flat), 3 = Point1(point with pointer finger), 4 = Point2(point with pointer and middle fingers), 5 = Grab(fingers curled as if to grab).

2. User – Integer. The ID of the user that contributed the record. No meaning other than as identifier.

3. Xi – Real. The x-coordinate of the i-th unlabeled marker position. 'i' range from 0 to 11. So, we have 'X0', 'X1', 'X2', 'X3', 'X4', 'X5', 'X6', 'X7', 'X8', 'X9', 'X10', 'X11' as the attributes in the dataset.

4. Yi – Real. The y-coordinate of the i-th unlabeled marker position. 'i' range from 0 to 11. So, we have 'Y0', 'Y1', 'Y2', 'Y3', 'Y4', 'Y5', 'Y6', 'Y7', 'Y8', 'Y9', 'Y10', 'Y11'as the attributes in the dataset.

5. Zi – Real. The z-coordinate of the i-th unlabeled marker position. 'i' range from 0 to 11. So, we have 'Z0', 'Z1', 'Z2', 'Z3', 'Z4', 'Z5', 'Z6', 'Z7', 'Z8', 'Z9', 'Z10', 'Z11'as the attributes in the dataset.

Please be noted that each record is a set. The i-th marker of a given record does not necessary correspond to the i-th marker of a different record. One may randomly permute the visible (i.e., not missing) marker of a given record without changing the set that the record represents. For the sake of convenience, all visible markers of a given record are given a lower index than any missing marker. A class is not guaranteed to have even a single record with all markers visible.

Variable	Definition
class	The class ID of the given record. Ranges from 1 to 5 with 1 = Fist(with thumb out), 2 = Stop(hand flat), 3 = Point1(point with pointer finger), 4 = Point2(point with pointer and middle fingers), 5 = Grab(fingers curled as if to grab).
user	The ID of the user that contributed the record.
Xi	The x-coordinate of the i-th unlabeled marker position. 'i' range from 0 to 11. So, we have 'X0', 'X1', 'X2', 'X3', 'X4', 'X5', 'X6', 'X7', 'X8', 'X9', 'X10', 'X11' as the attributes in the dataset.
Yi	The y-coordinate of the i-th unlabeled marker position. 'i' range from 0 to 11. So, we have 'Y0', 'Y1', 'Y2', 'Y3', 'Y4', 'Y5', 'Y6', 'Y7', 'Y8', 'Y9', 'Y10', 'Y11' as the attributes in the dataset.
Zi	The z-coordinate of the i-th unlabeled marker position. 'i' range from 0 to 11. So, we have 'Z0', 'Z1', 'Z2', 'Z3', 'Z4', 'Z5', 'Z6', 'Z7', 'Z8', 'Z9', 'Z10', 'Z11' as the attributes in the dataset.

4 Analysis

4.1 EDA

First of all, since the dataset contains lots of missing values which are recorded as ?, we need to clean up the missing values. Because when the data is recorded as ?, we get no value at that data point, I used integer 0 to replace the ? value.

```
In [4]:
1 df = df.replace('?',0).astype(float)
2 df
```

Out[4]:

	Class	User	X0	Y0	Z0	X1	Y1	Z1	X2	Y2	...	
0	1.0	0.0	54.263880	71.466776	-64.807709	76.895635	42.462500	-72.780545	36.621229	81.680557	...	0.00
1	1.0	0.0	56.527558	72.266609	-61.935252	39.135978	82.538530	-49.596509	79.223743	43.254091	...	0.00
2	1.0	0.0	55.849928	72.469064	-62.562788	37.988804	82.631347	-50.606259	78.451526	43.567403	...	0.00
3	1.0	0.0	55.329647	71.707275	-63.688956	36.561863	81.868749	-52.752784	86.320630	68.214645	...	0.00
4	1.0	0.0	55.142401	71.435607	-64.177303	36.175818	81.556874	-53.475747	76.986143	42.426849	...	0.00
...	...	...	...	...	...	...	...	...	...	...	...	
78090	5.0	14.0	54.251127	129.177414	-44.252511	27.720784	107.810661	11.099282	-1.270139	122.758679	...	-6.54
78091	5.0	14.0	54.334883	129.253842	-44.016320	27.767911	107.914808	11.069842	-30.334054	77.858214	...	-62.13
78092	5.0	14.0	54.151540	129.269502	-44.173273	27.725978	108.034006	11.020347	-22.574718	104.222208	...	3.83
78093	5.0	14.0	27.915311	108.007390	10.814957	-0.910435	122.464093	-47.271248	-30.084588	77.705861	...	-63.22
78094	5.0	14.0	27.808705	108.092877	11.107857	-30.031402	77.740295	-17.453099	-1.091566	122.827638	...	-63.02

Because this dataset doesn't have any categorical features, we don't need to transform the categorical features into numerical features.

Next, we wanted to figure out the distribution inside each feature. By using describe() method, we get the count, mean, standard deviation(std), min, max of each feature.

```
In [5]: 1 df.describe().transpose()
```

Out[5]:

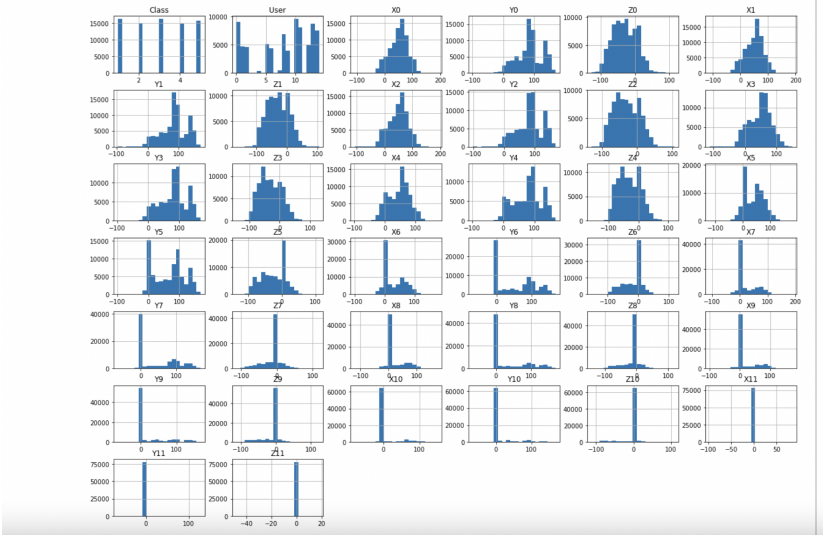
	count	mean	std	min	25%	50%	75%	max
Class	78095.0	2.983776	1.421152	1.000000	2.000000	3.000000	4.000000	5.000000
User	78095.0	7.959229	4.697754	0.000000	5.000000	9.000000	12.000000	14.000000
X0	78095.0	50.346308	32.695886	-108.552738	29.295141	54.620245	72.488833	190.017835
Y0	78095.0	85.813150	40.203448	-98.233756	63.497746	86.526334	113.108673	169.175464
Z0	78095.0	-29.985096	34.361971	-126.770872	-56.356593	-30.864248	-1.419462	113.345119
X1	78095.0	49.595844	32.477961	-111.685241	28.755679	54.215714	71.763080	188.691997
			452297	-96.142589	64.156450	87.543493	116.228881	170.209350
Z1	78095.0	-29.509579	34.764460	-166.006838	-57.360408	-30.185331	-0.368080	104.697852
X2	78095.0	48.612744	33.605155	-106.886524	25.173405	53.814592	71.561988	188.760168
Y2	78095.0	83.772387	41.022710	-100.789312	58.053733	86.459935	106.661720	168.186466
Z2	78095.0	-30.560906	35.120384	-129.595296	-58.654339	-32.356535	-0.946134	104.590879
X3	78095.0	48.064123	34.027102	-111.761053	22.749246	53.888390	71.216793	151.033472
Y3	78095.0	81.339253	42.093648	-97.603414	53.094602	85.477367	105.169964	168.292018
Z3	78095.0	-30.871450	35.892811	-143.540529	-59.223756	-33.440705	-0.554393	129.316870
X4	78095.0	46.472199	34.849901	-99.107635	16.989314	52.670592	71.036529	172.275978
Y4	78095.0	77.206639	44.634164	-97.948829	43.857348	84.372784	104.299414	168.258643
Z4	78095.0	-30.715371	36.210018	-157.199089	-59.733553	-32.521378	0.000000	119.237203
X5	78095.0	39.197998	36.296891	-120.657868	0.000000	43.772063	67.224900	180.563322
Y5	78095.0	67.821645	49.513061	-97.468548	17.838554	79.001681	101.803483	167.926171
Z5	78095.0	-25.222531	35.482206	-135.699430	-54.402155	-20.158252	0.000000	110.898899
X6	78095.0	30.559226	36.682674	-100.084275	0.000000	15.798763	61.142447	176.409004
Y6	78095.0	56.018467	52.886406	-67.283707	0.000000	52.723120	99.126611	168.598384
Z6	78095.0	-17.822143	31.784525	-153.449813	-40.880986	0.000000	0.000000	117.914907
X7	78095.0	22.172219	35.053134	-108.605639	0.000000	0.000000	49.409962	189.221529
Y7	78095.0	44.113298	52.673598	-64.972157	0.000000	0.000000	93.725568	169.127359
Z7	78095.0	-10.157104	26.043592	-113.733105	-19.296698	0.000000	0.000000	117.815967
X8	78095.0	18.867068	33.717126	-121.182089	0.000000	0.000000	35.523460	173.906643
Y8	78095.0	33.668643	49.403134	-65.077550	0.000000	0.000000	78.198374	169.322843
Z8	78095.0	-9.535343	25.321953	-142.654497	-7.750354	0.000000	0.000000	119.213101
X9	78095.0	16.802219	33.583912	-99.231688	0.000000	0.000000	12.497466	174.054403
Y9	78095.0	24.719138	44.032514	-64.734284	0.000000	0.000000	37.681938	167.942588
Z9	78095.0	-8.524955	23.820691	-113.397327	0.000000	0.000000	0.000000	123.380512
X10	78095.0	10.154914	26.873653	-80.196289	0.000000	0.000000	0.000000	149.486224
Y10	78095.0	13.979146	34.451890	-65.019295	0.000000	0.000000	0.000000	168.352478
Z10	78095.0	-5.617450	20.805720	-112.668930	0.000000	0.000000	0.000000	108.455548
X11	78095.0	-0.011789	1.149642	-96.951690	0.000000	0.000000	0.000000	84.683328
Y11	78095.0	0.010306	0.888304	-65.432143	0.000000	0.000000	0.000000	127.945490
Z11	78095.0	0.000674	0.393026	-48.274677	0.000000	0.000000	0.000000	18.062286

From the description of the data, we can clearly see that data point Y0, Y1 and Y2 are more possible to get larger values because they have larger mean values compared to the other features. Data point X11, Y11 and Z11 have very small mean value and very small standard deviation. This means X11, Y11 and Z11 have more possibility to get smaller values compared to the other data points. And much of the data observed of them are tightly clustered around

mean values. Y5, Y6, Y7 have top 3 larger standard deviations. This means there is a lot of variance in the observed data of them around the mean.

Next, we analyzed the distribution of each features by plotting the distributions.

```
In [13]:  1 df.hist(figsize=(20,15),bins=20)
          2 plt.show()
```



From the distribution plot, we can find that: (1) The X0, Z0, X1, Z1, X2, Z2, X3, Y3, Z3, X4, Y4, Z4 are normally distributed. (2) Both of the X5, Y5, Z5, X6, Y6, Z6, X7, Y7, Z7, X8, Y8, Z8, X9, Y9, X10, Y10, Z10, X11, Y11, Z11 has a very high bar, and the rest of the data are very less. Also, most of them are skewed.

Next, we wanted to find the counts of the class data in our dataset.

```
In [7]:  1 df['Class'].value_counts()

Out[7]: 3.0    16344
        1.0    16265
        5.0    15733
        2.0    14978
        4.0    14775
        Name: Class, dtype: int64
```

From the python output, we can clearly see that we get 16344 class 3 data, which is the largest amount. We also get 16265 class 1 data, 15733 class 5 data, 14978 class 2 data, and 14775 class 4 data. Our data class are distributed averagely. We don't have the situation that one class amount is several times more than other classes.

Next, we want to explore the relationship between the features and the class types.

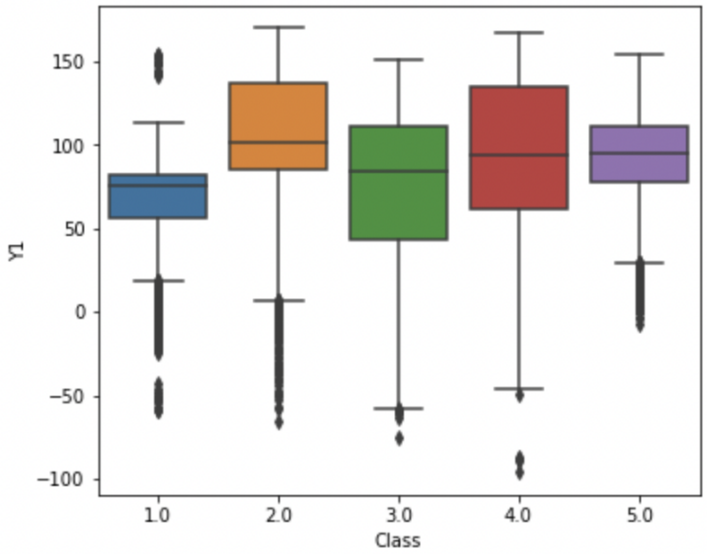
From the previous analysis we can see that Y1 has the largest mean value,

and Y6 has the largest std value. So we analyzed them. We also analyzed X4 and Z6.

Y1:

```
plt.figure(figsize=(6,5))
sns.boxplot(data=df,x='Class',y='Y1')
```

```
<AxesSubplot:xlabel='Class', ylabel='Y1'>
```

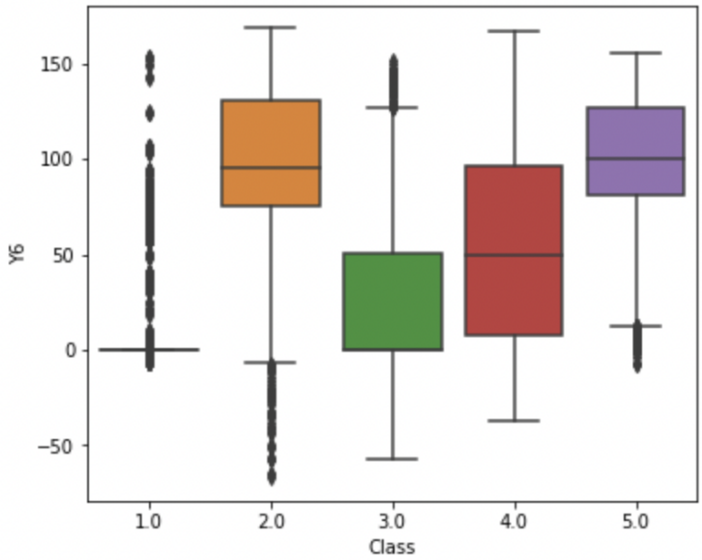


From the Y1 box-plot output, we can clearly see that class 1 Y1 data are concentrated. It has the lowest mean Y1 value(around 70) compared to the other 4 classes. Class 2 has the largest mean Y1 value (around 100) compared to the other 4 classes. Also, class 4 distributed much more widely compared to the other 4 classes. There is a lot of variance in the observed class 4 data around the mean Y1 values.

Y6:

```
1 plt.figure(figsize=(6,5))
2 sns.boxplot(data=df,x='Class',y='Y6')
```

```
<AxesSubplot:xlabel='Class', ylabel='Y6'>
```

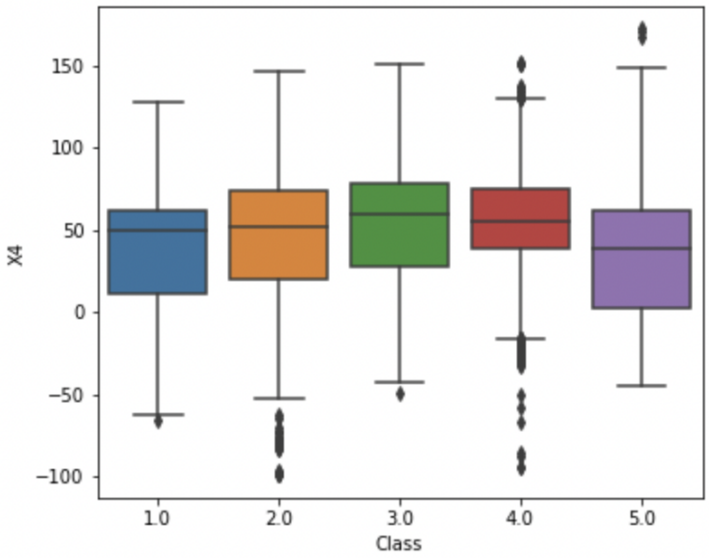


From the Y6 box-plot output, we can clearly see that class 1 Y6 data are outliers. They didn't form a perfect box-shape. Class 5 has the largest mean Y6 value (around 100) compared to the other 4 classes. Also, class 4 distributed much more widely compared to the other 4 classes. There is a lot of variance in the observed class 4 data around the mean Y6 values. Also, from the Y6 output, we can find that the mean and variance of each class are distributed very widely compared to the distribution of Y1. Their variances also have a large differences.

X4:

```
1 plt.figure(figsize=(6,5))
2 sns.boxplot(data=df,x='Class',y='X4')
```

```
<AxesSubplot:xlabel='Class', ylabel='X4'>
```

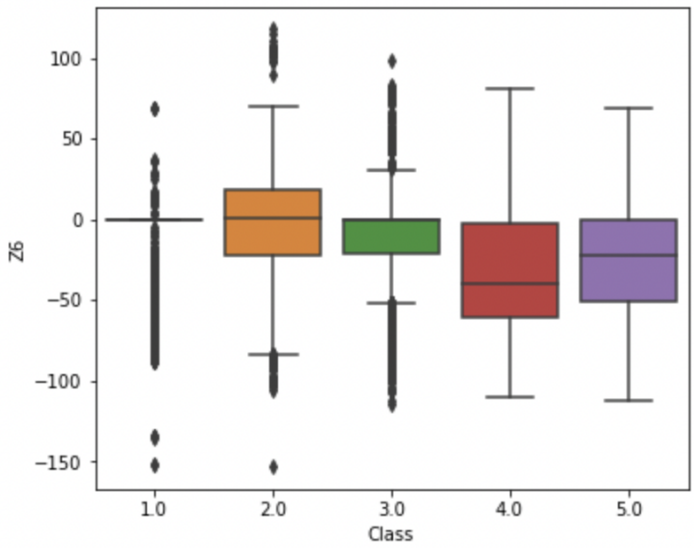


From the X4 box-plot output, we can clearly see that class 3 has the largest mean X4 value (around 100) compared to the other 4 classes. Also, class 5 distributed much more widely compared to the other 4 classes. There is a lot of variance in the observed class 5 data around the mean X4 values. Class 4 is the most concentrated. It has the smallest variance so their values are tightly clustered around their mean. Also, from the Y6 output, we can find that the mean and variance of each class are distributed very averagely compared to the distribution of Y6.

Z6:

```
1 plt.figure(figsize=(6,5))
2 sns.boxplot(data=df,x='Class',y='Z6')
```

```
<AxesSubplot:xlabel='Class', ylabel='Z6'>
```



From the Z6 box-plot output, we can clearly see that class 1 Z6 data are outliers. They didn't form a perfect box-shape. Class 2 has the largest mean Z6 value (around 100) compared to the other 4 classes. Also, class 4 distributed much more widely compared to the other 4 classes. There is a lot of variance in the observed class 4 data around the mean Z6 values. Also, from the Z6 output, we can find that the mean and variance of each class are distributed very widely compared to the distribution of X4. Their variances also have a large differences.

4.2 Algorithm and models

Before we applying the classification algorithms and models on our dataset, we define the features as X and the class as Y. Then, we scale the dataset and split the dataset into training and testing dataset as 80:20.

```
In [21]:
1 X = df.iloc[:,1:]
2 Y = df.iloc[:,0]
```

```
Out[21]: 0        1.0
         1        1.0
         2        1.0
         3        1.0
         4        1.0
                 ...
         78090    5.0
         78091    5.0
         78092    5.0
         78093    5.0
         78094    5.0
         Name: Class, Length: 78095, dtype: float64
```

```
In [22]:
1 # Scale the dataset
2 from sklearn.preprocessing import StandardScaler
3 sc = StandardScaler()
4 X = sc.fit_transform(X)
```

```
In [23]:
1 # Split the dataset into training and testing
2 from sklearn.model_selection import train_test_split
3 Xtrain, Xtest, Ytrain, Ytest = train_test_split( X, Y, test_size = 0
```

(1)SVM

The first model we applied is SVM model. This is a supervised machine learning algorithm used for both classification and regression. The objective of SVM algorithm is to find a hyperplane in an N-dimensional space that distinctly classfies the data points.

Here's the SVM class code:

```
# Create SVM model

from sklearn.base import BaseEstimator, ClassifierMixin
from sklearn.utils import check_random_state
from sklearn.preprocessing import LabelEncoder



def projection_simplex(v, z=1):

    """
    Projection onto the simplex:
        w^* = argmin_w 0.5 ||w-v||^2 s.t. \sum_i w_i = z, w_i >= 0
    """

    n_features = v.shape[0]
    u = np.sort(v)[::-1]
    cssv = np.cumsum(u) - z
    ind = np.arange(n_features) + 1
```

```
    cond = u - cssv / ind > 0
    rho = ind[cond][-1]
    theta = cssv[cond][-1] / float(rho)
    w = np.maximum(v - theta, 0)
    return w


class MulticlassSVM(BaseEstimator, ClassifierMixin):

    def __init__(self, C=1, max_iter=50, tol=0.05,
                 random_state=None, verbose=0):
        self.C = C
        self.max_iter = max_iter
        self.tol = tol,
        self.random_state = random_state
        self.verbose = verbose

    def _partial_gradient(self, X, y, i):
        # Partial gradient for the ith sample.
        g = np.dot(X[i], self.coef_.T) + 1
        g[y[i]] -= 1
        return g

    def _violation(self, g, y, i):
        # Optimality violation for the ith sample.
        smallest = np.inf
        for k in range(g.shape[0]):
            if k == y[i] and self.dual_coef_[k, i] >= self.C:
                continue
            elif k != y[i] and self.dual_coef_[k, i] >= 0:
                continue

            smallest = min(smallest, g[k])

        return g.max() - smallest

    def _solve_subproblem(self, g, y, norms, i):
        # Prepare inputs to the projection.
        Ci = np.zeros(g.shape[0])
        Ci[y[i]] = self.C
        beta_hat = norms[i] * (Ci - self.dual_coef_[:, i]) + g / norms[i]
        z = self.C * norms[i]

        # Compute projection onto the simplex.
        beta = projection_simplex(beta_hat, z)
```

```
        return Ci - self.dual_coef_[:, i] - beta / norms[i]

    def fit(self, X, y):
        n_samples, n_features = X.shape

        # Normalize labels.
        self._label_encoder = LabelEncoder()
        y = self._label_encoder.fit_transform(y)

        # Initialize primal and dual coefficients.
        n_classes = len(self._label_encoder.classes_)
        self.dual_coef_ = np.zeros((n_classes, n_samples), dtype=np.float64)
        self.coef_ = np.zeros((n_classes, n_features))

        # Pre-compute norms.
        norms = np.sqrt(np.sum(X ** 2, axis=1))

        # Shuffle sample indices.
        rs = check_random_state(self.random_state)
        ind = np.arange(n_samples)
        rs.shuffle(ind)

        violation_init = None
        for it in range(self.max_iter):
            violation_sum = 0

            for ii in range(n_samples):
                i = ind[ii]

                # All-zero samples can be safely ignored.
                if norms[i] == 0:
                    continue

                g = self._partial_gradient(X, y, i)
                v = self._violation(g, y, i)
                violation_sum += v

                if v < 1e-12:
                    continue

                # Solve subproblem for the ith sample.
                delta = self._solve_subproblem(g, y, norms, i)

                # Update primal and dual coefficients.
                self.coef_ += (delta * X[i][:, np.newaxis]).T
                self.dual_coef_[:, i] += delta
```

```
            if it == 0:
                violation_init = violation_sum

            vratio = violation_sum / violation_init

            if self.verbose >= 1:
                print("iter", it + 1, "violation", round(vratio,4))

            if vratio < self.tol:
                if self.verbose >= 1:
                    print("Converged")
                break

        return self

    def predict(self, X):
        decision = np.dot(X, self.coef_.T)
        pred = decision.argmax(axis=1)
        return self._label_encoder.inverse_transform(pred)
```

(2)Logistic Regression

Logistic Regression is applied to predict the categorical dependent variable. It is used to predict the categorical dependent variable using a given set of independent variables. Here, we used the multinomial logistic regression.

Here's the logistic regression class code:

```
class MultiClassLogisticRegression:
    """
    Multiclass logistic regression
    """

    def __init__(self, epochs = 10000, threshold=1e-3):
        """
        Constructor for multiclass regression

        Args:
            epochs (int, optional): No of iteration;
                                    Defaults to 10000.
            threshold (_type_, optional): Each iteration thresold.
                                          Defaults to 1e-3.
        """
        self.epochs = epochs
```

```
        self.threshold = threshold

    def train(self, X, y, batch_size=64, lr=0.001, rand_seed=4,
    verbose=False):
        """
        Train the model

        Args:
            X (_type_): Features
            y (_type_): Labels
            batch_size (int, optional): Batch size per iterations.
                                        Defaults to 64.
            lr (float, optional): Learning rate.
                                  Defaults to 0.001.
            rand_seed (int, optional): _description_. Defaults to 4.
            verbose (bool, optional): _description_. Defaults to False.

        Returns:
            _type_: return the instance
        """
        np.random.seed(rand_seed)
        self.classes = np.unique(y)
        self.class_labels = {c:i for i,c in enumerate(self.classes)}
        X = self.add_bias(X)
        y = self.one_hot(y)
        self.loss = []
        self.weights = np.zeros(shape=(len(self.classes),X.shape[1]))
        self.fit_data(X, y, batch_size, lr, verbose)
        return self

    def fit_data(self, X, y, batch_size, lr, verbose):
        i = 0
        while (not self.epochs or i < self.epochs):
            self.loss.append(self.cross_entropy(y, self.predict_(X)))
            idx = np.random.choice(X.shape[0], batch_size)
            X_batch, y_batch = X[idx], y[idx]
            error = y_batch - self.predict_(X_batch)
            update = (lr * np.dot(error.T, X_batch))
            self.weights += update
            if np.abs(update).max() < self.threshold:
                break
            if i % 1000 == 0 and verbose:
                print(' Training Accuray at {} iterations is {}'
                .format(i, self.evaluate_(X, y)))
            i +=1
```

```
def predict(self, X):
    return self.predict_(self.add_bias(X))

def predict_(self, X):
    pre_vals = np.dot(X, self.weights.T).reshape(-1,len(self.classes))
    return self.softmax(pre_vals)

def getLoss(self):
    return self.loss;

def softmax(self, z):
    return np.exp(z) / np.sum(np.exp(z), axis=1).reshape(-1,1)

def predict_classes(self, X):
    self.probs_ = self.predict(X)
    return np.vectorize(lambda c: self.classes[c])
    (np.argmax(self.probs_, axis=1))

def add_bias(self,X):
    return np.insert(X, 0, 1, axis=1)

def get_random_weights(self, row, col):
    return np.zeros(shape=(row,col))

def one_hot(self, y):
    return np.eye(len(self.classes))[np.vectorize(lambda c:
    self.class_labels[c])(y).reshape(-1)]

def score(self, X, y):
    return round(np.mean(self.predict_classes(X) == y),3)

def evaluate_(self, X, y):
    return np.mean(np.argmax(self.predict_(X), axis=1) ==
    np.argmax(y, axis=1))

def cross_entropy(self, y, probs):
    return -1 * np.mean(y * np.log(probs))
```

(3)Decision Tree with Adaboost

A decision tree is a non-parametric supervised learning algorithm, which is utilized for both classification and regression tasks. It has a hierarchical, tree structure, which consists of a root node, branches, internal nodes and leaf nodes.

The AdaBoost algorithm involves using very short(one-level) decision trees as weaker learners that are added sequentially to the ensemble. Each subsequent

model attempts to correct the predictions made by the model before it in the sequence.

Here's the code of decision tree with AdaBoost:

```
class Node():
    def __init__(self, feature_index=None, threshold=None, left=None,
    right=None, info_gain=None, value=None):
        ''' constructor '''

        # for decision node
        self.feature_index = feature_index
        self.threshold = threshold
        self.left = left
        self.right = right
        self.info_gain = info_gain

        # for leaf node
        self.value = value

class DecisionTreeClassifier():
    def __init__(self, min_samples_split=2, max_depth=2):
        ''' constructor '''

        # initialize the root of the tree
        self.root = None

        # stopping conditions
        self.min_samples_split = min_samples_split
        self.max_depth = max_depth

    def build_tree(self, dataset, curr_depth=0):
        ''' recursive function to build the tree '''

        X, Y = dataset[:,:-1], dataset[:,-1]
        num_samples, num_features = np.shape(X)

        # split until stopping conditions are met
        if num_samples>=self.min_samples_split and curr_depth<=self.max_depth:
            # find the best split
            best_split = self.get_best_split(dataset, num_samples, num_features)
            # check if information gain is positive
            if best_split["info_gain"]>0:
                # recur left
                left_subtree = self.build_tree(best_split["dataset_left"],
                curr_depth+1)
```

```
                # recur right
                right_subtree = self.build_tree(best_split["dataset_right"],
                curr_depth+1)
                # return decision node
                return Node(best_split["feature_index"], best_split["threshold"]
                left_subtree, right_subtree,
                best_split["info_gain"])

        # compute leaf node
        leaf_value = self.calculate_leaf_value(Y)
        # return leaf node
        return Node(value=leaf_value)

    def get_best_split(self, dataset, num_samples, num_features):
        ''' function to find the best split '''

        # dictionary to store the best split
        best_split = {}
        max_info_gain = -float("inf")

        # loop over all the features
        for feature_index in range(num_features):
            feature_values = dataset[:, feature_index]
            possible_thresholds = np.unique(feature_values)
            # loop over all the feature values present in the data
            for threshold in possible_thresholds:
                # get current split
                dataset_left, dataset_right = self.split(dataset,
                feature_index, threshold)
                # check if childs are not null
                if len(dataset_left)>0 and len(dataset_right)>0:
                    y, left_y, right_y = dataset[:, -1],
                    dataset_left[:, -1], dataset_right[:, -1]
                    # compute information gain
                    curr_info_gain = self.information_gain(y, left_y,
                    right_y, "gini")
                    # update the best split if needed
                    if curr_info_gain>max_info_gain:
                        best_split["feature_index"] = feature_index
                        best_split["threshold"] = threshold
                        best_split["dataset_left"] = dataset_left
                        best_split["dataset_right"] = dataset_right
                        best_split["info_gain"] = curr_info_gain
                        max_info_gain = curr_info_gain

        # return best split
```

```
        return best_split

    def split(self, dataset, feature_index, threshold):
        ''' function to split the data '''

        dataset_left = np.array([row for row in dataset if
        row[feature_index]<=threshold])
        dataset_right = np.array([row for row in dataset if
        row[feature_index]>threshold])
        return dataset_left, dataset_right

    def information_gain(self, parent, l_child, r_child, mode="entropy"):
        ''' function to compute information gain '''

        weight_l = len(l_child) / len(parent)
        weight_r = len(r_child) / len(parent)
        if mode=="gini":
            gain = self.gini_index(parent) - (weight_l*self.gini_index(l_child) +
            weight_r*self.gini_index(r_child))
        else:
            gain = self.entropy(parent) - (weight_l*self.entropy(l_child) +
            weight_r*self.entropy(r_child))
        return gain

    def entropy(self, y):
        ''' function to compute entropy '''

        class_labels = np.unique(y)
        entropy = 0
        for cls in class_labels:
            p_cls = len(y[y == cls]) / len(y)
            entropy += -p_cls * np.log2(p_cls)
        return entropy

    def gini_index(self, y):
        ''' function to compute gini index '''

        class_labels = np.unique(y)
        gini = 0
        for cls in class_labels:
            p_cls = len(y[y == cls]) / len(y)
            gini += p_cls**2
        return 1 - gini

    def calculate_leaf_value(self, Y):
        ''' function to compute leaf node '''
```

```
        Y = list(Y)
        return max(Y, key=Y.count)

    def print_tree(self, tree=None, indent=" "):
        ''' function to print the tree '''

        if not tree:
            tree = self.root

        if tree.value is not None:
            print(tree.value)

        else:
            print("X_"+str(tree.feature_index), "<=", tree.threshold,
            "?", tree.info_gain)
            print("%sleft:" % (indent), end="")
            self.print_tree(tree.left, indent + indent)
            print("%sright:" % (indent), end="")
            self.print_tree(tree.right, indent + indent)

    def fit(self, X, Y):
        ''' function to train the tree '''

        dataset = np.concatenate((X, Y), axis=1)
        self.root = self.build_tree(dataset)

    def predict(self, X):
        ''' function to predict new dataset '''

        preditions = [self.make_prediction(x, self.root) for x in X]
        return preditions

    def make_prediction(self, x, tree):
        ''' function to predict a single data point '''

        if tree.value!=None: return tree.value
        feature_val = x[tree.feature_index]
        if feature_val<=tree.threshold:
            return self.make_prediction(x, tree.left)
        else:
            return self.make_prediction(x, tree.right)

import numpy as np
from numpy.core.umath_tests import inner1d
from copy import deepcopy
```

```
class AdaBoostClassifier(object):
    '''
    Parameters
    ----------
    base_estimator: object
        The base model from which the boosted ensemble is built.
    n_estimators: integer, optional(default=50)
        The maximum number of estimators
    learning_rate: float, optional(default=1)
    algorithm: {'SAMME','SAMME.R'}, optional(default='SAMME.R')
        SAMME.R uses predicted probabilities to update wights,
        while SAMME uses class error rate
    random_state: int or None, optional(default=None)
    Attributes
    ----------
    estimators_: list of base estimators
    estimator_weights_: array of floats
        Weights for each base_estimator
    estimator_errors_: array of floats
        Classification error for each estimator in the boosted ensemble.
    Reference:
    1. [multi-adaboost](https://web.stanford.edu/~hastie/Papers/samme.pdf)
    2. [scikit-learn:weight_boosting](https://github.com/scikit-learn/
    scikit-learn/blob/51a765a/sklearn/ensemble/weight_boosting.py#L289)
    '''

    def __init__(self, *args, **kwargs):
        if kwargs and args:
            raise ValueError(
                '''AdaBoostClassifier can only be called with keyword
                   arguments for the following keywords: base_estimator
                   ,n_estimators,
                    learning_rate,algorithm,random_state''')
        allowed_keys = ['base_estimator', 'n_estimators', 'learning_rate',
        'algorithm', 'random_state']
        keywords_used = kwargs.keys()
        for keyword in keywords_used:
            if keyword not in allowed_keys:
                raise ValueError(keyword + ":  Wrong keyword used
                --- check spelling")

        n_estimators = 50
        learning_rate = 1
        algorithm = 'SAMME.R'
```

```
    random_state = None

    if kwargs and not args:
        if 'base_estimator' in kwargs:
            base_estimator = kwargs.pop('base_estimator')
        else:
            raise ValueError('''base_estimator can not be None''')
        if 'n_estimators' in kwargs: n_estimators =
        kwargs.pop('n_estimators')
        if 'learning_rate' in kwargs: learning_rate =
        kwargs.pop('learning_rate')
        if 'algorithm' in kwargs: algorithm =
        kwargs.pop('algorithm')
        if 'random_state' in kwargs: random_state =
        kwargs.pop('random_state')

    self.base_estimator_ = base_estimator
    self.n_estimators_ = n_estimators
    self.learning_rate_ = learning_rate
    self.algorithm_ = algorithm
    self.random_state_ = random_state
    self.estimators_ = list()
    self.estimator_weights_ = np.zeros(self.n_estimators_)
    self.estimator_errors_ = np.ones(self.n_estimators_)


def _samme_proba(self, estimator, n_classes, X):
    """Calculate algorithm 4, step 2, equation c) of Zhu et al [1].
    References
    ----------
    .. [1] J. Zhu, H. Zou, S. Rosset, T. Hastie, "Multi-class
    AdaBoost", 2009.
    """
    proba = estimator.predict_proba(X)

    # Displace zero probabilities so the log is defined.
    # Also fix negative elements which may occur with
    # negative sample weights.
    proba[proba < np.finfo(proba.dtype).eps] = np.finfo(proba.dtype).eps
    log_proba = np.log(proba)

    return (n_classes - 1) * (log_proba - (1. / n_classes)
                              * log_proba.sum(axis=1)[:, np.newaxis])


def fit(self, X, y):
```

```
        self.n_samples = X.shape[0]
        # There is hidden trouble for classes, here the classes will be sorted.
        # So in boost we have to ensure that the predict results have
        the same classes sort
        self.classes_ = np.array(sorted(list(set(y))))
        self.n_classes_ = len(self.classes_)
        for iboost in range(self.n_estimators_):
            if iboost == 0:
                sample_weight = np.ones(self.n_samples) / self.n_samples

            sample_weight, estimator_weight, estimator_error = self.boost(X, y,

            # early stop
            if estimator_error == None:
                break

            # append error and weight
            self.estimator_errors_[iboost] = estimator_error
            self.estimator_weights_[iboost] = estimator_weight

            if estimator_error <= 0:
                break

        return self


    def boost(self, X, y, sample_weight):
        if self.algorithm_ == 'SAMME':
            return self.discrete_boost(X, y, sample_weight)
        elif self.algorithm_ == 'SAMME.R':
            return self.real_boost(X, y, sample_weight)

    def real_boost(self, X, y, sample_weight):
        estimator = deepcopy(self.base_estimator_)
        if self.random_state_:
            estimator.set_params(random_state=1)

        estimator.fit(X, y, sample_weight=sample_weight)

        y_pred = estimator.predict(X)
        incorrect = y_pred != y
        estimator_error = np.dot(incorrect, sample_weight) /
        np.sum(sample_weight, axis=0)

        # if worse than random guess, stop boosting
        if estimator_error >= 1.0 - 1 / self.n_classes_:
```

```
            return None, None, None

        y_predict_proba = estimator.predict_proba(X)
        # repalce zero
        y_predict_proba[y_predict_proba <
        np.finfo(y_predict_proba.dtype).eps] =
        np.finfo(y_predict_proba.dtype).eps

        y_codes = np.array([-1. / (self.n_classes_ - 1), 1.])
        y_coding = y_codes.take(self.classes_ == y[:, np.newaxis])

        # for sample weight update
        intermediate_variable = (-1. * self.learning_rate_ *
        (((self.n_classes_ - 1) / self.n_classes_) *
        inner1d(y_coding, np.log(y_predict_proba))))
        #dot iterate for each row

        # update sample weight
        sample_weight *= np.exp(intermediate_variable)

        sample_weight_sum = np.sum(sample_weight, axis=0)
        if sample_weight_sum <= 0:
            return None, None, None

        # normalize sample weight
        sample_weight /= sample_weight_sum

        # append the estimator
        self.estimators_.append(estimator)

        return sample_weight, 1, estimator_error


    def discrete_boost(self, X, y, sample_weight):
        estimator = deepcopy(self.base_estimator_)
        if self.random_state_:
            estimator.set_params(random_state=1)

        estimator.fit(X, y, sample_weight=sample_weight)

        y_pred = estimator.predict(X)
        incorrect = y_pred != y
        estimator_error = np.dot(incorrect, sample_weight) /
        np.sum(sample_weight, axis=0)

        # if worse than random guess, stop boosting
```

```
        if estimator_error >= 1 - 1 / self.n_classes_:
            return None, None, None

        # update estimator_weight
        estimator_weight = self.learning_rate_ *
        np.log((1 - estimator_error) / estimator_error) +
        np.log(self.n_classes_ - 1)

        if estimator_weight <= 0:
            return None, None, None

        # update sample weight
        sample_weight *= np.exp(estimator_weight * incorrect)

        sample_weight_sum = np.sum(sample_weight, axis=0)
        if sample_weight_sum <= 0:
            return None, None, None

        # normalize sample weight
        sample_weight /= sample_weight_sum

        # append the estimator
        self.estimators_.append(estimator)

        return sample_weight, estimator_weight, estimator_error

    def predict(self, X):
        n_classes = self.n_classes_
        classes = self.classes_[:, np.newaxis]
        pred = None

        if self.algorithm_ == 'SAMME.R':
            # The weights are all 1. for SAMME.R
            pred = sum(self._samme_proba(estimator, n_classes, X)
            for estimator in self.estimators_)
        else:  # self.algorithm == "SAMME"
            pred = sum((estimator.predict(X) == classes).T * w
                       for estimator, w in zip(self.estimators_,
                                               self.estimator_weights_))

        pred /= self.estimator_weights_.sum()
        if n_classes == 2:
            pred[:, 0] *= -1
            pred = pred.sum(axis=1)
            return self.classes_.take(pred > 0, axis=0)
```

```
        return self.classes_.take(np.argmax(pred, axis=1), axis=0)


    def predict_proba(self, X):
        if self.algorithm_ == 'SAMME.R':
            # The weights are all 1. for SAMME.R
            proba = sum(self._samme_proba(estimator, self.n_classes_, X)
                        for estimator in self.estimators_)
        else:  # self.algorithm == "SAMME"
            proba = sum(estimator.predict_proba(X) * w
                        for estimator, w in zip(self.estimators_,
                                                self.estimator_weights_))

        proba /= self.estimator_weights_.sum()
        proba = np.exp((1. / (n_classes - 1)) * proba)
        normalizer = proba.sum(axis=1)[:, np.newaxis]
        normalizer[normalizer == 0.0] = 1.0
        proba /= normalizer

        return proba
```

4.3 Methodology

The method we used is using fit function in each of the class to train the model, then using predict method to get the prediction of the model.

(1)SVM

```
In [25]:
1 clf = MulticlassSVM(C=0.1, tol=0.01, max_iter=20, random_state=10, v
2 clf.fit(Xtrain, Ytrain)
3 print("Accuracy of SVM is ",round(clf.score(Xtrain, Ytrain),4))

iter 1 violation 1.0
iter 2 violation 0.9633
iter 3 violation 0.7859
iter 4 violation 0.6094
iter 5 violation 0.4767
iter 6 violation 0.3844
iter 7 violation 0.3189
iter 8 violation 0.2682
iter 9 violation 0.2304
iter 10 violation 0.2016
iter 11 violation 0.1776
iter 12 violation 0.1581
iter 13 violation 0.1418
iter 14 violation 0.1284
iter 15 violation 0.117
iter 16 violation 0.107
iter 17 violation 0.0988
iter 18 violation 0.0918
iter 19 violation 0.0855
iter 20 violation 0.0804
Accuracy of SVM is  0.8361
```

```
In [26]:
1 Ypredict = clf.predict(Xtest)
2 Ypredict

Out[26]: array([2., 5., 3., ..., 4., 3., 1.])
```

Here, the Ypredict is the prediction we get from the SVM model.

(2)Logistic Regression

```
In [41]:
1 LogRegression = MultiClassLogisticRegression()
2 LogRegression.train(Xtrain,Ytrain)

Out[41]: <__main__.MultiClassLogisticRegression at 0x7faba11d04c0>

In [42]:
1 Ypred = LogRegression.predict_classes(Xtest)
2 Ypred

Out[42]: array([2., 5., 3., ..., 4., 3., 1.])
```

Here, the Ypred is the prediction we get from the logistic regression model.

(3)Decision Tree with AdaBoost

```
1 # Fit the Adaboost model
2
3 Adaboost = AdaBoostClassifier(base_estimator = DecisionTreeClassifie
4                                learning_rate = 1, algorithm = 'SAM
5                                random_state = None)
6 Adaboost.fit(Xtrain,Ytrain)
```

```
In [49]:
1 Ypredict = Adaboost.predict(Xtest)
2 Ypredict

Out[49]: array([5., 5., 3., ..., 3., 1., 3.])
```

Here, the Ypredict is the prediction that we get from the decision tree with AdaBoost.

4.4 Model outcome/Performance metrics

I showed the confusion matrix of the testing data and get the accuracy of the testing dataset for each model.

4.4.1 SVM

```
print('----- Testing dataset validation -----')
print('')
print('The confusion matrix of testing dataset is:')
print(confusion_matrix(Ytest,Ypredict))
print('')
print("Accuracy of SVM for testing dataset is:",round(accuracy_score(Ytest,Ypredict),4))
```

```
----- Testing dataset validation -----

The confusion matrix of testing dataset is:
[[2953   11  273    9    1]
 [  27 2601   37   15  325]
 [ 324    1 2584  294   12]
 [  95   37  198 2530  148]
 [  35  408   58  260 2383]]

Accuracy of SVM for testing dataset is: 0.8356
```

From the SVM model output, we can find that the accuracy of our SVM model is 0.8356, which means we have 83.56% accuracy on predicting our testing dataset. This is a very high rate. This means most of our data are predicted correctly.

From the confusion matrix, we can find that when our model predicting class 1, which is Fist (with thumb out), 2953 / 3247 = 90.05% of the class 1 data are predicted correctly. This is a pretty good accuracy, which means that our model predicts very well on class 1 data. Also, we can find that 273 out of 3247 class 1 data are predicted as class 3. Most of the bad prediction of the class 1 data are predicted as class 3, which is Point1 (point with pointer finger).

When our model predicting class 2, which is Stop (hand flat), 2601 / 3005 = 86.56% of the class 1 data are predicted correctly. This is a pretty good accuracy, which means that our model predicts very well on class 2 data. Also, we can find that 325 out of 3005 class 2 data are predicted as class 5. Most of the bad prediction of the class 2 data are predicted as class 5, which is Grab (fingers curled as if to grab).

When our model predicting class 3, which is Point1 (point with pointer finger), 2584 / 3215 = 80.37% of the class 3 data are predicted correctly. This is a pretty good accuracy, which means that our model predicts very well on class 3 data. Also, we can find that 324 out of 3215 class 3 data are predicted as class 1. Most of the bad prediction of the class 3 data are predicted as class 1, which is Fist (with thumb out). Also, there are also 294 out of 3215 class 3 data are predicted as class 4. Some of the bad predictions of the class 3 data are predicted as class 4, which is Point 2 (point with pointer and middle finger).

When our model predicting class 4, which is Point 2 (point with pointer and middle fingers), 2530 / 3008 = 84.11% of the class 4 data are predicted correctly. This is a pretty good accuracy, which means that our model predicts very well on class 4 data. Also, we can find that 198 out of 3008 class 4 data are predicted as class 3. Most of the bad prediction of the class 4 data are predicted as class 3, which is Point1 (point with pointer finger).

When our model predicting class 5, which is Grab (fingers curled as if to grab), 2383 / 3144 = 75.80% of the class 5 data are predicted correctly. This is a pretty good accuracy, which means that our model predicts very well on class 5 data. Also, we can find that 408 out of 3144 class 5 data are predicted as class 2. Most of the bad prediction of the class 5 data are predicted as class 2, which is Stop (hand flat).

From the output, we can see that the overall accuracy and predictions are very good. However, class 5 and class 2 are always get confused to each other. Also, class 1 and class 3, class 5 and class 3 are also confused.

4.4.2 Logistics Regression

```
print('----- Testing dataset validation -----')
print('')
print('The confusion matrix of testing dataset is:')
print(confusion_matrix(Ytest,Ypred))
print('')
print("Accuracy of logistic regression for testing dataset is:",roun
```

```
----- Testing dataset validation -----

The confusion matrix of testing dataset is:
[[2938   12  291    4    2]
 [  28 2645   34   26  272]
 [ 273    1 2586  338   17]
 [  76   69  245 2475  143]
 [  30  318   56  288 2452]]

Accuracy of logistic regression for testing dataset is: 0.8385
```

From the Logistic Regression model output, we can find that the accuracy of our SVM model is 0.8385, which means we have 83.85% accuracy on predicting our testing dataset. This is a very high rate. This means most of our data are predicted correctly.

From the confusion matrix, we can find that when our model predicting class 1, which is Fist (with thumb out), 2938 / 3247 = 90.48% of the class 1 data are predicted correctly. This is a pretty good accuracy, which means that our model predicts very well on class 1 data. Also, we can find that 291 out of 3247 class 1 data are predicted as class 3. Most of the bad prediction of the class 1 data are predicted as class 3, which is Point1 (point with pointer finger).

When our model predicting class 2, which is Stop (hand flat), 2645 / 3005 = 88.02% of the class 1 data are predicted correctly. This is a pretty good accuracy, which means that our model predicts very well on class 2 data. Also, we can find that 272 out of 3005 class 2 data are predicted as class 5. Most of the bad prediction of the class 2 data are predicted as class 5, which is Grab (fingers curled as if to grab).

When our model predicting class 3, which is Point1 (point with pointer finger), 2586 / 3215 = 80.44% of the class 3 data are predicted correctly. This is a pretty good accuracy, which means that our model predicts very well on class 3 data. Also, we can find that 273 out of 3215 class 3 data are predicted

as class 1. Most of the bad prediction of the class 3 data are predicted as class 1, which is Fist (with thumb out). Also, there are also 338 out of 3215 class 3 data are predicted as class 4. Some of the bad predictions of the class 3 data are predicted as class 4, which is Point 2 (point with pointer and middle finger).

When our model predicting class 4, which is Point 2 (point with pointer and middle fingers), 2475 / 3008 = 82.28% of the class 4 data are predicted correctly. This is a pretty good accuracy, which means that our model predicts very well on class 4 data. Also, we can find that 245 out of 3008 class 4 data are predicted as class 3. Most of the bad prediction of the class 4 data are predicted as class 3, which is Point1 (point with pointer finger).

When our model predicting class 5, which is Grab (fingers curled as if to grab), 2383 / 3144 = 75.80% of the class 5 data are predicted correctly. This is a pretty good accuracy, which means that our model predicts very well on class 5 data. Also, we can find that 408 out of 3144 class 5 data are predicted as class 2. Most of the bad prediction of the class 5 data are predicted as class 2, which is Stop (hand flat).

From the output, we can see that the overall accuracy and predictions are very good. However, class 5 and class 2 are always get confused to each other. Also, class 1 and class 3, class 5 and class 3 are also confused.

4.4.3 Decision Tree with AdaBoost

```
1 print('----- Testing dataset validation -----')
2 print('')
3 print('The confusion matrix of testing dataset is:')
4 print(confusion_matrix(Ytest,Ypredict))
5 print('')
6 print("Accuracy of logistic regression for testing dataset is:",roun
```

```
----- Testing dataset validation -----

The confusion matrix of testing dataset is:
[[2886    4  349    0    8]
 [  94 1304   35   60 1512]
 [  82   13 2868  204   48]
 [  21   78  712 1955  242]
 [  87  487   52  163 2355]]

Accuracy of logistic regression for testing dataset is: 0.7278
```

From the Adaboost Decision Tree model output, we can find that the accuracy of our SVM model is 0.7278, which means we have 72.78% accuracy on predicting our testing dataset. This is not a very high rate. While most of our data are predicted correctly, some of our data not predicted correct compared to the other 2 models

From the confusion matrix, we can find that when our model predicting class 1, which is Fist (with thumb out), 2886 / 3247 = 88.89% of the class 1 data are predicted correctly. This is a pretty good accuracy, which means that our model predicts very well on class 1 data. Also, we can find that 349 out of 3247 class 1 data are predicted as class 3. Most of the bad prediction of the class 1

data are predicted as class 3, which is Point1 (point with pointer finger).

When our model predicting class 2, which is Stop (hand flat), 1304 / 3005 = 43.39% of the class 1 data are predicted correctly. This is a low accuracy, which means that our model predicts very bad on class 2 data. Our model cannot predict even 50% of the class 2 data correctly. Also, we can find that 1512 out of 3005 class 2 data are predicted as class 5. Most of the bad prediction of the class 2 data are predicted as class 5, which is Grab (fingers curled as if to grab).

When our model predicting class 3, which is Point1 (point with pointer finger), 2868 / 3215 = 89.21% of the class 3 data are predicted correctly. This is a pretty good accuracy, which means that our model predicts very well on class 3 data.

When our model predicting class 4, which is Point 2 (point with pointer and middle fingers), 1955 / 3008 = 64.99% of the class 4 data are predicted correctly. This is a low accuracy, which means that our model predicts very bad on class 4 data. Also, we can find that 721 out of 3008 class 4 data are predicted as class 3. Most of the bad prediction of the class 4 data are predicted as class 3, which is Point1 (point with pointer finger).

When our model predicting class 5, which is Grab (fingers curled as if to grab), 2355 / 3144 = 74.90% of the class 5 data are predicted correctly. This is a pretty good accuracy, which means that our model predicts very well on class 5 data. Also, we can find that 487 out of 3144 class 5 data are predicted as class 2. Most of the bad prediction of the class 5 data are predicted as class 2, which is Stop (hand flat).

From the output, we can see that the overall accuracy and predictions are very good. However, class 5 and class 2 are always get confused to each other. Also, class 1 and class 3, class 5 and class 3 are also confused.

5 Conclusion

Since we don't need to let the model be readable, I will not consider decision tree as my optimal model because it has the lowest accuracy compared to the other 2 models. Also, we don't need to read the model manually. We just need to get the final result of the model. So decision tree is not more helpful compared with other models.

By comparing the SVM and the logistic regression models, we can find that for class 1, the SVM model predicts more accurate compared to the logistic regression model. Also, the logistic regression model gets more wrong predictions on class 3 when predicting class 1.

For class 2, logistic regression model predicts much more accurate compared to the SVM model. Compared with the logistic regression model, the SVM model gets more wrong predictions (1.18 times more bad predictions than logistic regression model) on class 5 when predicting class 2.

For class 3, logistic regression model predicts a little bit more accurate compared to the SVM model. However, the logistic regression model has more

wrong predictions on class 4 when predicting class 3, while the SVM model has more wrong predictions on class 1 when predicting class 3.

For class 4, SVM model predicts more accurate compared to the logistic regression model. Compared with SVM model, the logistic regression model gets more wrong predictions on class 3 when predicting class 4.

For class 5, logistic regression model predicts much more accurate compared to the SVM model. Compared with the logistic regression model, the SVM model gets more wrong predictions (1.28 times more bad predictions than logistic regression model) on class 2 when predicting class 5.

Therefore, to conclude, the logistic regression model has a little higher accuracy (logistic regression model accuracy is 0.8385, while SVM model accuracy is 0.8356). However, the SVM model predicts more accurate when predicting class 1 and class 4 values. And SVM model predicts badly when distinguish the class 2 and class 5 values. The logistic regression model doing more accurate on predicting class 2, class 5 and class 3.

In the case that SVM model running time is roughly equal to the logistic regression model running time, we could select the optimal model based on the accuracy and the model performance. For example, if you want the model do much more accurate when distinguish the class 2 and class 5, we could choose logistic regression as the optimal model. If you want the model do much more accurate when predict the class 1 and class 3, we could choose SVM as the optimal model.

6 Reference

Motion Capture Hand Postures Data Set. (2017). UC Irvine Machine Learning Repository. https://archive.ics.uci.edu/ml/datasets/Motion+Capture+Hand+Postures

IE 7300 Project Xinan Wang

```
# import the required packages

%matplotlib inline
import pandas as pd
import numpy as np
import itertools
import time
import statsmodels.api as sm
import matplotlib.pyplot as plt
import seaborn as sns
```

```
df = pd.read_csv('Postures.csv')
df
```

	Class	User	X0	Y0	Z0	X1	Y1	Z1	X2	Y2	...	Z8
0	1	0	54.263880	71.466776	-64.807709	76.895635	42.462500	-72.780545	36.621229	81.680557	...	?
1	1	0	56.527558	72.266609	-61.935252	39.135978	82.538530	-49.596509	79.223743	43.254091	...	?
2	1	0	55.849928	72.469064	-62.562788	37.988804	82.631347	-50.606259	78.451526	43.567403	...	?
3	1	0	55.329647	71.707275	-63.688956	36.561863	81.868749	-52.752784	86.320630	68.214645	...	?
4	1	0	55.142401	71.435607	-64.177303	36.175818	81.556874	-53.475747	76.986143	42.426849	...	?
...	...	...	...	...	...	...	...	...	...	...	...	...
78090	5	14	54.251127	129.177414	-44.252511	27.720784	107.810661	11.099282	-1.270139	122.758679	...	-6.54311453354464
78091	5	14	54.334883	129.253842	-44.016320	27.767911	107.914808	11.069842	-30.334054	77.858214	...	-62.1305625712145
78092	5	14	54.151540	129.269502	-44.173273	27.725978	108.034006	11.020347	-22.574718	104.222208	...	3.83590370588782
78093	5	14	27.915311	108.007390	10.814957	-0.910435	122.464093	-47.271248	-30.084588	77.705861	...	-63.2216259324485
78094	5	14	27.898705	108.092877	11.107857	-30.031402	77.740235	-17.453099	-1.091566	122.827638	...	-63.0265744155222

78095 rows × 38 columns

EDA

```
df = df.replace('?',0).astype(float)
df
```

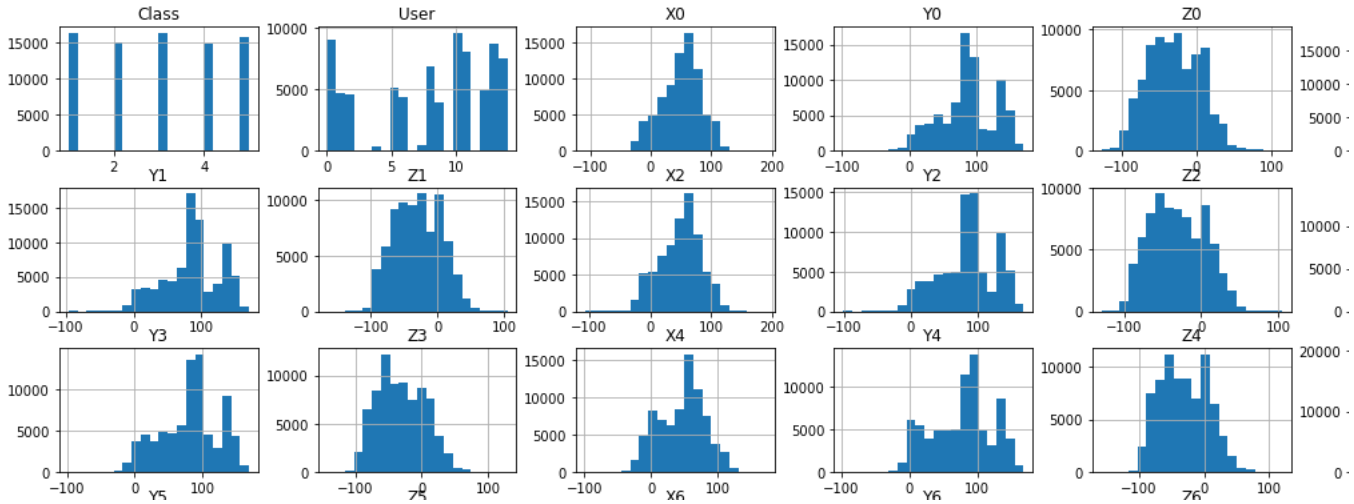
	Class	User	X0	Y0	Z0	X1	Y1	Z1	X2	Y2	...	Z8	
0	1.0	0.0	54.263880	71.466776	-64.807709	76.895635	42.462500	-72.780545	36.621229	81.680557	...	0.000000	0.000
1	1.0	0.0	56.527558	72.266609	-61.935252	39.135978	82.538530	-49.596509	79.223743	43.254091	...	0.000000	0.000
2	1.0	0.0	55.849928	72.469064	-62.562788	37.988804	82.631347	-50.606259	78.451526	43.567403	...	0.000000	0.000
3	1.0	0.0	55.329647	71.707275	-63.688956	36.561863	81.868749	-52.752784	86.320630	68.214645	...	0.000000	0.000
4	1.0	0.0	55.142401	71.435607	-64.177303	36.175818	81.556874	-53.475747	76.986143	42.426849	...	0.000000	0.000
...	...	...	...	...	...	...	...	...	...	...	...	...	
78090	5.0	14.0	54.251127	129.177414	-44.252511	27.720784	107.810661	11.099282	-1.270139	122.758679	...	-6.543115	87.733
78091	5.0	14.0	54.334883	129.253842	-44.016320	27.767911	107.914808	11.069842	-30.334054	77.858214	...	-62.130563	78.229
78092	5.0	14.0	54.151540	129.269502	-44.173273	27.725978	108.034006	11.020347	-22.574718	104.222208	...	3.835904	78.591
78093	5.0	14.0	27.915311	108.007390	10.814957	-0.910435	122.464093	-47.271248	-30.084588	77.705861	...	-63.221626	0.000
78094	5.0	14.0	27.898705	108.092877	11.107857	-30.031402	77.740235	-17.453099	-1.091566	122.827638	...	-63.026574	78.879

78095 rows × 38 columns

```
df.describe().transpose()
```

	count	mean	std	min	25%	50%	75%	max
Class	78095.0	2.983776	1.421152	1.000000	2.000000	3.000000	4.000000	5.000000
User	78095.0	7.959229	4.697754	0.000000	5.000000	9.000000	12.000000	14.000000
X0	78095.0	50.346308	32.695886	-108.552738	29.295141	54.620245	72.488833	190.017835
Y0	78095.0	85.813150	40.203448	-98.233756	63.497746	86.526334	113.108673	169.175464
Z0	78095.0	-29.985096	34.361971	-126.770872	-56.356593	-30.864248	-1.419462	113.345119
X1	78095.0	49.595844	32.477961	-111.685241	28.755679	54.215714	71.763080	188.691997
Y1	78095.0	86.193751	40.452297	-96.142589	64.156450	87.543493	116.228881	170.209350
Z1	78095.0	-29.509579	34.764460	-166.006838	-57.360408	-30.185331	-0.368080	104.697852
X2	78095.0	48.612744	33.605155	-106.886524	25.173405	53.814592	71.561988	188.760168
Y2	78095.0	83.772387	41.022710	-100.789312	58.053733	86.459935	106.661720	168.186466
Z2	78095.0	-30.560906	35.120384	-129.595296	-58.654339	-32.356535	-0.946134	104.590879
X3	78095.0	48.064123	34.027102	-111.761053	22.749246	53.888390	71.216793	151.033472
Y3	78095.0	81.339253	42.093648	-97.603414	53.094602	85.477367	105.169964	168.292018
Z3	78095.0	-30.871450	35.892811	-143.540529	-59.223756	-33.440705	-0.554393	129.316870
X4	78095.0	46.472199	34.849901	-99.107635	16.989314	52.670592	71.036529	172.275978
Y4	78095.0	77.206639	44.634164	-97.948829	43.857348	84.372784	104.299414	168.258643
Z4	78095.0	-30.715371	36.210018	-157.199089	-59.733553	-32.521378	0.000000	119.237203
X5	78095.0	39.197998	36.296891	-120.657868	0.000000	43.772063	67.224900	180.563322
Y5	78095.0	67.821645	49.513061	-97.468548	17.838554	79.001681	101.803483	167.926171
Z5	78095.0	-25.222531	35.482206	-135.699430	-54.402155	-20.158252	0.000000	110.898899
X6	78095.0	30.559226	36.682674	-100.084275	0.000000	15.798763	61.142447	176.409004
Y6	78095.0	56.018467	52.886406	-67.283707	0.000000	52.723120	99.126611	168.598384
Z6	78095.0	-17.822143	31.784525	-153.449813	-40.880986	0.000000	0.000000	117.914907
X7	78095.0	22.172219	35.053134	-108.605639	0.000000	0.000000	49.409962	189.221529
Y7	78095.0	44.113298	52.673598	-64.972157	0.000000	0.000000	93.725568	169.127359
Z7	78095.0	-10.157104	26.043592	-113.733105	-19.296698	0.000000	0.000000	117.815967
X8	78095.0	18.867068	33.717126	-121.182089	0.000000	0.000000	35.523460	173.906643
Y8	78095.0	33.668643	49.403134	-65.077550	0.000000	0.000000	78.198374	169.322843
Z8	78095.0	-9.535343	25.321953	-142.654497	-7.750354	0.000000	0.000000	119.213101
X9	78095.0	16.802219	33.583912	-99.231688	0.000000	0.000000	12.497466	174.054403
Y9	78095.0	24.719138	44.032514	-64.734284	0.000000	0.000000	37.681938	167.942588
Z9	78095.0	-8.524955	23.820691	-113.397327	0.000000	0.000000	0.000000	123.380512
X10	78095.0	10.154914	26.873653	-80.196289	0.000000	0.000000	0.000000	149.486224
Y10	78095.0	13.979146	34.451890	-65.019295	0.000000	0.000000	0.000000	168.352478
Z10	78095.0	-5.617450	20.805720	-112.668930	0.000000	0.000000	0.000000	108.455548

```
df.hist(figsize=(20,15),bins=20)
plt.show()
```




```
# Get the data information

df.info()
```

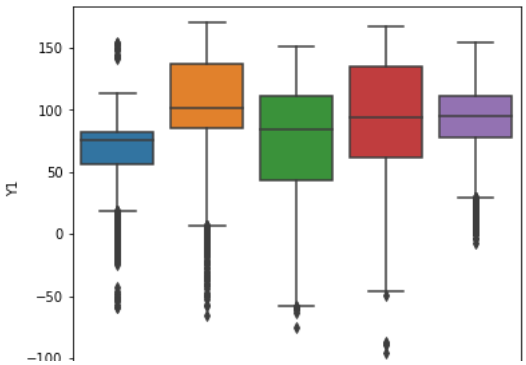
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 78095 entries, 0 to 78094
Data columns (total 38 columns):
 #   Column  Non-Null Count  Dtype  
---  ------  --------------  -----  
 0   Class   78095 non-null  float64
 1   User    78095 non-null  float64
 2   X0      78095 non-null  float64
 3   Y0      78095 non-null  float64
 4   Z0      78095 non-null  float64
 5   X1      78095 non-null  float64
 6   Y1      78095 non-null  float64
 7   Z1      78095 non-null  float64
 8   X2      78095 non-null  float64
 9   Y2      78095 non-null  float64
 10  Z2      78095 non-null  float64
 11  X3      78095 non-null  float64
 12  Y3      78095 non-null  float64
 13  Z3      78095 non-null  float64
 14  X4      78095 non-null  float64
 15  Y4      78095 non-null  float64
 16  Z4      78095 non-null  float64
 17  X5      78095 non-null  float64
 18  Y5      78095 non-null  float64
 19  Z5      78095 non-null  float64
 20  X6      78095 non-null  float64
 21  Y6      78095 non-null  float64
 22  Z6      78095 non-null  float64
 23  X7      78095 non-null  float64
 24  Y7      78095 non-null  float64
 25  Z7      78095 non-null  float64
 26  X8      78095 non-null  float64
 27  Y8      78095 non-null  float64
 28  Z8      78095 non-null  float64
 29  X9      78095 non-null  float64
 30  Y9      78095 non-null  float64
 31  Z9      78095 non-null  float64
 32  X10     78095 non-null  float64
 33  Y10     78095 non-null  float64
 34  Z10     78095 non-null  float64
 35  X11     78095 non-null  float64
 36  Y11     78095 non-null  float64
 37  Z11     78095 non-null  float64
dtypes: float64(38)
memory usage: 22.6 MB
```

```
df['Class'].value_counts()
```

```
3.0    16344
1.0    16265
5.0    15733
2.0    14978
4.0    14775
Name: Class, dtype: int64
```

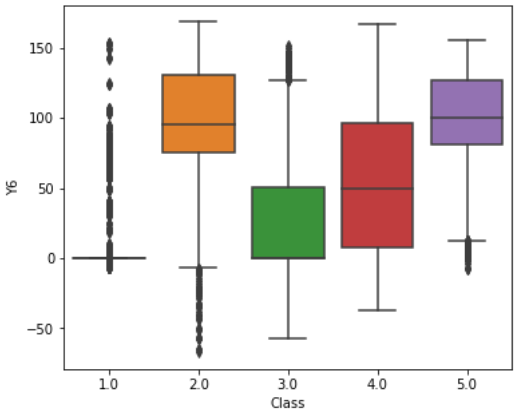
```
plt.figure(figsize=(6,5))
sns.boxplot(data=df,x='Class',y='Y1')
```

```
<AxesSubplot:xlabel='Class', ylabel='Y1'>
```



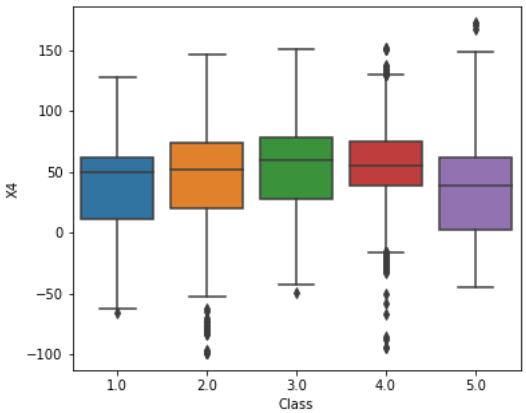

```
plt.figure(figsize=(6,5))
sns.boxplot(data=df,x='Class',y='Y6')
```

```
<AxesSubplot:xlabel='Class', ylabel='Y6'>
```



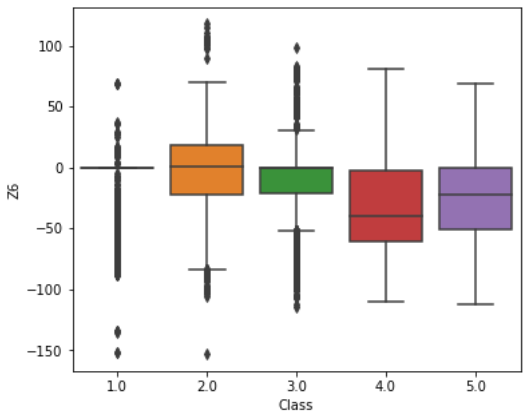

```
plt.figure(figsize=(6,5))
sns.boxplot(data=df,x='Class',y='X4')
```

```
<AxesSubplot:xlabel='Class', ylabel='X4'>
```




```
plt.figure(figsize=(6,5))
sns.boxplot(data=df,x='Class',y='Z6')
```

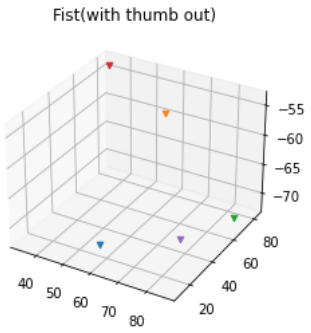
```
<AxesSubplot:xlabel='Class', ylabel='Z6'>
```



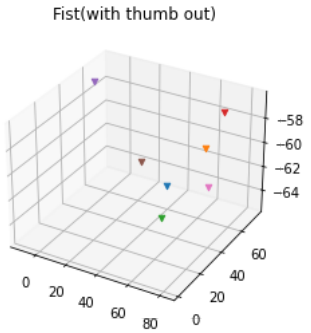
```
# We need a function to plot the points

def pltHand(handPoints):
    plt.close('all')
    fig = plt.figure()
    ax = fig.add_subplot(projection='3d')
    for i in range(11):
        pntx = f'X{i}'
        pnty = f'Y{i}'
        pntz = f'Z{i}'
        if(handPoints[pntx].values[0] == 0 or
            handPoints[pnty].values[0] == 0 or
            handPoints[pntz].values[0] == 0):
            n = 0;
        else:
            xlocation = handPoints[pntx]
            ylocation = handPoints[pnty]
            zlocation = handPoints[pntz]
            ax.scatter(xlocation, ylocation, zlocation, marker='v')
    crntClass = handPoints['Class'].values[0]
    if (crntClass == 1):
        title = 'Fist(with thumb out)'
    if(crntClass == 2):
        title = 'Stop(hand flat)'
    if (crntClass == 3):
        title = 'Point1(point with pointer finger)'
    if (crntClass == 4):
        title = 'Point2(point with pointer and middle fingers)'
    if (crntClass == 5):
        title = 'Grab(fingers curled as if to grab).'
    plt.title(title)


w = np.random.randint(df.shape[1], size=1)[0]
pltHand(df[w:w+1])
```




```
w = w = np.random.randint(df.shape[0], size=1)[0]
pltHand(df[w:w+1])
```

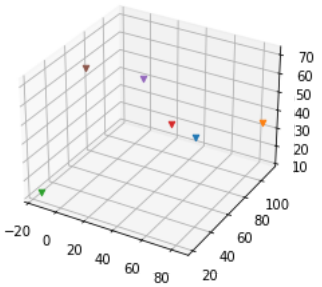



```
w = np.random.randint(df.shape[0], size=2)[0]
pltHand(df[w:w+1])
```

Point2(point with pointer and middle fingers)

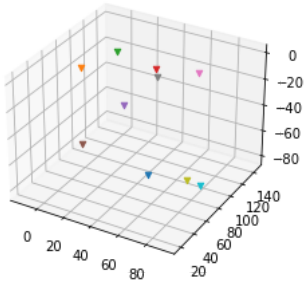
```
w = np.random.randint(df.shape[0], size=6)[0]
pltHand(df[w:w+1])
```

Point1(point with pointer finger)




```
w = np.random.randint(df.shape[0], size=1)[0]
pltHand(df[w:w+1])
```

Grab(fingers curled as if to grab).




```
df
```

	Class	User	X0	Y0	Z0	X1	Y1	Z1	X2	Y2	...	Z8	
0	1.0	0.0	54.263880	71.466776	-64.807709	76.895635	42.462500	-72.780545	36.621229	81.680557	...	0.000000	0.000
1	1.0	0.0	56.527558	72.266609	-61.935252	39.135978	82.538530	-49.596509	79.223743	43.254091	...	0.000000	0.000
2	1.0	0.0	55.849928	72.469064	-62.562788	37.988804	82.631347	-50.606259	78.451526	43.567403	...	0.000000	0.000
3	1.0	0.0	55.329647	71.707275	-63.688956	36.561863	81.868749	-52.752784	86.320630	68.214645	...	0.000000	0.000
4	1.0	0.0	55.142401	71.435607	-64.177303	36.175818	81.556874	-53.475747	76.986143	42.426849	...	0.000000	0.000
...	...	...	...	...	...	...	...	...	...	...	...	...	
78090	5.0	14.0	54.251127	129.177414	-44.252511	27.720784	107.810661	11.099282	-1.270139	122.758679	...	-6.543115	87.733
78091	5.0	14.0	54.334883	129.253842	-44.016320	27.767911	107.914808	11.069842	-30.334054	77.858214	...	-62.130563	78.229
78092	5.0	14.0	54.151540	129.269502	-44.173273	27.725978	108.034006	11.020347	-22.574718	104.222208	...	3.835904	78.591
78093	5.0	14.0	27.915311	108.007390	10.814957	-0.910435	122.464093	-47.271248	-30.084588	77.705861	...	-63.221626	0.000
78094	5.0	14.0	27.898705	108.092877	11.107857	-30.031402	77.740235	-17.453099	-1.091566	122.827638	...	-63.026574	78.879

78095 rows × 38 columns

```
X = df.iloc[:,1:]
Y = df.iloc[:,0]
```

```
0         1.0
1         1.0
2         1.0
3         1.0
4         1.0
         ...
78090     5.0
78091     5.0
78092     5.0
78093     5.0
78094     5.0
Name: Class, Length: 78095, dtype: float64
```

```
# Scale the dataset
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X = sc.fit_transform(X)


# Split the dataset into training and testing
from sklearn.model_selection import train_test_split
Xtrain, Xtest, Ytrain, Ytest = train_test_split( X, Y, test_size = 0.2,random_state=42)
```

SVM

```
# Create SVM model

from sklearn.base import BaseEstimator, ClassifierMixin
from sklearn.utils import check_random_state
from sklearn.preprocessing import LabelEncoder


def projection_simplex(v, z=1):
    """
    Projection onto the simplex:
        w^* = argmin_w 0.5 ||w-v||^2 s.t. \sum_i w_i = z, w_i >= 0
    """
    # For other algorithms computing the same projection, see
    # https://gist.github.com/mblondel/6f3b7aaad90606b98f71
    n_features = v.shape[0]
    u = np.sort(v)[::-1]
    cssv = np.cumsum(u) - z
    ind = np.arange(n_features) + 1
    cond = u - cssv / ind > 0
    rho = ind[cond][-1]
    theta = cssv[cond][-1] / float(rho)
    w = np.maximum(v - theta, 0)
    return w


class MulticlassSVM(BaseEstimator, ClassifierMixin):

    def __init__(self, C=1, max_iter=50, tol=0.05,
                 random_state=None, verbose=0):
        self.C = C
        self.max_iter = max_iter
        self.tol = tol,
        self.random_state = random_state
        self.verbose = verbose

    def _partial_gradient(self, X, y, i):
        # Partial gradient for the ith sample.
        g = np.dot(X[i], self.coef_.T) + 1
        g[y[i]] -= 1
        return g

    def _violation(self, g, y, i):
        # Optimality violation for the ith sample.
        smallest = np.inf
        for k in range(g.shape[0]):
            if k == y[i] and self.dual_coef_[k, i] >= self.C:
                continue
            elif k != y[i] and self.dual_coef_[k, i] >= 0:
                continue

            smallest = min(smallest, g[k])

        return g.max() - smallest

    def _solve_subproblem(self, g, y, norms, i):
        # Prepare inputs to the projection.
        Ci = np.zeros(g.shape[0])
        Ci[y[i]] = self.C
        beta_hat = norms[i] * (Ci - self.dual_coef_[:, i]) + g / norms[i]
        z = self.C * norms[i]

        # Compute projection onto the simplex.
        beta = projection_simplex(beta_hat, z)

        return Ci - self.dual_coef_[:, i] - beta / norms[i]

    def fit(self, X, y):
        n_samples, n_features = X.shape
```

```
        # Normalize labels.
        self._label_encoder = LabelEncoder()
        y = self._label_encoder.fit_transform(y)

        # Initialize primal and dual coefficients.
        n_classes = len(self._label_encoder.classes_)
        self.dual_coef_ = np.zeros((n_classes, n_samples), dtype=np.float64)
        self.coef_ = np.zeros((n_classes, n_features))

        # Pre-compute norms.
        norms = np.sqrt(np.sum(X ** 2, axis=1))

        # Shuffle sample indices.
        rs = check_random_state(self.random_state)
        ind = np.arange(n_samples)
        rs.shuffle(ind)

        violation_init = None
        for it in range(self.max_iter):
            violation_sum = 0

            for ii in range(n_samples):
                i = ind[ii]

                # All-zero samples can be safely ignored.
                if norms[i] == 0:
                    continue

                g = self._partial_gradient(X, y, i)
                v = self._violation(g, y, i)
                violation_sum += v

                if v < 1e-12:
                    continue

                # Solve subproblem for the ith sample.
                delta = self._solve_subproblem(g, y, norms, i)

                # Update primal and dual coefficients.
                self.coef_ += (delta * X[i][:, np.newaxis]).T
                self.dual_coef_[:, i] += delta

            if it == 0:
                violation_init = violation_sum

            vratio = violation_sum / violation_init

            if self.verbose >= 1:
                print("iter", it + 1, "violation", round(vratio,4))

            if vratio < self.tol:
                if self.verbose >= 1:
                    print("Converged")
                break

        return self

    def predict(self, X):
        decision = np.dot(X, self.coef_.T)
        pred = decision.argmax(axis=1)
        return self._label_encoder.inverse_transform(pred)


clf = MulticlassSVM(C=0.1, tol=0.01, max_iter=20, random_state=10, verbose=1)
clf.fit(Xtrain, Ytrain)
print("Accuracy of SVM is ",round(clf.score(Xtrain, Ytrain),4))
```

```
iter 1 violation 1.0
iter 2 violation 0.9633
iter 3 violation 0.7859
iter 4 violation 0.6094
iter 5 violation 0.4767
iter 6 violation 0.3844
iter 7 violation 0.3189
iter 8 violation 0.2682
iter 9 violation 0.2304
iter 10 violation 0.2016
iter 11 violation 0.1776
iter 12 violation 0.1581
iter 13 violation 0.1418
iter 14 violation 0.1284
iter 15 violation 0.117
iter 16 violation 0.107
iter 17 violation 0.0988
```

```
iter 18 violation 0.0918
iter 19 violation 0.0855
iter 20 violation 0.0804
Accuracy of SVM is  0.8361
```

```
Ypredict = clf.predict(Xtest)
Ypredict
```

```
array([2., 5., 3., ..., 4., 3., 1.])
```

```
from sklearn.metrics import confusion_matrix
from sklearn.metrics import accuracy_score


print('----- Testing dataset validation -----')
print('')
print('The confusion matrix of testing dataset is:')
print(confusion_matrix(Ytest,Ypredict))
print('')
print("Accuracy of SVM for testing dataset is:",round(accuracy_score(Ytest,Ypredict),4))
```

```
----- Testing dataset validation -----

The confusion matrix of testing dataset is:
[[2953   11  273    9    1]
 [  27 2601   37   15  325]
 [ 324    1 2584  294   12]
 [  95   37  198 2530  148]
 [  35  408   58  260 2383]]

Accuracy of SVM for testing dataset is: 0.8356
```

```
class MultiClassLogisticRegression:
    """
    Multiclass logistic regression
    """

    def __init__(self, epochs = 10000, threshold=1e-3):
        """
        Constructor for multiclass regression

        Args:
            epochs (int, optional): No of iteration Defaults to 10000.
            threshold (_type_, optional): Each iteration thresold. Defaults to 1e-3.
        """
        self.epochs = epochs
        self.threshold = threshold

    def train(self, X, y, batch_size=64, lr=0.001, rand_seed=4, verbose=False):
        """
        Train the model

        Args:
            X (_type_): Features
            y (_type_): Labels
            batch_size (int, optional): Batch size per iterations. Defaults to 64.
            lr (float, optional): Learning rate. Defaults to 0.001.
            rand_seed (int, optional): _description_. Defaults to 4.
            verbose (bool, optional): _description_. Defaults to False.

        Returns:
            _type_: return the instance
        """
        np.random.seed(rand_seed)
        self.classes = np.unique(y)
        self.class_labels = {c:i for i,c in enumerate(self.classes)}
        X = self.add_bias(X)
        y = self.one_hot(y)
        self.loss = []
        self.weights = np.zeros(shape=(len(self.classes),X.shape[1]))
        self.fit_data(X, y, batch_size, lr, verbose)
        return self

    def fit_data(self, X, y, batch_size, lr, verbose):
        i = 0
        while (not self.epochs or i < self.epochs):
            self.loss.append(self.cross_entropy(y, self.predict_(X)))
            idx = np.random.choice(X.shape[0], batch_size)
            X_batch, y_batch = X[idx], y[idx]
            error = y_batch - self.predict_(X_batch)
            update = (lr * np.dot(error.T, X_batch))
            self.weights += update
```

```
            if np.abs(update).max() < self.threshold:
                break
            if i % 1000 == 0 and verbose:
                print(' Training Accuray at {} iterations is {}'.format(i, self.evaluate_(X, y)))
            i +=1

    def predict(self, X):
        return self.predict_(self.add_bias(X))

    def predict_(self, X):
        pre_vals = np.dot(X, self.weights.T).reshape(-1,len(self.classes))
        return self.softmax(pre_vals)

    def getLoss(self):
        return self.loss;

    def softmax(self, z):
        return np.exp(z) / np.sum(np.exp(z), axis=1).reshape(-1,1)

    def predict_classes(self, X):
        self.probs_ = self.predict(X)
        return np.vectorize(lambda c: self.classes[c])(np.argmax(self.probs_, axis=1))

    def add_bias(self,X):
        return np.insert(X, 0, 1, axis=1)

    def get_random_weights(self, row, col):
        return np.zeros(shape=(row,col))

    def one_hot(self, y):
        return np.eye(len(self.classes))[np.vectorize(lambda c: self.class_labels[c])(y).reshape(-1)]

    def score(self, X, y):
        return round(np.mean(self.predict_classes(X) == y),3)

    def evaluate_(self, X, y):
        return np.mean(np.argmax(self.predict_(X), axis=1) == np.argmax(y, axis=1))

    def cross_entropy(self, y, probs):
        return -1 * np.mean(y * np.log(probs))

LogRegression = MultiClassLogisticRegression()
LogRegression.train(Xtrain,Ytrain)
```

```
    <__main__.MultiClassLogisticRegression at 0x7faba11d04c0>
```

```
Ypred = LogRegression.predict_classes(Xtest)
Ypred
```

```
    array([2., 5., 3., ..., 4., 3., 1.])
```

```
print('----- Testing dataset validation -----')
print('')
print('The confusion matrix of testing dataset is:')
print(confusion_matrix(Ytest,Ypred))
print('')
print("Accuracy of logistic regression for testing dataset is:",round(accuracy_score(Ytest,Ypred),4))
```

```
    ----- Testing dataset validation -----

    The confusion matrix of testing dataset is:
    [[2938   12  291    4    2]
     [  28 2645   34   26  272]
     [ 273    1 2586  338   17]
     [  76   69  245 2475  143]
     [  30  318   56  288 2452]]

    Accuracy of logistic regression for testing dataset is: 0.8385
```

```
class Node():
    def __init__(self, feature_index=None, threshold=None, left=None, right=None, info_gain=None, value=None):
        ''' constructor '''

        # for decision node
        self.feature_index = feature_index
        self.threshold = threshold
        self.left = left
        self.right = right
        self.info_gain = info_gain
```

```
        # for leaf node
        self.value = value

class DecisionTreeClassifier():
    def __init__(self, min_samples_split=2, max_depth=2):
        ''' constructor '''

        # initialize the root of the tree
        self.root = None

        # stopping conditions
        self.min_samples_split = min_samples_split
        self.max_depth = max_depth

    def build_tree(self, dataset, curr_depth=0):
        ''' recursive function to build the tree '''

        X, Y = dataset[:,:-1], dataset[:,-1]
        num_samples, num_features = np.shape(X)

        # split until stopping conditions are met
        if num_samples>=self.min_samples_split and curr_depth<=self.max_depth:
            # find the best split
            best_split = self.get_best_split(dataset, num_samples, num_features)
            # check if information gain is positive
            if best_split["info_gain"]>0:
                # recur left
                left_subtree = self.build_tree(best_split["dataset_left"], curr_depth+1)
                # recur right
                right_subtree = self.build_tree(best_split["dataset_right"], curr_depth+1)
                # return decision node
                return Node(best_split["feature_index"], best_split["threshold"],
                            left_subtree, right_subtree, best_split["info_gain"])

        # compute leaf node
        leaf_value = self.calculate_leaf_value(Y)
        # return leaf node
        return Node(value=leaf_value)

    def get_best_split(self, dataset, num_samples, num_features):
        ''' function to find the best split '''

        # dictionary to store the best split
        best_split = {}
        max_info_gain = -float("inf")

        # loop over all the features
        for feature_index in range(num_features):
            feature_values = dataset[:, feature_index]
            possible_thresholds = np.unique(feature_values)
            # loop over all the feature values present in the data
            for threshold in possible_thresholds:
                # get current split
                dataset_left, dataset_right = self.split(dataset, feature_index, threshold)
                # check if childs are not null
                if len(dataset_left)>0 and len(dataset_right)>0:
                    y, left_y, right_y = dataset[:, -1], dataset_left[:, -1], dataset_right[:, -1]
                    # compute information gain
                    curr_info_gain = self.information_gain(y, left_y, right_y, "gini")
                    # update the best split if needed
                    if curr_info_gain>max_info_gain:
                        best_split["feature_index"] = feature_index
                        best_split["threshold"] = threshold
                        best_split["dataset_left"] = dataset_left
                        best_split["dataset_right"] = dataset_right
                        best_split["info_gain"] = curr_info_gain
                        max_info_gain = curr_info_gain

        # return best split
        return best_split

    def split(self, dataset, feature_index, threshold):
        ''' function to split the data '''

        dataset_left = np.array([row for row in dataset if row[feature_index]<=threshold])
        dataset_right = np.array([row for row in dataset if row[feature_index]>threshold])
        return dataset_left, dataset_right

    def information_gain(self, parent, l_child, r_child, mode="entropy"):
        ''' function to compute information gain '''

        weight_l = len(l_child) / len(parent)
        weight_r = len(r_child) / len(parent)
```

```
        if mode=="gini":
            gain = self.gini_index(parent) - (weight_l*self.gini_index(l_child) + weight_r*self.gini_index(r_child))
        else:
            gain = self.entropy(parent) - (weight_l*self.entropy(l_child) + weight_r*self.entropy(r_child))
        return gain

    def entropy(self, y):
        ''' function to compute entropy '''

        class_labels = np.unique(y)
        entropy = 0
        for cls in class_labels:
            p_cls = len(y[y == cls]) / len(y)
            entropy += -p_cls * np.log2(p_cls)
        return entropy

    def gini_index(self, y):
        ''' function to compute gini index '''

        class_labels = np.unique(y)
        gini = 0
        for cls in class_labels:
            p_cls = len(y[y == cls]) / len(y)
            gini += p_cls**2
        return 1 - gini

    def calculate_leaf_value(self, Y):
        ''' function to compute leaf node '''

        Y = list(Y)
        return max(Y, key=Y.count)

    def print_tree(self, tree=None, indent=" "):
        ''' function to print the tree '''

        if not tree:
            tree = self.root

        if tree.value is not None:
            print(tree.value)

        else:
            print("X_"+str(tree.feature_index), "<=", tree.threshold, "?", tree.info_gain)
            print("%sleft:" % (indent), end="")
            self.print_tree(tree.left, indent + indent)
            print("%sright:" % (indent), end="")
            self.print_tree(tree.right, indent + indent)

    def fit(self, X, Y):
        ''' function to train the tree '''

        dataset = np.concatenate((X, Y), axis=1)
        self.root = self.build_tree(dataset)

    def predict(self, X):
        ''' function to predict new dataset '''

        preditions = [self.make_prediction(x, self.root) for x in X]
        return preditions

    def make_prediction(self, x, tree):
        ''' function to predict a single data point '''

        if tree.value!=None: return tree.value
        feature_val = x[tree.feature_index]
        if feature_val<=tree.threshold:
            return self.make_prediction(x, tree.left)
        else:
            return self.make_prediction(x, tree.right)

import numpy as np
from numpy.core.umath_tests import inner1d
from copy import deepcopy


class AdaBoostClassifier(object):
    '''
    Parameters
    -----------
    base_estimator: object
        The base model from which the boosted ensemble is built.
    n_estimators: integer, optional(default=50)
```

```
    The maximum number of estimators
learning_rate: float, optional(default=1)
algorithm: {'SAMME','SAMME.R'}, optional(default='SAMME.R')
    SAMME.R uses predicted probabilities to update wights, while SAMME uses class error rate
random_state: int or None, optional(default=None)
Attributes
-------------
estimators_: list of base estimators
estimator_weights_: array of floats
    Weights for each base_estimator
estimator_errors_: array of floats
    Classification error for each estimator in the boosted ensemble.
Reference:
1. [multi-adaboost](https://web.stanford.edu/~hastie/Papers/samme.pdf)
2. [scikit-learn:weight_boosting](https://github.com/scikit-learn/
scikit-learn/blob/51a765a/sklearn/ensemble/weight_boosting.py#L289)
'''

def __init__(self, *args, **kwargs):
    if kwargs and args:
        raise ValueError(
            '''AdaBoostClassifier can only be called with keyword
               arguments for the following keywords: base_estimator ,n_estimators,
                learning_rate,algorithm,random_state''')
    allowed_keys = ['base_estimator', 'n_estimators', 'learning_rate', 'algorithm', 'random_state']
    keywords_used = kwargs.keys()
    for keyword in keywords_used:
        if keyword not in allowed_keys:
            raise ValueError(keyword + ":  Wrong keyword used --- check spelling")

    n_estimators = 50
    learning_rate = 1
    algorithm = 'SAMME.R'
    random_state = None

    if kwargs and not args:
        if 'base_estimator' in kwargs:
            base_estimator = kwargs.pop('base_estimator')
        else:
            raise ValueError('''base_estimator can not be None''')
        if 'n_estimators' in kwargs: n_estimators = kwargs.pop('n_estimators')
        if 'learning_rate' in kwargs: learning_rate = kwargs.pop('learning_rate')
        if 'algorithm' in kwargs: algorithm = kwargs.pop('algorithm')
        if 'random_state' in kwargs: random_state = kwargs.pop('random_state')

    self.base_estimator_ = base_estimator
    self.n_estimators_ = n_estimators
    self.learning_rate_ = learning_rate
    self.algorithm_ = algorithm
    self.random_state_ = random_state
    self.estimators_ = list()
    self.estimator_weights_ = np.zeros(self.n_estimators_)
    self.estimator_errors_ = np.ones(self.n_estimators_)


def _samme_proba(self, estimator, n_classes, X):
    """Calculate algorithm 4, step 2, equation c) of Zhu et al [1].
    References
    ----------
    .. [1] J. Zhu, H. Zou, S. Rosset, T. Hastie, "Multi-class AdaBoost", 2009.
    """
    proba = estimator.predict_proba(X)

    # Displace zero probabilities so the log is defined.
    # Also fix negative elements which may occur with
    # negative sample weights.
    proba[proba < np.finfo(proba.dtype).eps] = np.finfo(proba.dtype).eps
    log_proba = np.log(proba)

    return (n_classes - 1) * (log_proba - (1. / n_classes)
                              * log_proba.sum(axis=1)[:, np.newaxis])


def fit(self, X, y):
    self.n_samples = X.shape[0]
    # There is hidden trouble for classes, here the classes will be sorted.
    # So in boost we have to ensure that the predict results have the same classes sort
    self.classes_ = np.array(sorted(list(set(y))))
    self.n_classes_ = len(self.classes_)
    for iboost in range(self.n_estimators_):
        if iboost == 0:
            sample_weight = np.ones(self.n_samples) / self.n_samples
```

```
        sample_weight, estimator_weight, estimator_error = self.boost(X, y, sample_weight)

        # early stop
        if estimator_error == None:
            break

        # append error and weight
        self.estimator_errors_[iboost] = estimator_error
        self.estimator_weights_[iboost] = estimator_weight

        if estimator_error <= 0:
            break

    return self


def boost(self, X, y, sample_weight):
    if self.algorithm_ == 'SAMME':
        return self.discrete_boost(X, y, sample_weight)
    elif self.algorithm_ == 'SAMME.R':
        return self.real_boost(X, y, sample_weight)

def real_boost(self, X, y, sample_weight):
    estimator = deepcopy(self.base_estimator_)
    if self.random_state_:
        estimator.set_params(random_state=1)

    estimator.fit(X, y, sample_weight=sample_weight)

    y_pred = estimator.predict(X)
    incorrect = y_pred != y
    estimator_error = np.dot(incorrect, sample_weight) / np.sum(sample_weight, axis=0)

    # if worse than random guess, stop boosting
    if estimator_error >= 1.0 - 1 / self.n_classes_:
        return None, None, None

    y_predict_proba = estimator.predict_proba(X)
    # repalce zero
    y_predict_proba[y_predict_proba < np.finfo(y_predict_proba.dtype).eps] = np.finfo(y_predict_proba.dtype).eps

    y_codes = np.array([-1. / (self.n_classes_ - 1), 1.])
    y_coding = y_codes.take(self.classes_ == y[:, np.newaxis])

    # for sample weight update
    intermediate_variable = (-1. * self.learning_rate_ * (((self.n_classes_ - 1) / self.n_classes_) *
                                                           inner1d(y_coding, np.log(
                                                               y_predict_proba))))  #dot iterate for each row

    # update sample weight
    sample_weight *= np.exp(intermediate_variable)

    sample_weight_sum = np.sum(sample_weight, axis=0)
    if sample_weight_sum <= 0:
        return None, None, None

    # normalize sample weight
    sample_weight /= sample_weight_sum

    # append the estimator
    self.estimators_.append(estimator)

    return sample_weight, 1, estimator_error


def discrete_boost(self, X, y, sample_weight):
    estimator = deepcopy(self.base_estimator_)
    if self.random_state_:
        estimator.set_params(random_state=1)

    estimator.fit(X, y, sample_weight=sample_weight)

    y_pred = estimator.predict(X)
    incorrect = y_pred != y
    estimator_error = np.dot(incorrect, sample_weight) / np.sum(sample_weight, axis=0)

    # if worse than random guess, stop boosting
    if estimator_error >= 1 - 1 / self.n_classes_:
        return None, None, None

    # update estimator_weight
```

```
        estimator_weight = self.learning_rate_ * np.log((1 - estimator_error) / estimator_error) + np.log(
            self.n_classes_ - 1)

        if estimator_weight <= 0:
            return None, None, None

        # update sample weight
        sample_weight *= np.exp(estimator_weight * incorrect)

        sample_weight_sum = np.sum(sample_weight, axis=0)
        if sample_weight_sum <= 0:
            return None, None, None

        # normalize sample weight
        sample_weight /= sample_weight_sum

        # append the estimator
        self.estimators_.append(estimator)

        return sample_weight, estimator_weight, estimator_error

    def predict(self, X):
        n_classes = self.n_classes_
        classes = self.classes_[:, np.newaxis]
        pred = None

        if self.algorithm_ == 'SAMME.R':
            # The weights are all 1. for SAMME.R
            pred = sum(self._samme_proba(estimator, n_classes, X) for estimator in self.estimators_)
        else:  # self.algorithm == "SAMME"
            pred = sum((estimator.predict(X) == classes).T * w
                       for estimator, w in zip(self.estimators_,
                                               self.estimator_weights_))

        pred /= self.estimator_weights_.sum()
        if n_classes == 2:
            pred[:, 0] *= -1
            pred = pred.sum(axis=1)
            return self.classes_.take(pred > 0, axis=0)

        return self.classes_.take(np.argmax(pred, axis=1), axis=0)


    def predict_proba(self, X):
        if self.algorithm_ == 'SAMME.R':
            # The weights are all 1. for SAMME.R
            proba = sum(self._samme_proba(estimator, self.n_classes_, X)
                        for estimator in self.estimators_)
        else:  # self.algorithm == "SAMME"
            proba = sum(estimator.predict_proba(X) * w
                        for estimator, w in zip(self.estimators_,
                                                self.estimator_weights_))

        proba /= self.estimator_weights_.sum()
        proba = np.exp((1. / (n_classes - 1)) * proba)
        normalizer = proba.sum(axis=1)[:, np.newaxis]
        normalizer[normalizer == 0.0] = 1.0
        proba /= normalizer

        return proba
```

```
/var/folders/t3/ctmyp9cd5j7ctjhqp2_g843h0000gn/T/ipykernel_91204/1786590862.py:2: DeprecationWarning: numpy.core.umath_te
  from numpy.core.umath_tests import inner1d
```

```
# Fit the Adaboost model

Adaboost = AdaBoostClassifier(base_estimator = DecisionTreeClassifier(max_depth = 1), n_estimators = 50,
                              learning_rate = 1, algorithm = 'SAMME.R',
                              random_state = None)
Adaboost.fit(Xtrain,Ytrain)
```

```
/var/folders/t3/ctmyp9cd5j7ctjhqp2_g843h0000gn/T/ipykernel_91204/1786590862.py:138: FutureWarning: Support for multi-dime
  y_coding = y_codes.take(self.classes_ == y[:, np.newaxis])
/var/folders/t3/ctmyp9cd5j7ctjhqp2_g843h0000gn/T/ipykernel_91204/1786590862.py:138: FutureWarning: Support for multi-dime
  y_coding = y_codes.take(self.classes_ == y[:, np.newaxis])
/var/folders/t3/ctmyp9cd5j7ctjhqp2_g843h0000gn/T/ipykernel_91204/1786590862.py:138: FutureWarning: Support for multi-dime
  y_coding = y_codes.take(self.classes_ == y[:, np.newaxis])
/var/folders/t3/ctmyp9cd5j7ctjhqp2_g843h0000gn/T/ipykernel_91204/1786590862.py:138: FutureWarning: Support for multi-dime
  y_coding = y_codes.take(self.classes_ == y[:, np.newaxis])
/var/folders/t3/ctmyp9cd5j7ctjhqp2_g843h0000gn/T/ipykernel_91204/1786590862.py:138: FutureWarning: Support for multi-dime
  y_coding = y_codes.take(self.classes_ == y[:, np.newaxis])
/var/folders/t3/ctmyp9cd5j7ctjhqp2_g843h0000gn/T/ipykernel_91204/1786590862.py:138: FutureWarning: Support for multi-dime
  y_coding = y_codes.take(self.classes_ == y[:, np.newaxis])
/var/folders/t3/ctmyp9cd5j7ctjhqp2_g843h0000gn/T/ipykernel_91204/1786590862.py:138: FutureWarning: Support for multi-dime
  y_coding = y_codes.take(self.classes_ == y[:, np.newaxis])
```

```
/var/folders/t3/ctmyp9cd5j7ctjhqp2_g843h0000gn/T/ipykernel_91204/1786590862.py:138: FutureWarning: Support for multi-dime
  y_coding = y_codes.take(self.classes_ == y[:, np.newaxis])
/var/folders/t3/ctmyp9cd5j7ctjhqp2_g843h0000gn/T/ipykernel_91204/1786590862.py:138: FutureWarning: Support for multi-dime
  y_coding = y_codes.take(self.classes_ == y[:, np.newaxis])
/var/folders/t3/ctmyp9cd5j7ctjhqp2_g843h0000gn/T/ipykernel_91204/1786590862.py:138: FutureWarning: Support for multi-dime
  y_coding = y_codes.take(self.classes_ == y[:, np.newaxis])
/var/folders/t3/ctmyp9cd5j7ctjhqp2_g843h0000gn/T/ipykernel_91204/1786590862.py:138: FutureWarning: Support for multi-dime
  y_coding = y_codes.take(self.classes_ == y[:, np.newaxis])
/var/folders/t3/ctmyp9cd5j7ctjhqp2_g843h0000gn/T/ipykernel_91204/1786590862.py:138: FutureWarning: Support for multi-dime
  y_coding = y_codes.take(self.classes_ == y[:, np.newaxis])
/var/folders/t3/ctmyp9cd5j7ctjhqp2_g843h0000gn/T/ipykernel_91204/1786590862.py:138: FutureWarning: Support for multi-dime
  y_coding = y_codes.take(self.classes_ == y[:, np.newaxis])
/var/folders/t3/ctmyp9cd5j7ctjhqp2_g843h0000gn/T/ipykernel_91204/1786590862.py:138: FutureWarning: Support for multi-dime
  y_coding = y_codes.take(self.classes_ == y[:, np.newaxis])
/var/folders/t3/ctmyp9cd5j7ctjhqp2_g843h0000gn/T/ipykernel_91204/1786590862.py:138: FutureWarning: Support for multi-dime
  y_coding = y_codes.take(self.classes_ == y[:, np.newaxis])
/var/folders/t3/ctmyp9cd5j7ctjhqp2_g843h0000gn/T/ipykernel_91204/1786590862.py:138: FutureWarning: Support for multi-dime
  y_coding = y_codes.take(self.classes_ == y[:, np.newaxis])
/var/folders/t3/ctmyp9cd5j7ctjhqp2_g843h0000gn/T/ipykernel_91204/1786590862.py:138: FutureWarning: Support for multi-dime
  y_coding = y_codes.take(self.classes_ == y[:, np.newaxis])
/var/folders/t3/ctmyp9cd5j7ctjhqp2_g843h0000gn/T/ipykernel_91204/1786590862.py:138: FutureWarning: Support for multi-dime
  y_coding = y_codes.take(self.classes_ == y[:, np.newaxis])
/var/folders/t3/ctmyp9cd5j7ctjhqp2_g843h0000gn/T/ipykernel_91204/1786590862.py:138: FutureWarning: Support for multi-dime
  y_coding = y_codes.take(self.classes_ == y[:, np.newaxis])
/var/folders/t3/ctmyp9cd5j7ctjhqp2_g843h0000gn/T/ipykernel_91204/1786590862.py:138: FutureWarning: Support for multi-dime
  y_coding = y_codes.take(self.classes_ == y[:, np.newaxis])
/var/folders/t3/ctmyp9cd5j7ctjhqp2_g843h0000gn/T/ipykernel_91204/1786590862.py:138: FutureWarning: Support for multi-dime
  y_coding = y_codes.take(self.classes_ == y[:, np.newaxis])
/var/folders/t3/ctmyp9cd5j7ctjhqp2_g843h0000gn/T/ipykernel_91204/1786590862.py:138: FutureWarning: Support for multi-dime
  y_coding = y_codes.take(self.classes_ == y[:, np.newaxis])
/var/folders/t3/ctmyp9cd5j7ctjhqp2_g843h0000gn/T/ipykernel_91204/1786590862.py:138: FutureWarning: Support for multi-dime
  y_coding = y_codes.take(self.classes_ == y[:, np.newaxis])
/var/folders/t3/ctmyp9cd5j7ctjhqp2_g843h0000gn/T/ipykernel_91204/1786590862.py:138: FutureWarning: Support for multi-dime
  y_coding = y_codes.take(self.classes_ == y[:, np.newaxis])
/var/folders/t3/ctmyp9cd5j7ctjhqp2_g843h0000gn/T/ipykernel_91204/1786590862.py:138: FutureWarning: Support for multi-dime
  y_coding = y_codes.take(self.classes_ == y[:, np.newaxis])
/var/folders/t3/ctmyp9cd5j7ctjhqp2_g843h0000gn/T/ipykernel_91204/1786590862.py:138: FutureWarning: Support for multi-dime
  y_coding = y_codes.take(self.classes_ == y[:, np.newaxis])
/var/folders/t3/ctmyp9cd5j7ctjhqp2_g843h0000gn/T/ipykernel_91204/1786590862.py:138: FutureWarning: Support for multi-dime
  y_coding = y_codes.take(self.classes_ == y[:, np.newaxis])
/var/folders/t3/ctmyp9cd5j7ctjhqp2_g843h0000gn/T/ipykernel_91204/1786590862.py:138: FutureWarning: Support for multi-dime
```

```python
Ypredict = Adaboost.predict(Xtest)
Ypredict
```

```
array([5., 5., 3., ..., 3., 1., 3.])
```

```python
print('----- Testing dataset validation -----')
print('')
print('The confusion matrix of testing dataset is:')
print(confusion_matrix(Ytest,Ypredict))
print('')
print("Accuracy of logistic regression for testing dataset is:",round(accuracy_score(Ytest,Ypredict),4))
```

```
----- Testing dataset validation -----

The confusion matrix of testing dataset is:
[[2886    4  349    0    8]
 [  94 1304   35   60 1512]
 [  82   13 2868  204   48]
 [  21   78  712 1955  242]
 [  87  487   52  163 2355]]

Accuracy of logistic regression for testing dataset is: 0.7278
```