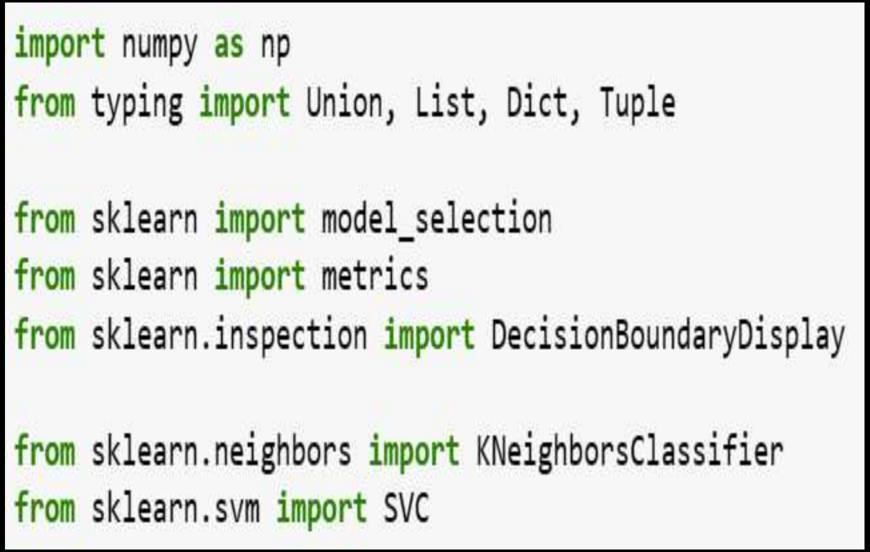
PATTERN RECOGNITION ASSIGNMENT

Team 30

Antonios Antoniou 9482
Konstantinos Kalamaras 9716

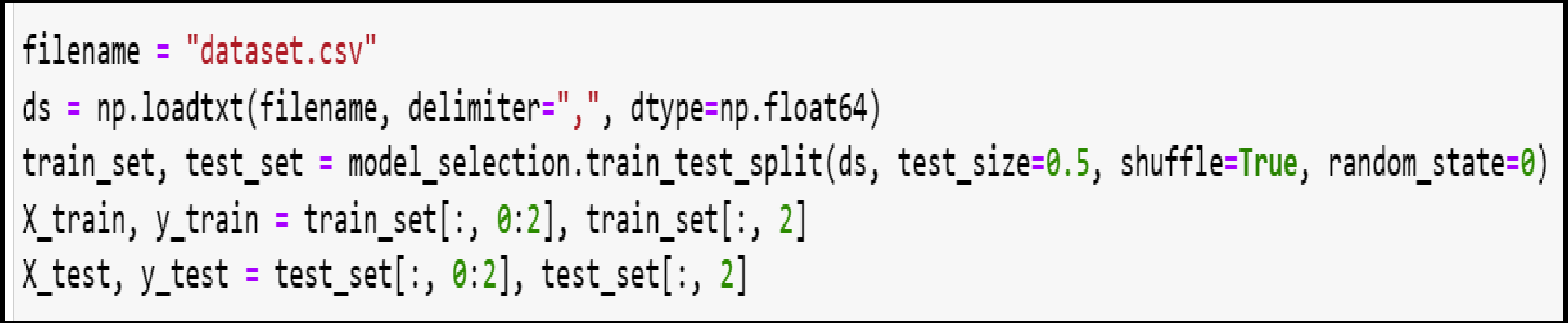
INTRODUCTION AND LIBRARIES

First of all , we import numpy and sklearn libraries in addition to some of their modules , in order to : implement classification methods(SVC & KNeighborsClassifier) , measure classification performance (metrics) , split data into two sets (model_selection) etc.



```
import numpy as np
from typing import Union, List, Dict, Tuple

from sklearn import model_selection
from sklearn import metrics
from sklearn.inspection import DecisionBoundaryDisplay

from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
```

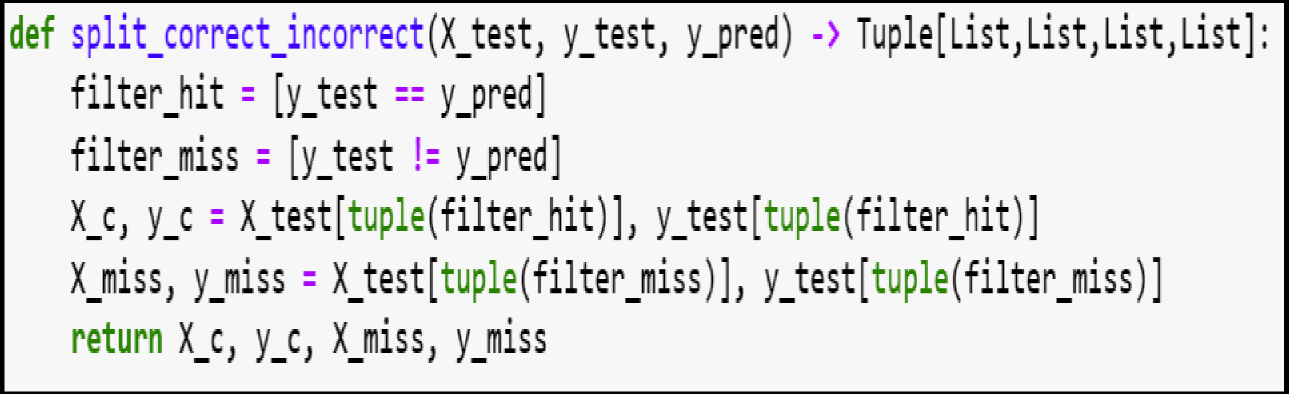
Then , we load our data and split it into two sets (training and testing). Finally , we divide each of those sets into their two parts , the features and labels(the last column).



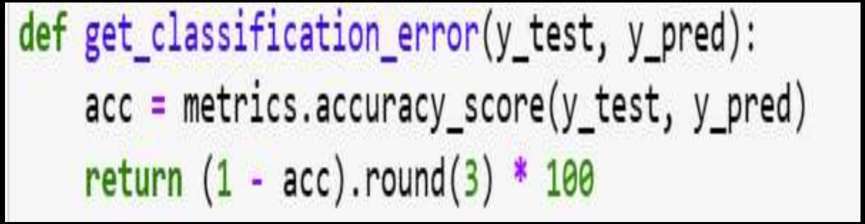
```
filename = "dataset.csv"
ds = np.loadtxt(filename, delimiter=",", dtype=np.float64)
train_set, test_set = model_selection.train_test_split(ds, test_size=0.5, shuffle=True, random_state=0)
X_train, y_train = train_set[:, 0:2], train_set[:, 2]
X_test, y_test = test_set[:, 0:2], test_set[:, 2]
```

GENERALLY USED FUNCTIONS

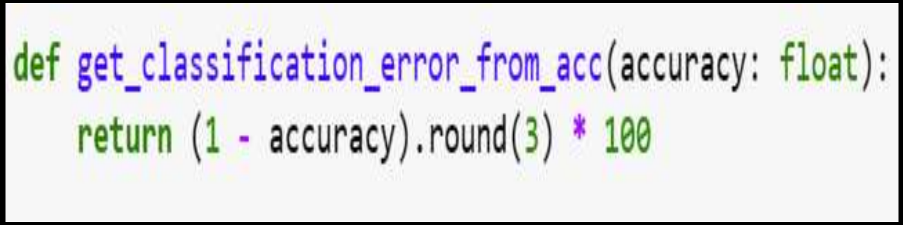
Firstly, we create the split_correct_incorrect , to split the classified datapoints into two categories: correctly placed and misclassified ones.
We also create two functions : get_classification_error , get_classification_error_from_acc , to calculate the classification error using the accuracy score(pre-computed or not).



```
def split_correct_incorrect(X_test, y_test, y_pred) -> Tuple[List,List,List,List]:
    filter_hit = [y_test == y_pred]
    filter_miss = [y_test != y_pred]
    X_c, y_c = X_test[tuple(filter_hit)], y_test[tuple(filter_hit)]
    X_miss, y_miss = X_test[tuple(filter_miss)], y_test[tuple(filter_miss)]
    return X_c, y_c, X_miss, y_miss
```

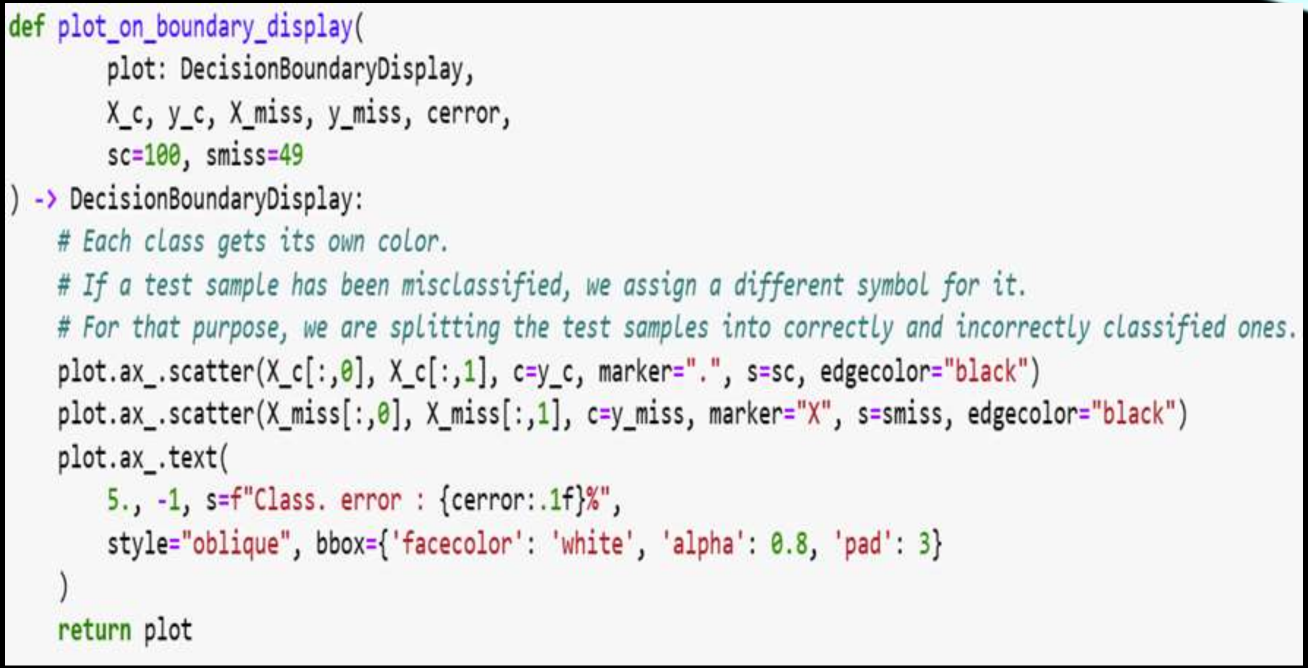


```
def get_classification_error(y_test, y_pred):
    acc = metrics.accuracy_score(y_test, y_pred)
    return (1 - acc).round(3) * 100
```

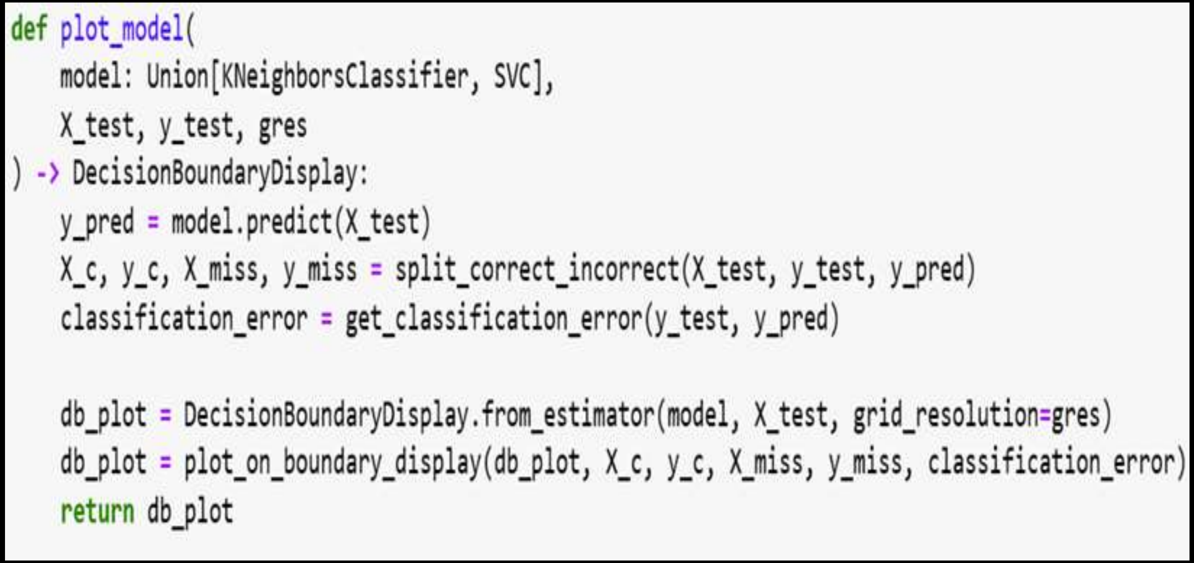


```
def get_classification_error_from_acc(accuracy: float):
    return (1 - accuracy).round(3) * 100
```

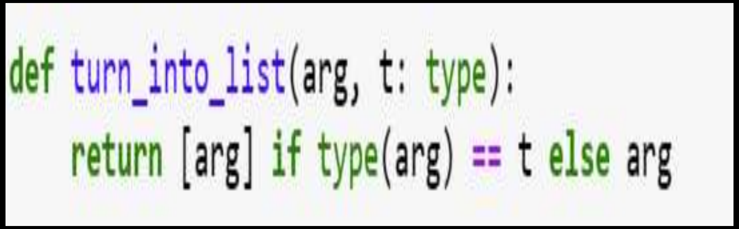
Afterwards, we create two main functions to achieve the plot of our datapoints in every question. The first one (plot_on_boundary_display), targets to split points into different colored decision boundaries and differentiate the symbolism of correctly and misclassified datapoints. The second one (plot_model), calculates the prediction of a model on the test data, plots the decision boundary and the distribution of the datapoints.

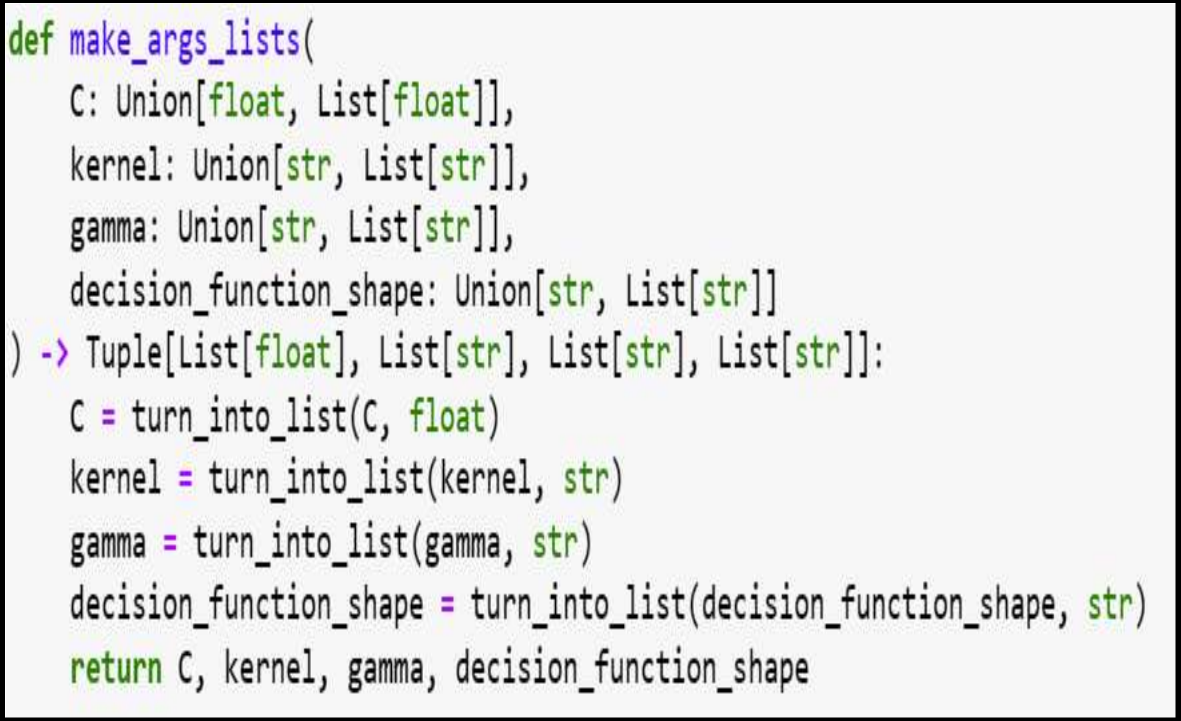


```python
def plot_on_boundary_display(
        plot: DecisionBoundaryDisplay,
        X_c, y_c, X_miss, y_miss, cerror,
        sc=100, smiss=49
) -> DecisionBoundaryDisplay:
    # Each class gets its own color.
    # If a test sample has been misclassified, we assign a different symbol for it.
    # For that purpose, we are splitting the test samples into correctly and incorrectly classified ones.
    plot.ax_.scatter(X_c[:,0], X_c[:,1], c=y_c, marker=".", s=sc, edgecolor="black")
    plot.ax_.scatter(X_miss[:,0], X_miss[:,1], c=y_miss, marker="X", s=smiss, edgecolor="black")
    plot.ax_.text(
        5., -1, s=f"Class. error : {cerror:.1f}%",
        style="oblique", bbox={'facecolor': 'white', 'alpha': 0.8, 'pad': 3}
    )
    return plot
```



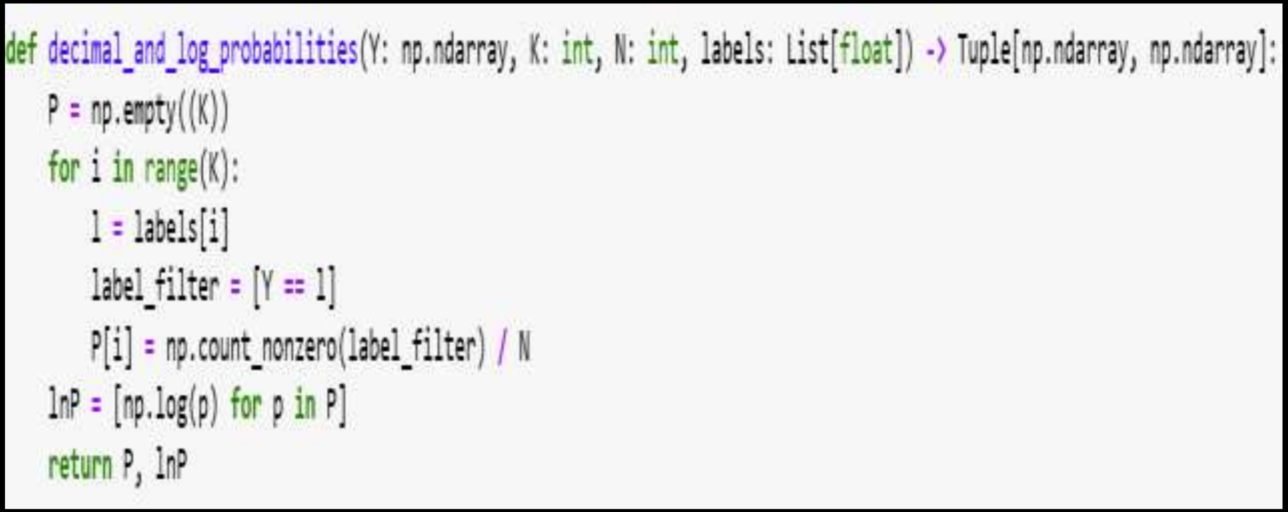
```python
def plot_model(
    model: Union[KNeighborsClassifier, SVC],
    X_test, y_test, gres
) -> DecisionBoundaryDisplay:
    y_pred = model.predict(X_test)
    X_c, y_c, X_miss, y_miss = split_correct_incorrect(X_test, y_test, y_pred)
    classification_error = get_classification_error(y_test, y_pred)

    db_plot = DecisionBoundaryDisplay.from_estimator(model, X_test, grid_resolution=gres)
    db_plot = plot_on_boundary_display(db_plot, X_c, y_c, X_miss, y_miss, classification_error)
    return db_plot
```


```
def turn_into_list(arg, t: type):
    return [arg] if type(arg) == t else arg
```


```
def make_args_lists(
    C: Union[float, List[float]],
    kernel: Union[str, List[str]],
    gamma: Union[str, List[str]],
    decision_function_shape: Union[str, List[str]]
) -> Tuple[List[float], List[str], List[str], List[str]]:
    C = turn_into_list(C, float)
    kernel = turn_into_list(kernel, str)
    gamma = turn_into_list(gamma, str)
    decision_function_shape = turn_into_list(decision_function_shape, str)
    return C, kernel, gamma, decision_function_shape
```

Those two last generally used function's goal is to turn all the required data into lists or check if they already are in the correct form. In more detail , the turn_into_list function takes an argument of a certain type and checks if it is a single value or a list of values. If we talk about a single value , it is transformed into a list of size 1, otherwise the argument is returned intact. The make_args_list , takes advantage of the previous function , to turn a whole list of arguments into individual lists , so as to make the iteration over all of them possible.

QUESTION A
BAYES CLASSIFIER

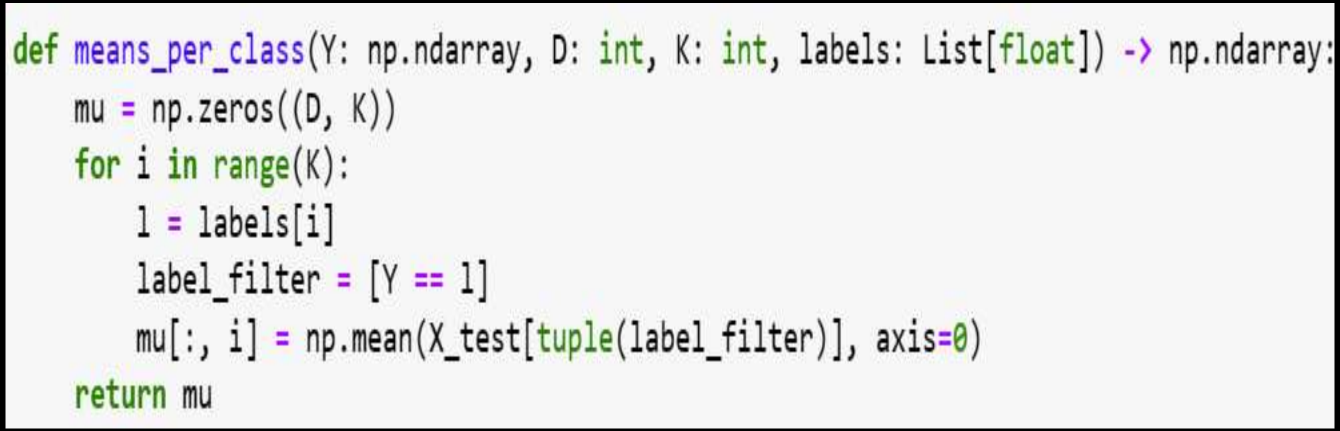


```
def decimal_and_log_probabilities(Y: np.ndarray, K: int, N: int, labels: List[float]) -> Tuple[np.ndarray, np.ndarray]:
    P = np.empty((K))
    for i in range(K):
        l = labels[i]
        label_filter = [Y == l]
        P[i] = np.count_nonzero(label_filter) / N
    lnP = [np.log(p) for p in P]
    return P, lnP
```

In this first question , we are asked to implement the Maximum Likelihood technique to create a Bayes Classifier. The first function we make , is the one that calculates the probabilities of the occurrence of each class , together with their natural logarithm , as the ML Theory demands : P = $p(D|\theta) = \prod_{n=1}^{N} p(x_n|\theta)$ and lnP = $l(\theta) = \ln \prod_{n=1}^{N} p(x_n|\theta)$.

Note : We have already divided our data into training and test sets in the introduction code.

Subsequently , we create the means_per_class function , targeting the creation of a statistical board that contains the means of some valuable features. Each row of the returned matrix contains the mean of a feature for each class. On the other hand , each column consists of the means for all features of one class.



```
def means_per_class(Y: np.ndarray, D: int, K: int, labels: List[float]) -> np.ndarray:
    mu = np.zeros((D, K))
    for i in range(K):
        l = labels[i]
        label_filter = [Y == l]
        mu[:, i] = np.mean(X_test[tuple(label_filter)], axis=0)
    return mu
```

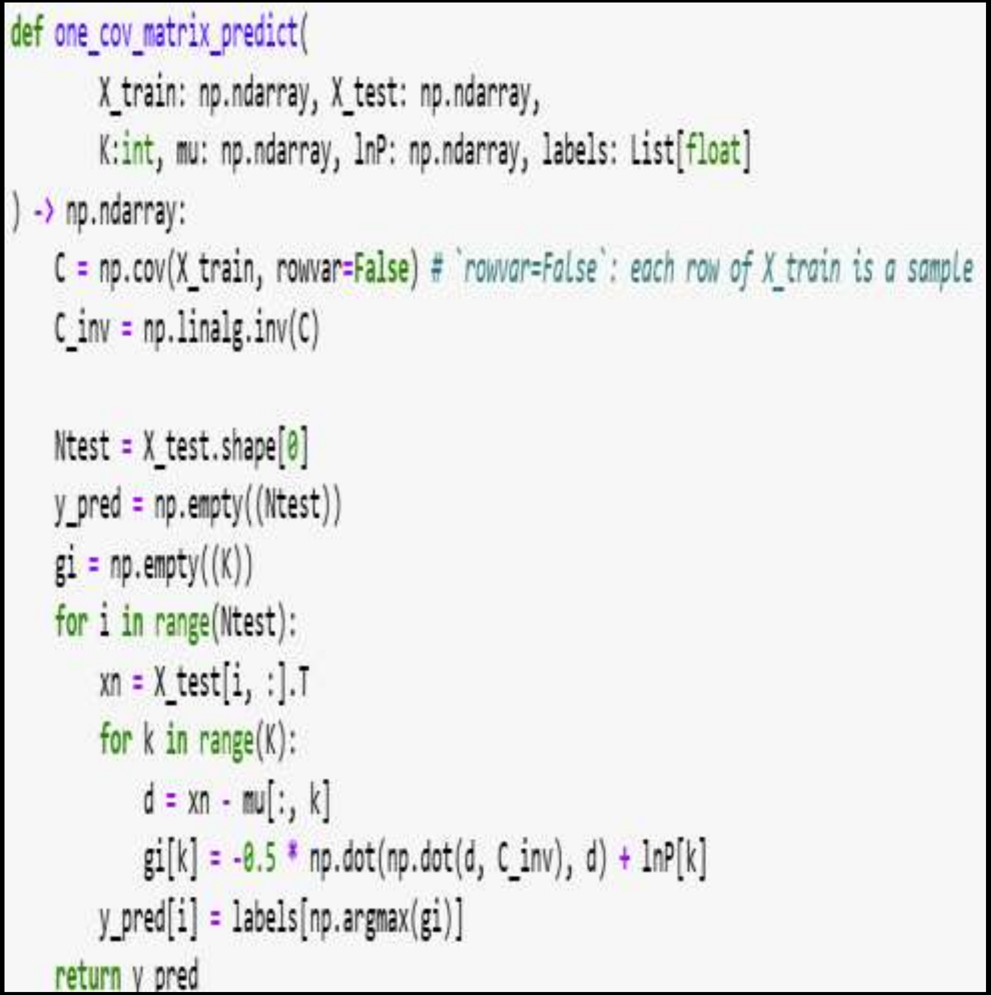
In more detail , in this question we are expected to create a Bayes Classifier for two different cases : 1) assuming that all classes have the same covariance table and 2) assuming that every class has a different covariance table.

As our theory indicates , taking into account that we are dealing with a normal distribution , the discriminant function's , we are using to classify our data , general form is :

$$g_i(x) = \frac{-1}{2}(x-\mu_i)^t\Sigma_i^{-1}(x-\mu_i) - \frac{d}{2}\ln(2\pi) - \frac{1}{2}ln|\Sigma_i| + lnP(\omega_i)$$

Dealing with the first case (same covariance table) , we create the one_cov_matrix_predict , to help us classify our datapoints. Now , our samples fall in a hyper-ellipsoidal clusters of equal size and same shape centered about the mean vector μ_i and the discriminant function is simplified into :

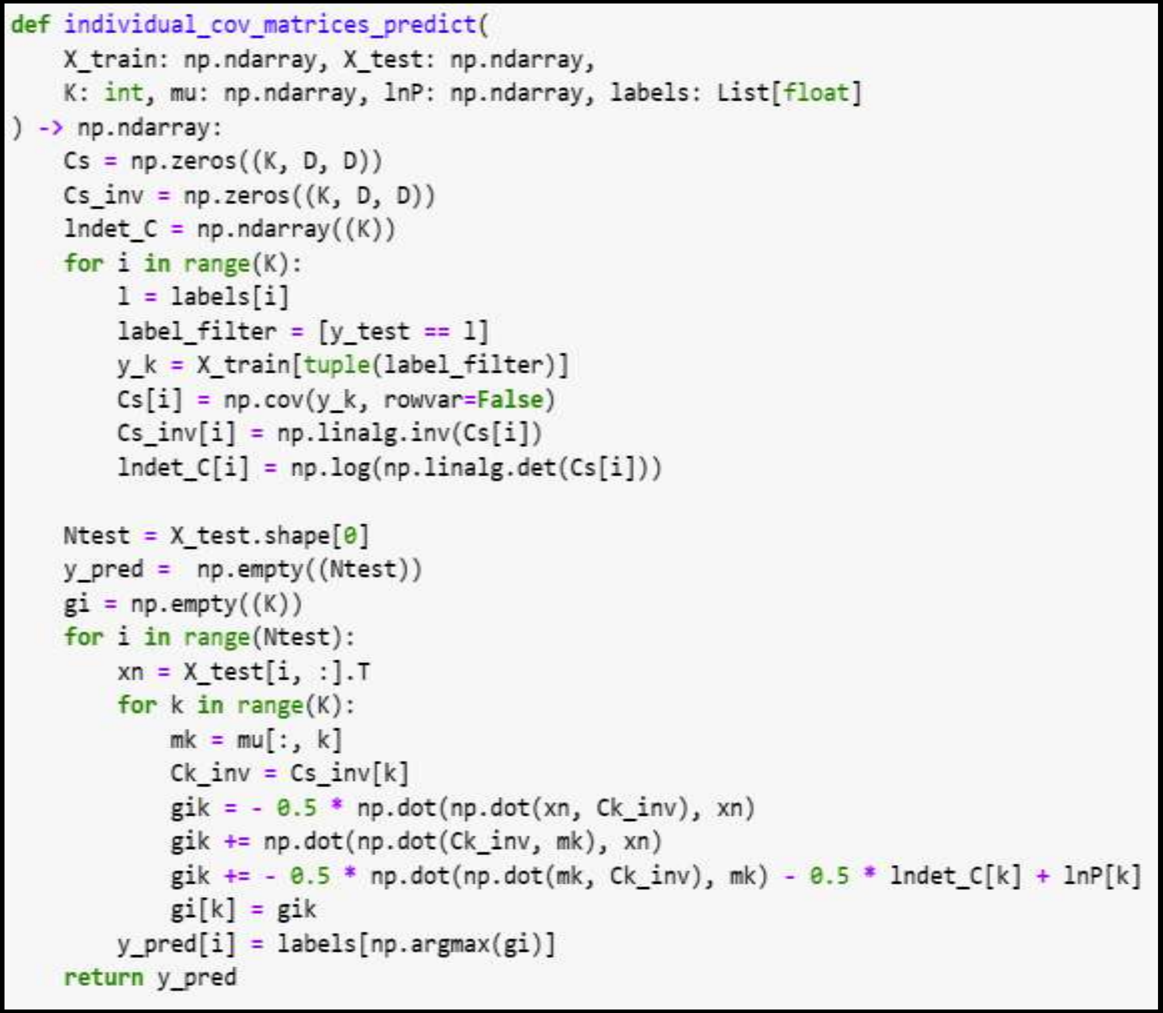
$$g_i(x) = \frac{-1}{2}(x-\mu_i)^t\Sigma_i^{-1}(x-\mu_i) + lnP(\omega_i)$$



```
def one_cov_matrix_predict(
        X_train: np.ndarray, X_test: np.ndarray,
        K:int, mu: np.ndarray, lnP: np.ndarray, labels: List[float]
) -> np.ndarray:
    C = np.cov(X_train, rowvar=False) # `rowvar=False`: each row of X_train is a sample
    C_inv = np.linalg.inv(C)

    Ntest = X_test.shape[0]
    y_pred = np.empty((Ntest))
    gi = np.empty((K))
    for i in range(Ntest):
        xn = X_test[i, :].T
        for k in range(K):
            d = xn - mu[:, k]
            gi[k] = -0.5 * np.dot(np.dot(d, C_inv), d) + lnP[k]
        y_pred[i] = labels[np.argmax(gi)]
    return y_pred
```

Moving on to the second case (different covariance tables) , we create the individual_cov_matrices_predict , to help us classify our datapoints. In this case , the decision surfaces are hyperquadrics and get any of the general forms, e.g. , hyperplanes, hyperspheres, hyperellipsoids etc. and the discriminant function has a form :

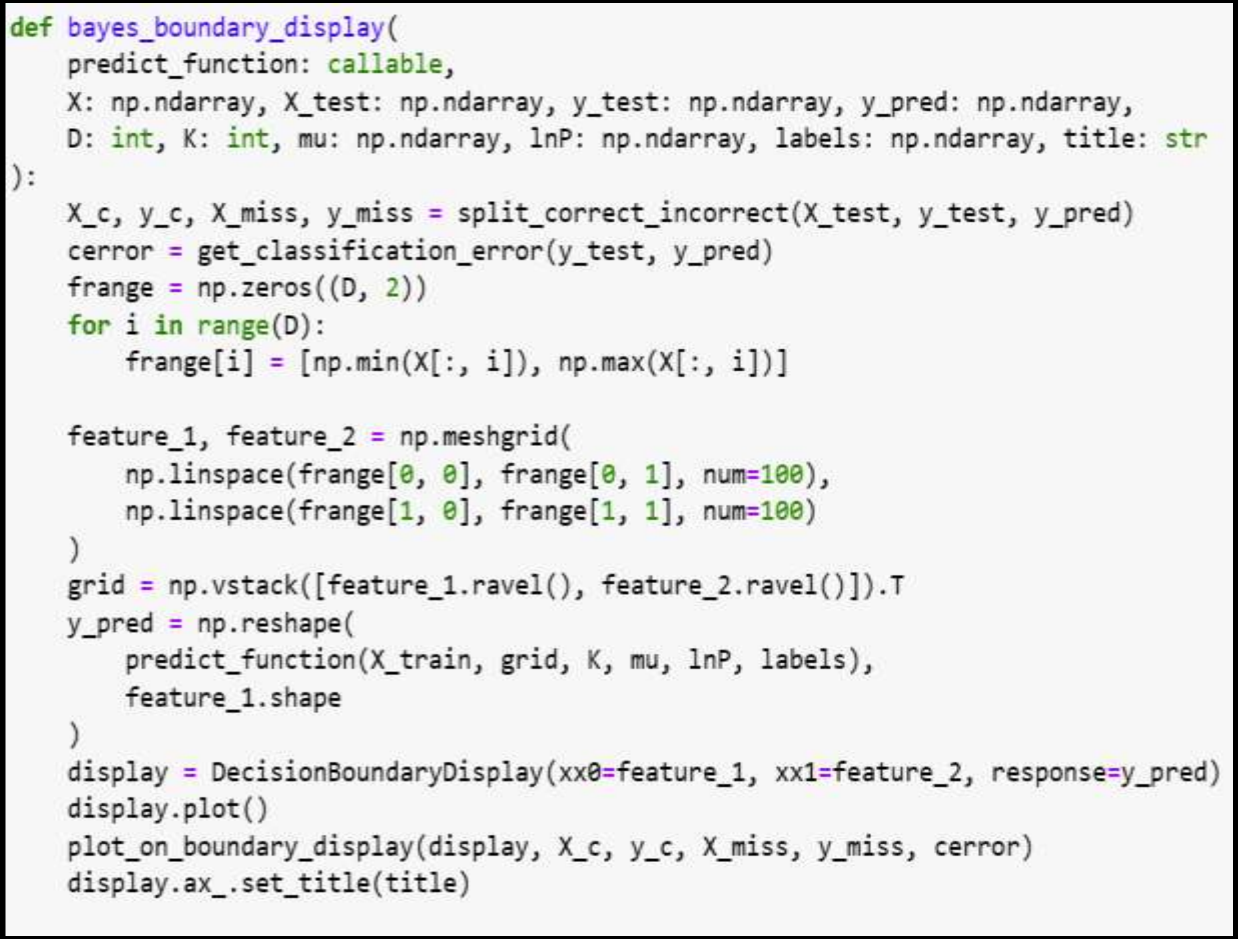
$$g_i(x) = \frac{-1}{2}(x-\mu_i)^t \Sigma_i^{-1}(x-\mu_i) - \frac{1}{2} ln|\Sigma_i| + lnP(\omega_i)$$



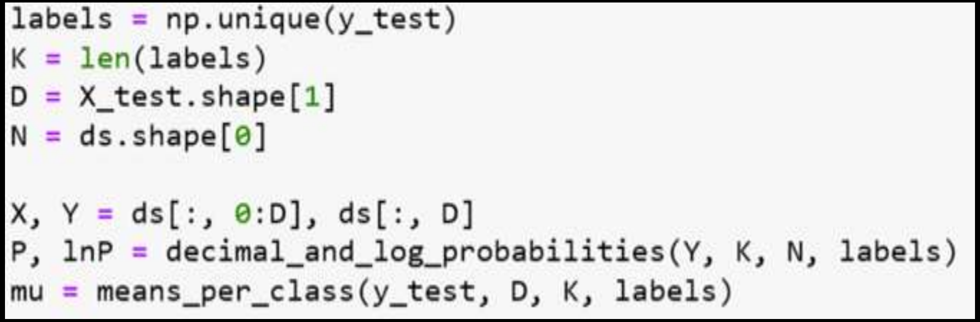
```python
def individual_cov_matrices_predict(
    X_train: np.ndarray, X_test: np.ndarray,
    K: int, mu: np.ndarray, lnP: np.ndarray, labels: List[float]
) -> np.ndarray:
    Cs = np.zeros((K, D, D))
    Cs_inv = np.zeros((K, D, D))
    lndet_C = np.ndarray((K))
    for i in range(K):
        l = labels[i]
        label_filter = [y_test == l]
        y_k = X_train[tuple(label_filter)]
        Cs[i] = np.cov(y_k, rowvar=False)
        Cs_inv[i] = np.linalg.inv(Cs[i])
        lndet_C[i] = np.log(np.linalg.det(Cs[i]))

    Ntest = X_test.shape[0]
    y_pred =  np.empty((Ntest))
    gi = np.empty((K))
    for i in range(Ntest):
        xn = X_test[i, :].T
        for k in range(K):
            mk = mu[:, k]
            Ck_inv = Cs_inv[k]
            gik = - 0.5 * np.dot(np.dot(xn, Ck_inv), xn)
            gik += np.dot(np.dot(Ck_inv, mk), xn)
            gik += - 0.5 * np.dot(np.dot(mk, Ck_inv), mk) - 0.5 * lndet_C[k] + lnP[k]
            gi[k] = gik
        y_pred[i] = labels[np.argmax(gi)]
    return y_pred
```

The last function we create in this question , is the bayes_boundary_display. The aim of this function , is to make a display of the decision boundaries of our bayes classifier , using the callable classifier predict_function to make predictions on the X_test. Additionally , it scatters the correctly and incorrectly classified points in our display and notes the classification error that is computed.

Moving to the implementation and visualization of this question , we start by extracting some valuable data , such as : K : number of classes , D : dimension of samples , N : number of total samples and computing the statistical features we talked about in the first two slides of this question (P , lnP , means).

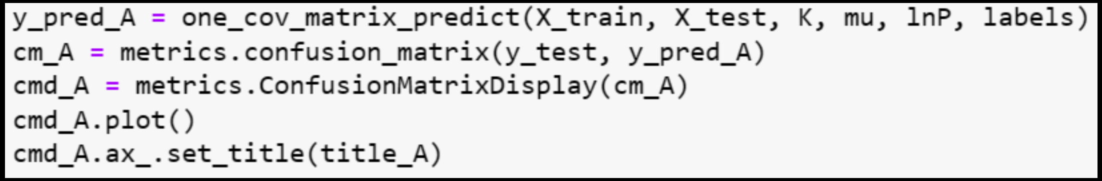


```
def bayes_boundary_display(
    predict_function: callable,
    X: np.ndarray, X_test: np.ndarray, y_test: np.ndarray, y_pred: np.ndarray,
    D: int, K: int, mu: np.ndarray, lnP: np.ndarray, labels: np.ndarray, title: str
):
    X_c, y_c, X_miss, y_miss = split_correct_incorrect(X_test, y_test, y_pred)
    cerror = get_classification_error(y_test, y_pred)
    frange = np.zeros((D, 2))
    for i in range(D):
        frange[i] = [np.min(X[:, i]), np.max(X[:, i])]

    feature_1, feature_2 = np.meshgrid(
        np.linspace(frange[0, 0], frange[0, 1], num=100),
        np.linspace(frange[1, 0], frange[1, 1], num=100)
    )
    grid = np.vstack([feature_1.ravel(), feature_2.ravel()]).T
    y_pred = np.reshape(
        predict_function(X_train, grid, K, mu, lnP, labels),
        feature_1.shape
    )
    display = DecisionBoundaryDisplay(xx0=feature_1, xx1=feature_2, response=y_pred)
    display.plot()
    plot_on_boundary_display(display, X_c, y_c, X_miss, y_miss, cerror)
    display.ax_.set_title(title)
```



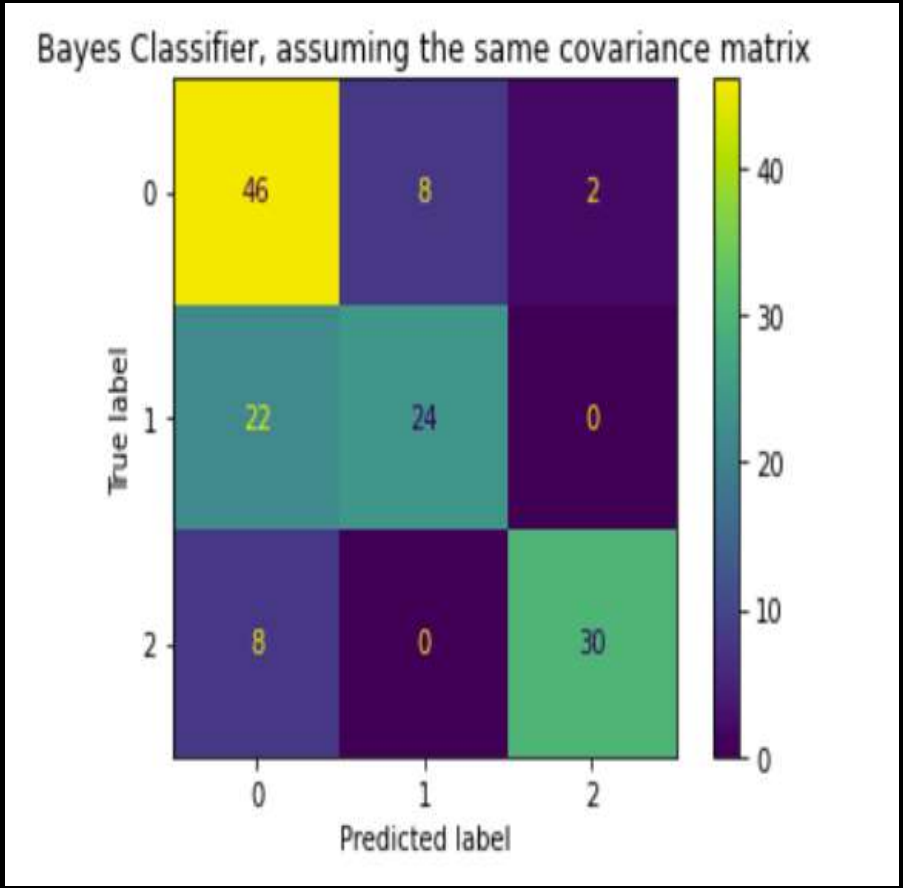
```
labels = np.unique(y_test)
K = len(labels)
D = X_test.shape[1]
N = ds.shape[0]

X, Y = ds[:, 0:D], ds[:, D]
P, lnP = decimal_and_log_probabilities(Y, K, N, labels)
mu = means_per_class(y_test, D, K, labels)
```

SAME COVARIANCE

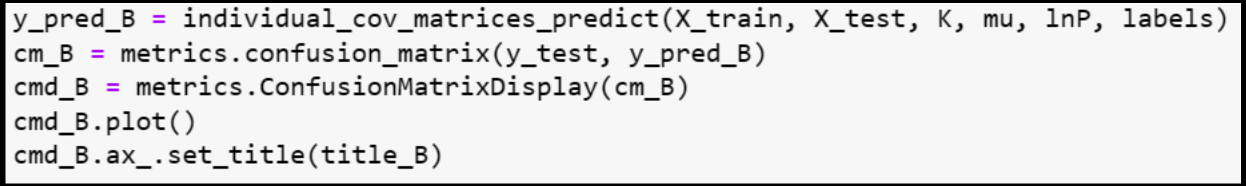


```
y_pred_A = one_cov_matrix_predict(X_train, X_test, K, mu, lnP, labels)
cm_A = metrics.confusion_matrix(y_test, y_pred_A)
cmd_A = metrics.ConfusionMatrixDisplay(cm_A)
cmd_A.plot()
cmd_A.ax_.set_title(title_A)
```

Assuming that all classes have the same covariance table C , we calculate this table using the one_cov_matrix_predict function for our training data and then take advantage of the metrics library , to plot the confusion matrix of this classifier.

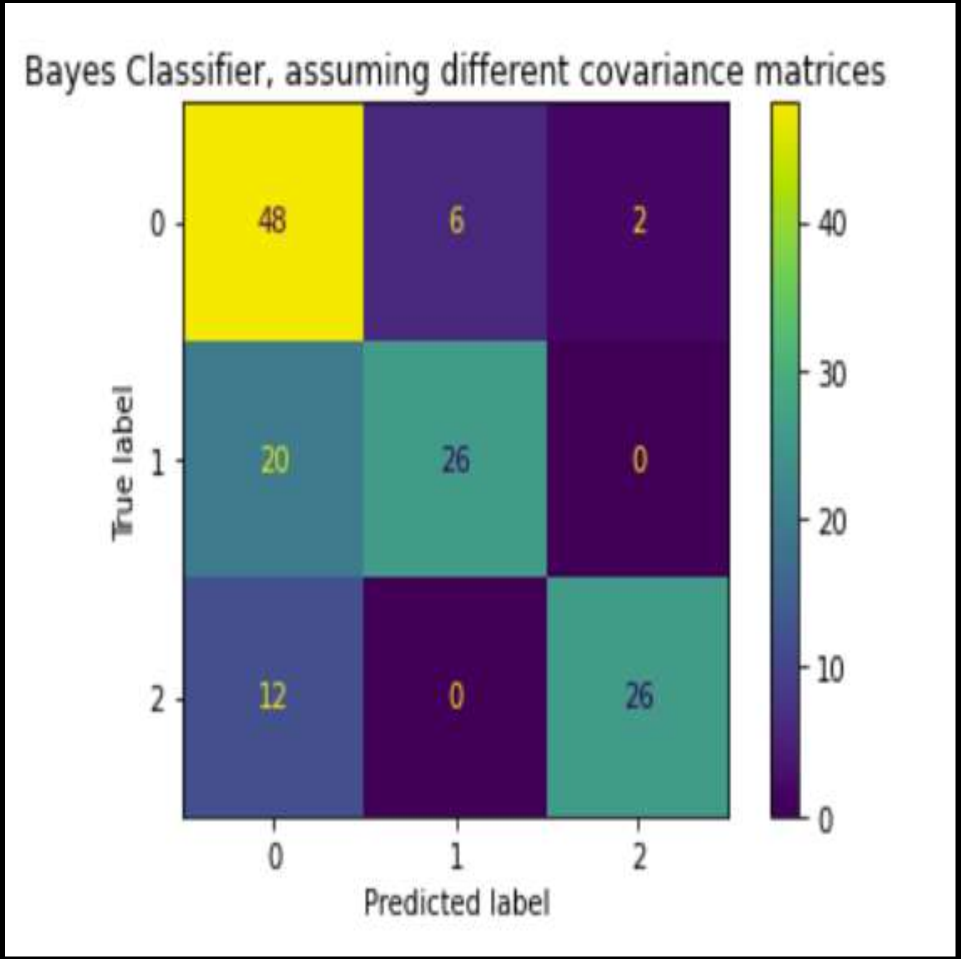


DIFFERENT COVARIANCES



```
y_pred_B = individual_cov_matrices_predict(X_train, X_test, K, mu, lnP, labels)
cm_B = metrics.confusion_matrix(y_test, y_pred_B)
cmd_B = metrics.ConfusionMatrixDisplay(cm_B)
cmd_B.plot()
cmd_B.ax_.set_title(title_B)
```

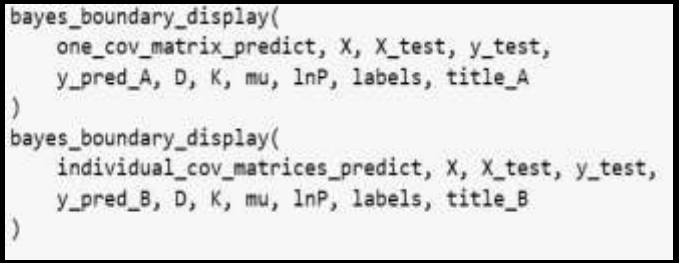
Same as in the previous slide , we generate a confusion matrix for the Bayes Classifier that assumes that every class has a different covariance matrix , this time using the individual_cov_matrices_predict function.



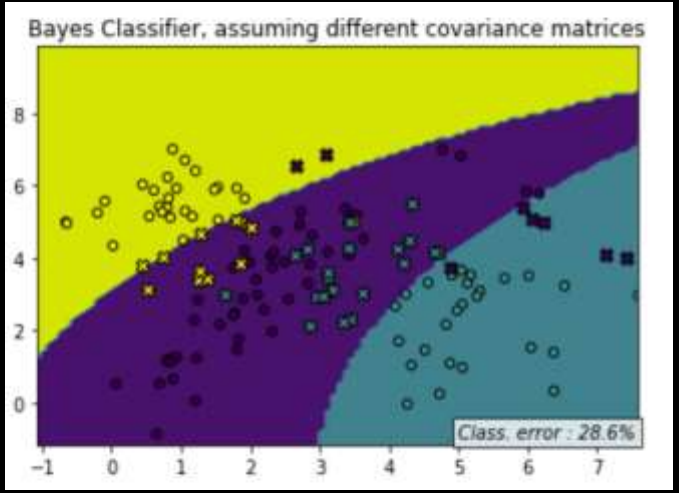
RESULTS AND CONCLUSION

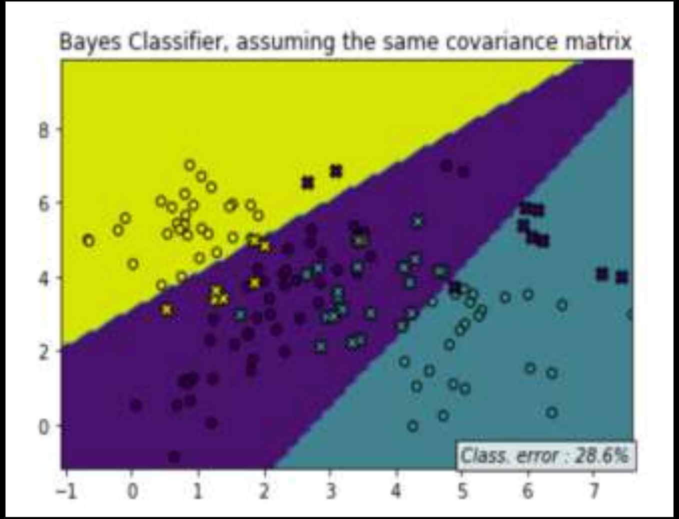
Our final move in this question , is to call the bayes_boundary_display function for both occasions and thus generate displays of the Decision Boundaries , with the misclassified datapoints being marked.

A general conclusion we can make , is that while the decision regions are different (of different shape but around the same bounds), and less linear in the case where we consider different covariance matrices for each class, we can see that the nature of the dataset doesn't allow for the benefits of a non-linear classifier. What we are left with is a more flexible classifier, that makes the same error as its linear counterpart.



```
bayes_boundary_display(
    one_cov_matrix_predict, X, X_test, y_test,
    y_pred_A, D, K, mu, lnP, labels, title_A
)
bayes_boundary_display(
    individual_cov_matrices_predict, X, X_test, y_test,
    y_pred_B, D, K, mu, lnP, labels, title_B
)
```



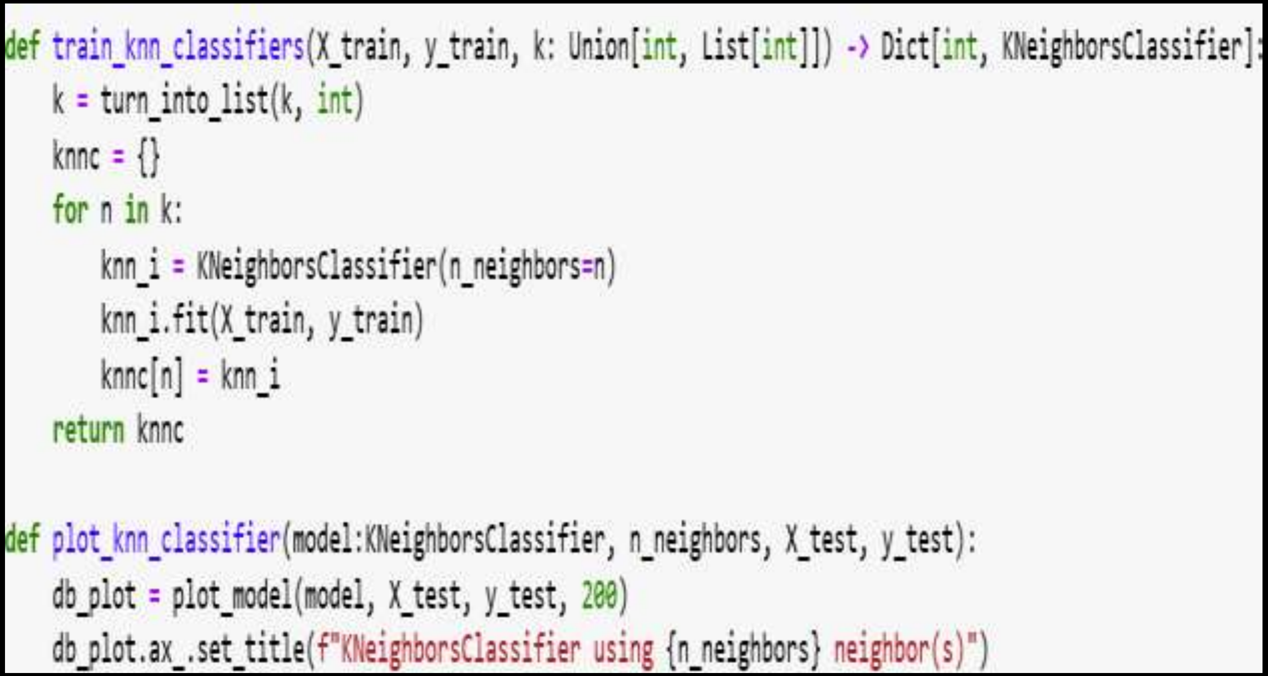


QUESTION B
K-NN CLASSIFIER FOR I = (1 … 10)

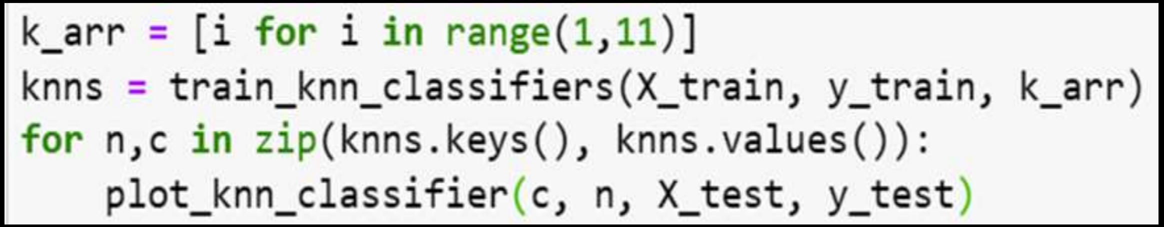
In this question , we start by creating a training function that takes a list of our data and accordingly to the number of neighbors we want to train our model upon , creates KNeighborsClassifiers and returns a Dictionary of classified data.

After that , we make the plot_knn_classifier function by using the previously created and explained plot_model function , in order to visualize the classification results for each number of neighbors we want to use.

Finally , we implement those two previously created functions for every number of neighbors between 1 and 10(as asked). The result is ten different classification diagrams , showing the number of neighbors used, the classification areas and the classification error in each case.



```python
def train_knn_classifiers(X_train, y_train, k: Union[int, List[int]]) -> Dict[int, KNeighborsClassifier]:
    k = turn_into_list(k, int)
    knnc = {}
    for n in k:
        knn_i = KNeighborsClassifier(n_neighbors=n)
        knn_i.fit(X_train, y_train)
        knnc[n] = knn_i
    return knnc

def plot_knn_classifier(model:KNeighborsClassifier, n_neighbors, X_test, y_test):
    db_plot = plot_model(model, X_test, y_test, 200)
    db_plot.ax_.set_title(f"KNeighborsClassifier using {n_neighbors} neighbor(s)")
```



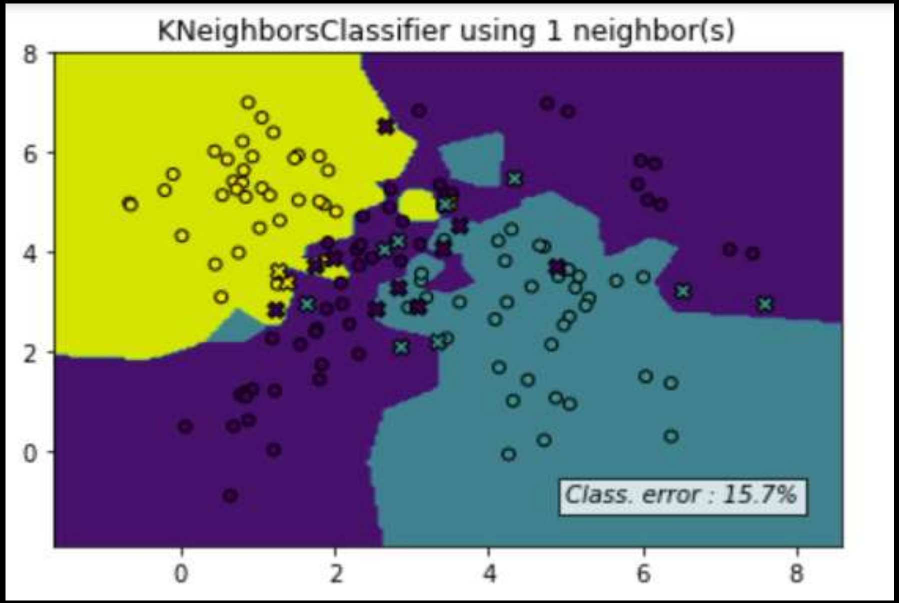
```python
k_arr = [i for i in range(1,11)]
knns = train_knn_classifiers(X_train, y_train, k_arr)
for n,c in zip(knns.keys(), knns.values()):
    plot_knn_classifier(c, n, X_test, y_test)
```

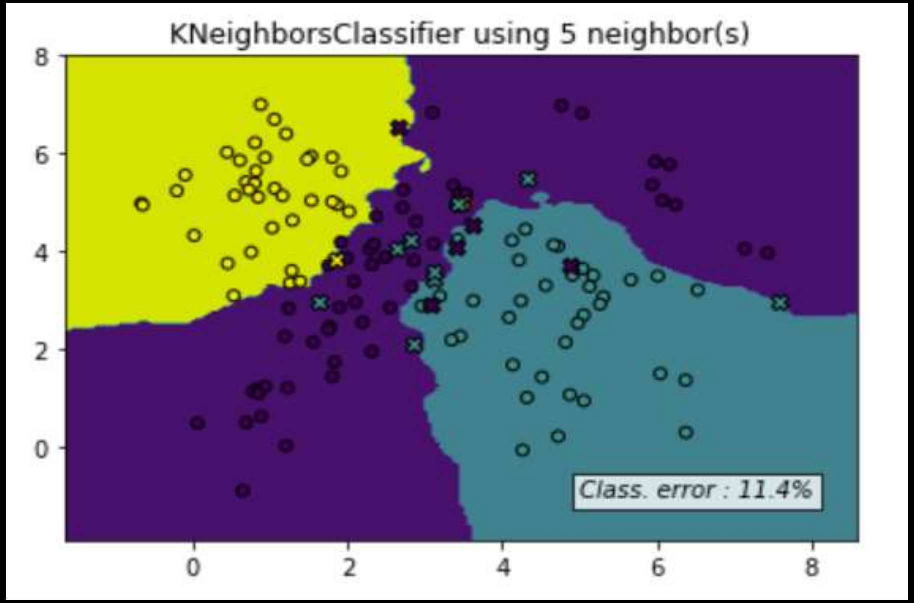
BEST VS WORST CLASSIFIER

Here, we present the best and worst case classifier scenarios , based on the classification error percentage. The best case , is the classifier that uses 5 neighbors with a classification error of 11,4 %. On the other hand , the worst case is the classifier that uses 1 neighbor with a classification error of 15,7 %.

The reason behind this fact , is that we have samples of different classes very close to each other , thus requiring more neighbors to achieve a more precise classification result.

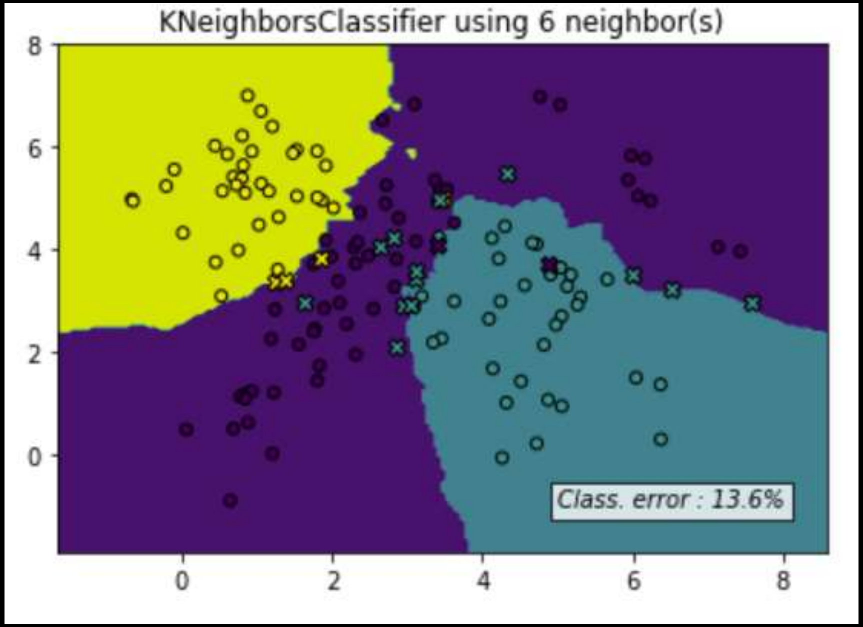
However , as we'll see in the next slide there is a limit in the number of neighbors we can use in order to maximize our classification efficiency.

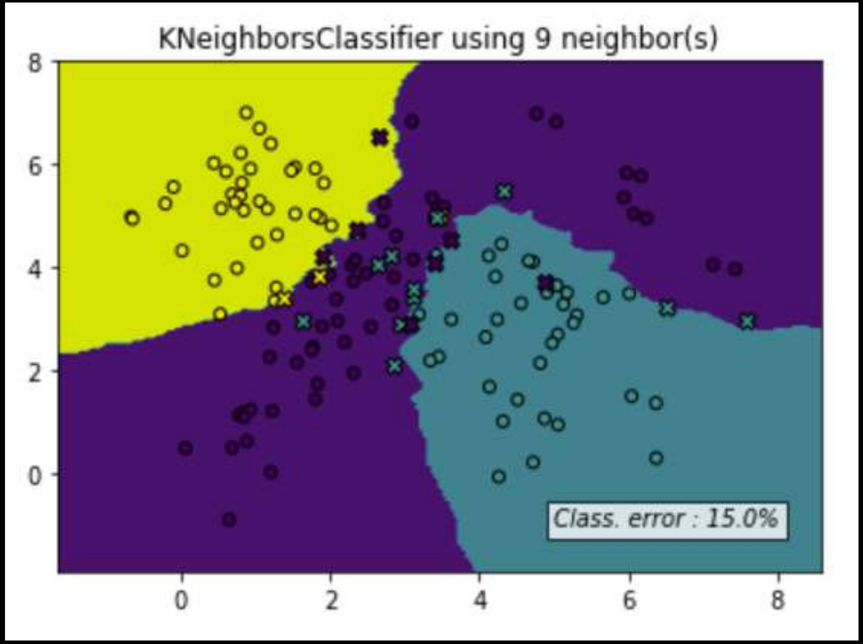


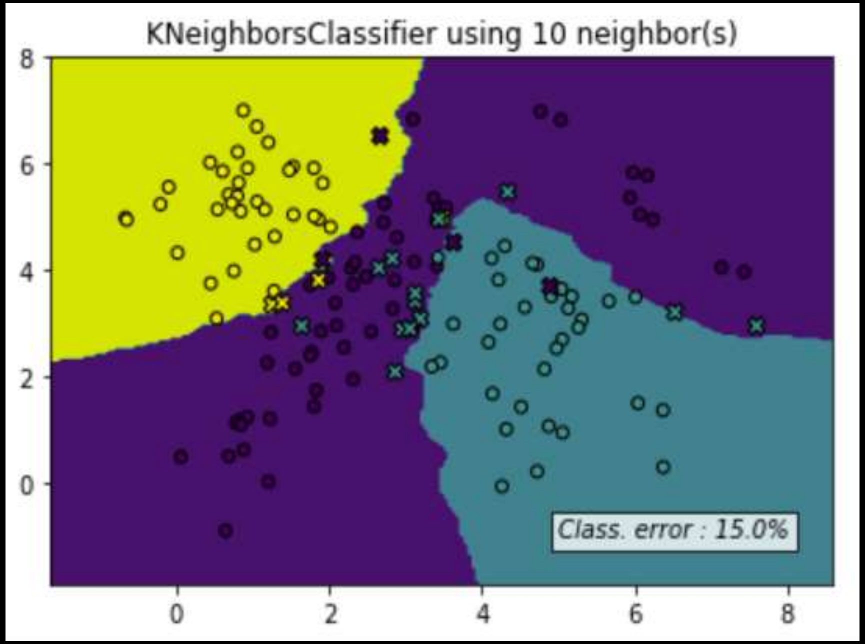


PERFORMANCE PLATEAU

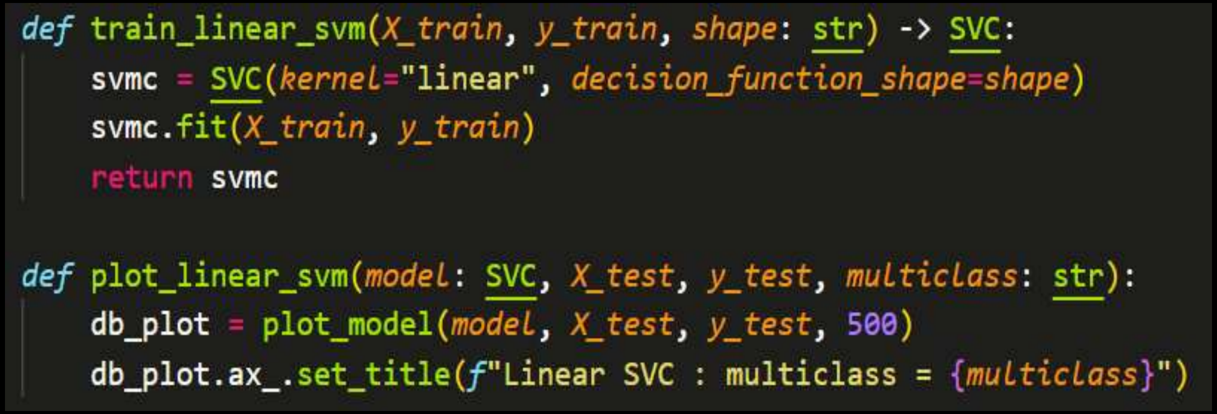
As we mentioned before , the classification efficiency does not always increase together with the number of neighbors that are used. Those pictures show that , after reaching the minimum classification error for the 5-neighbor case , the error starts to increase , before being stabilized after the 9-neighbor case.





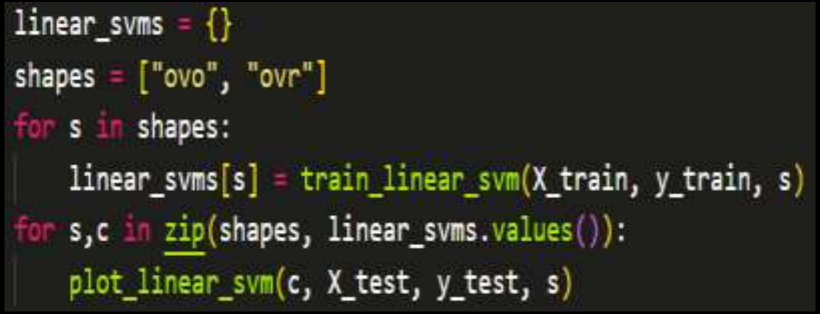


QUESTION C
LINEAR SVM CLASSIFIER



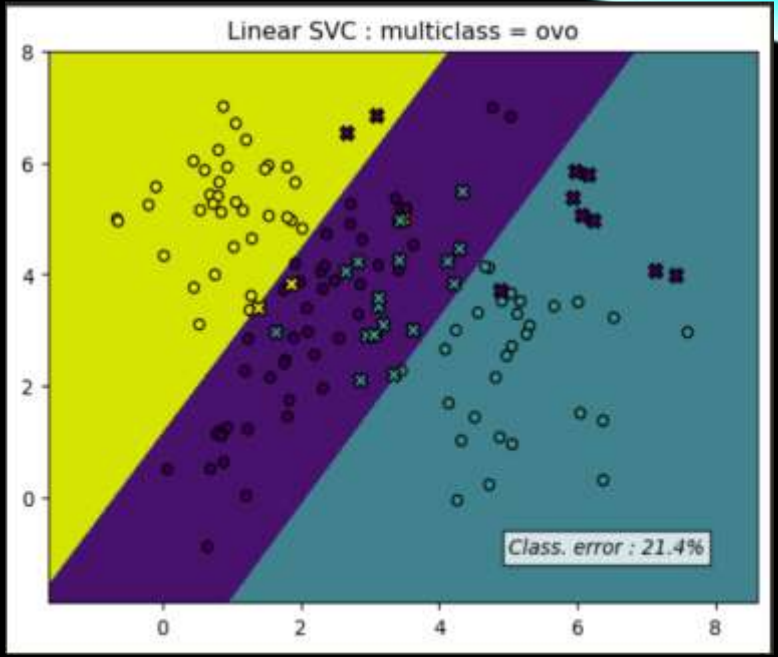
```python
def train_linear_svm(X_train, y_train, shape: str) -> SVC:
    svmc = SVC(kernel="linear", decision_function_shape=shape)
    svmc.fit(X_train, y_train)
    return svmc

def plot_linear_svm(model: SVC, X_test, y_test, multiclass: str):
    db_plot = plot_model(model, X_test, y_test, 500)
    db_plot.ax_.set_title(f"Linear SVC : multiclass = {multiclass}")
```

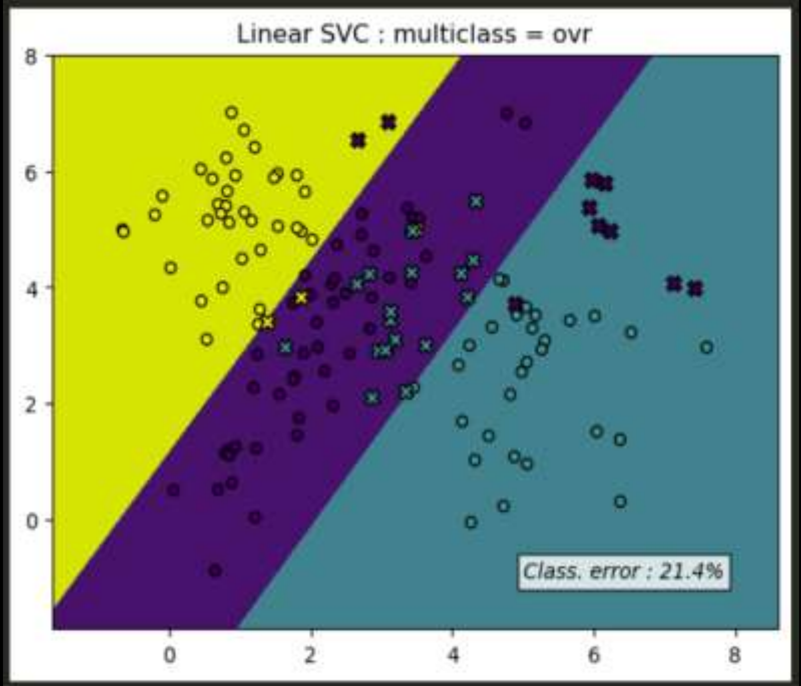
In this question , we are dealing with the creation of a linear SVM classification model , using two different implementation methods : one-vs-one(ovo) and one-vs-rest(ovr). First of all , we create the two functions shown in the top of this slide , one to train the linear svm classifier (using the sklearn function SVC) and the other one to visualize the results of the classification (using the previously created plot_model function).
After that, we are ready to call those functions for our two different shape types (ovo and ovr).



```python
linear_svms = {}
shapes = ["ovo", "ovr"]
for s in shapes:
    linear_svms[s] = train_linear_svm(X_train, y_train, s)
for s,c in zip(shapes, linear_svms.values()):
    plot_linear_svm(c, X_test, y_test, s)
```

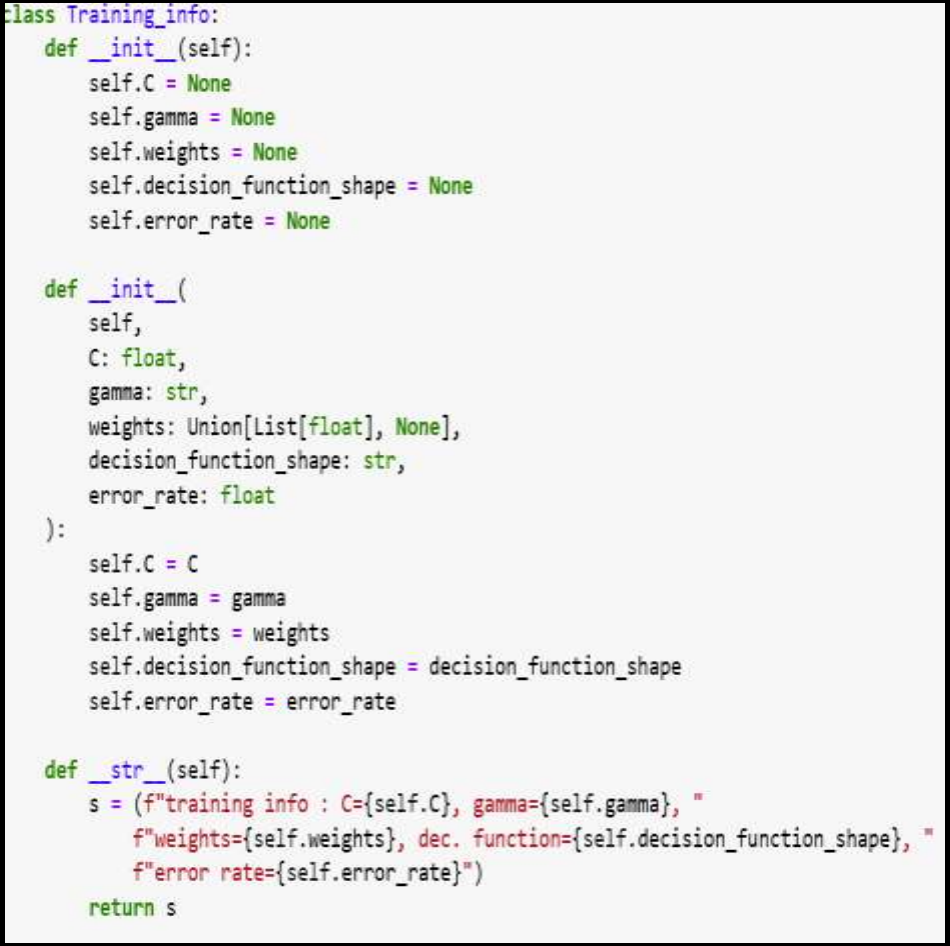
In more detail , about the two Multi-Class Classification methods we use , we can say that :
One-vs-rest, involves splitting the multi-class dataset into multiple binary classification problems. A binary classifier is then trained on each binary classification problem and predictions are made using the model that is the most confident.
Unlike one-vs-rest that splits it into one binary dataset for each class, the one-vs-one approach splits the dataset into one dataset for each class versus every other class.
Despite the fact that, the results shown in this slide are generated using those two different methods for the linear SVC case , we conclude that there are not any differences in them.



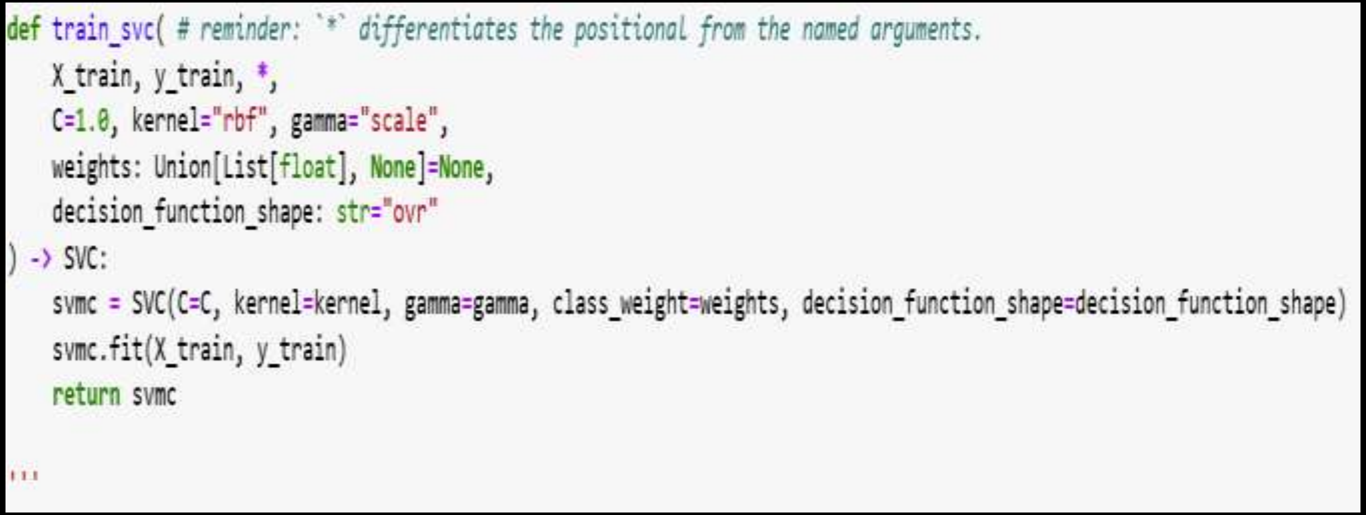


RBF KERNEL SVM CLASSIFIER

Moving forward , we now have to deal with the implementation of an RBF Kernel SVM Classifier. Our general plan in this question, is to generate a grid search algorithm that will help us train and use the classifier with the best metrics , accordingly to the values of some hyperparameters (C, gamma, class weight, decision function shape). First of all , we start by generating the Training_info class which will keep the hyperparameters used to train the best classifier, when grid search is executed. Then, we create the training function (train_svc) in the same way we did in the linear SVM case (using the sklearn function SVC) , but with an rbf kernel and more hyperparameters used .

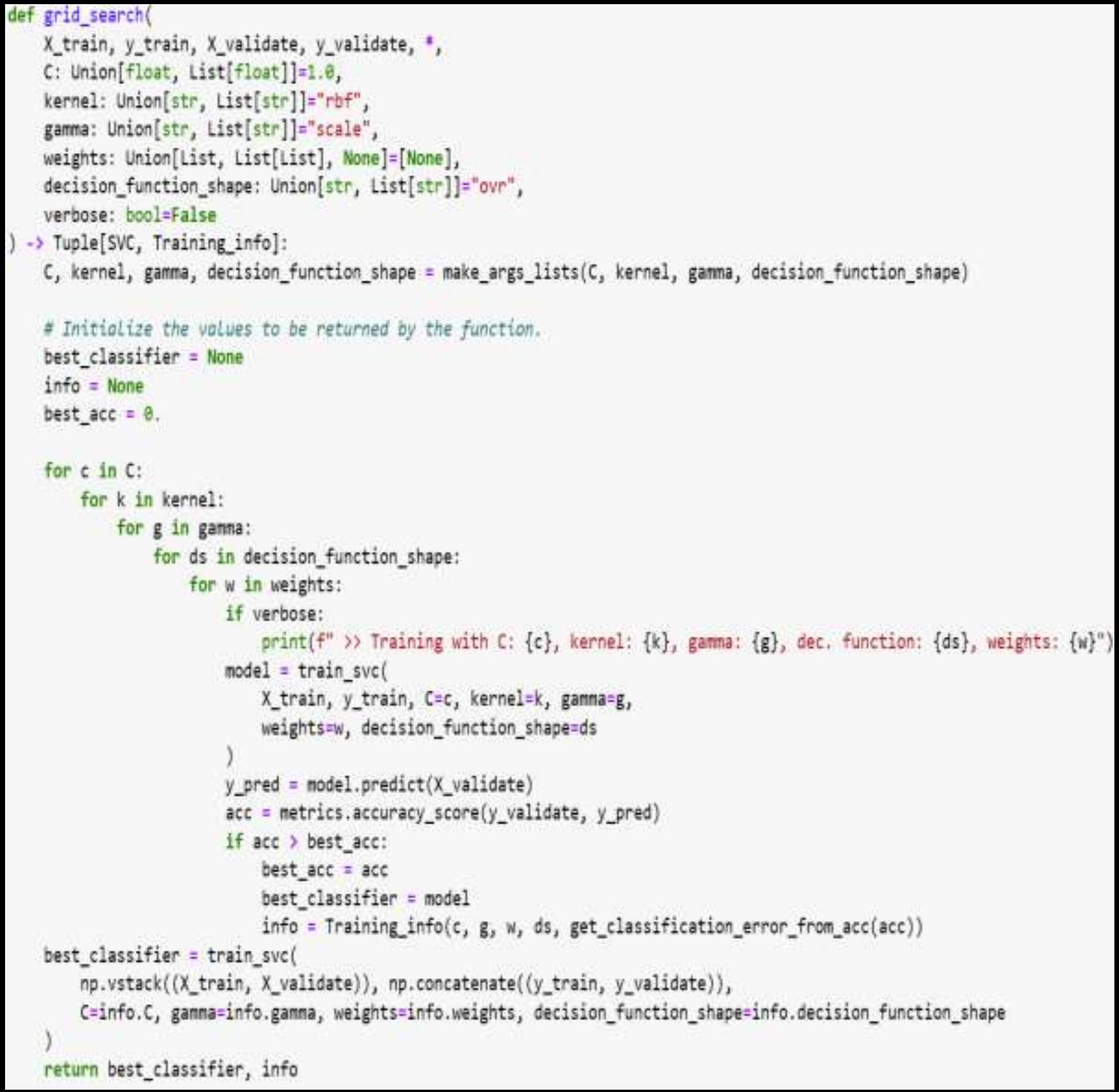


```
lass Training_info:
    def __init__(self):
        self.C = None
        self.gamma = None
        self.weights = None
        self.decision_function_shape = None
        self.error_rate = None

    def __init__(
        self,
        C: float,
        gamma: str,
        weights: Union[List[float], None],
        decision_function_shape: str,
        error_rate: float
    ):
        self.C = C
        self.gamma = gamma
        self.weights = weights
        self.decision_function_shape = decision_function_shape
        self.error_rate = error_rate

    def __str__(self):
        s = (f"training info : C={self.C}, gamma={self.gamma}, "
            f"weights={self.weights}, dec. function={self.decision_function_shape}, "
            f"error rate={self.error_rate}")
        return s
```



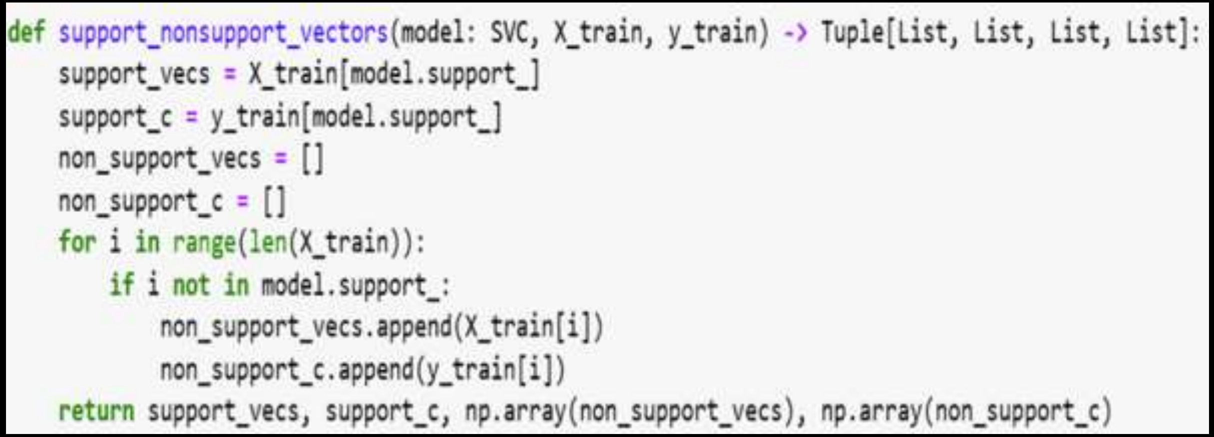
```
def train_svc( # reminder: `*` differentiates the positional from the named arguments.
    X_train, y_train, *,
    C=1.0, kernel="rbf", gamma="scale",
    weights: Union[List[float], None]=None,
    decision_function_shape: str="ovr"
) -> SVC:
    svmc = SVC(C=C, kernel=kernel, gamma=gamma, class_weight=weights, decision_function_shape=decision_function_shape)
    svmc.fit(X_train, y_train)
    return svmc

'''
```

Proceeding , we implement the grid search we talked about in the previous slide. In order to determine which are the optimal parameters for our classifier , then report on its misclassification rate , we split the testing set into two : The validation set and the final testing set. We test each classifier created during grid search using the validation set , then the best model is re-trained utilizing a concatenation of the training and validation data. The held-out testing set is ultimately used , in order to get a final sense of the resulting classifier’s accuracy.

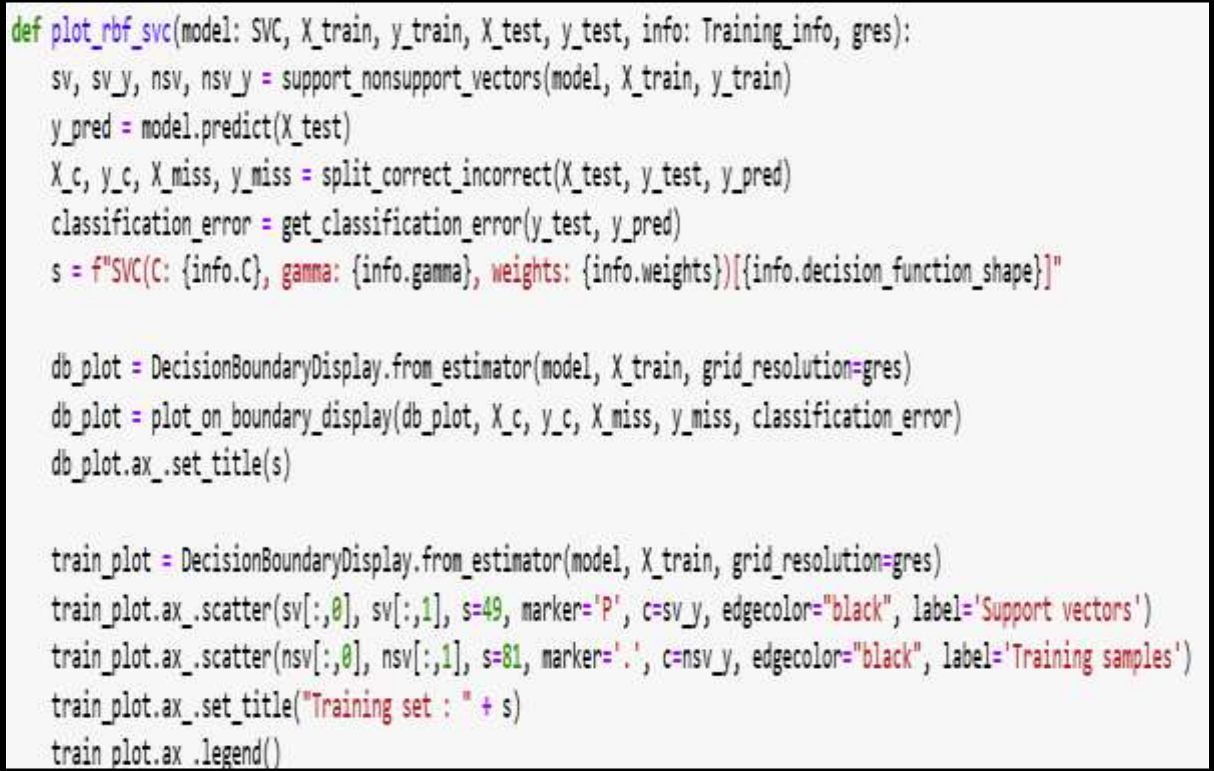


```python
def grid_search(
    X_train, y_train, X_validate, y_validate, *,
    C: Union[float, List[float]]=1.0,
    kernel: Union[str, List[str]]="rbf",
    gamma: Union[str, List[str]]="scale",
    weights: Union[List, List[List], None]=[None],
    decision_function_shape: Union[str, List[str]]="ovr",
    verbose: bool=False
) -> Tuple[SVC, Training_info]:
    C, kernel, gamma, decision_function_shape = make_args_lists(C, kernel, gamma, decision_function_shape)

    # Initialize the values to be returned by the function.
    best_classifier = None
    info = None
    best_acc = 0.

    for c in C:
        for k in kernel:
            for g in gamma:
                for ds in decision_function_shape:
                    for w in weights:
                        if verbose:
                            print(f" >> Training with C: {c}, kernel: {k}, gamma: {g}, dec. function: {ds}, weights: {w}")
                        model = train_svc(
                            X_train, y_train, C=c, kernel=k, gamma=g,
                            weights=w, decision_function_shape=ds
                        )
                        y_pred = model.predict(X_validate)
                        acc = metrics.accuracy_score(y_validate, y_pred)
                        if acc > best_acc:
                            best_acc = acc
                            best_classifier = model
                            info = Training_info(c, g, w, ds, get_classification_error_from_acc(acc))
    best_classifier = train_svc(
        np.vstack((X_train, X_validate)), np.concatenate((y_train, y_validate)),
        C=info.C, gamma=info.gamma, weights=info.weights, decision_function_shape=info.decision_function_shape
    )
    return best_classifier, info
```

Afterwards, we create a simple function to split the training datapoints into support and non-support vectors and we return them together with the corresponding classes. Finally, the last function we create in this question is the one that plots the model's training datapoints, with the support vectors being plotted distinctively. Then , it creates a plot like the previous ones , which distinguishes the correctly and improperly classified datapoints and reports on the classification error rate on the bottom right.



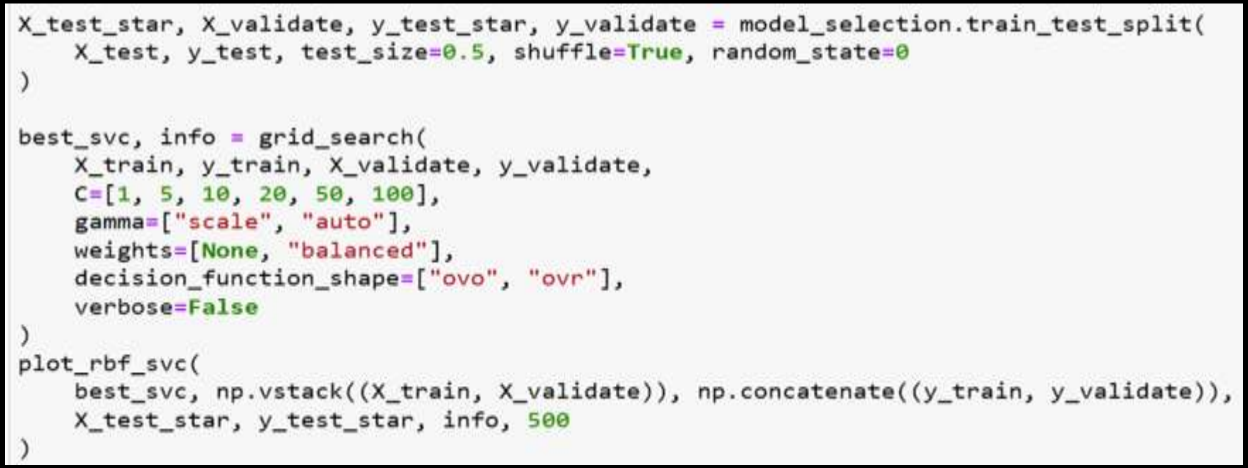
```python
def support_nonsupport_vectors(model: SVC, X_train, y_train) -> Tuple[List, List, List, List]:
    support_vecs = X_train[model.support_]
    support_c = y_train[model.support_]
    non_support_vecs = []
    non_support_c = []
    for i in range(len(X_train)):
        if i not in model.support_:
            non_support_vecs.append(X_train[i])
            non_support_c.append(y_train[i])
    return support_vecs, support_c, np.array(non_support_vecs), np.array(non_support_c)
```



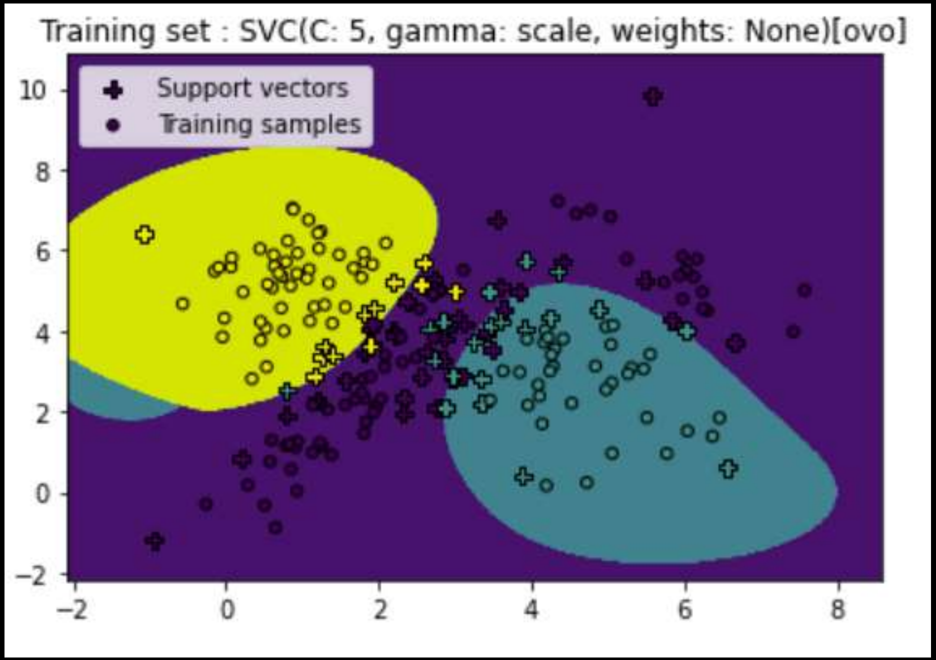
```python
def plot_rbf_svc(model: SVC, X_train, y_train, X_test, y_test, info: Training_info, gres):
    sv, sv_y, nsv, nsv_y = support_nonsupport_vectors(model, X_train, y_train)
    y_pred = model.predict(X_test)
    X_c, y_c, X_miss, y_miss = split_correct_incorrect(X_test, y_test, y_pred)
    classification_error = get_classification_error(y_test, y_pred)
    s = f"SVC(C: {info.C}, gamma: {info.gamma}, weights: {info.weights})[{info.decision_function_shape}]"

    db_plot = DecisionBoundaryDisplay.from_estimator(model, X_train, grid_resolution=gres)
    db_plot = plot_on_boundary_display(db_plot, X_c, y_c, X_miss, y_miss, classification_error)
    db_plot.ax_.set_title(s)

    train_plot = DecisionBoundaryDisplay.from_estimator(model, X_train, grid_resolution=gres)
    train_plot.ax_.scatter(sv[:,0], sv[:,1], s=49, marker='P', c=sv_y, edgecolor="black", label='Support vectors')
    train_plot.ax_.scatter(nsv[:,0], nsv[:,1], s=81, marker='.', c=nsv_y, edgecolor="black", label='Training samples')
    train_plot.ax_.set_title("Training set : " + s)
    train_plot.ax_.legend()
```

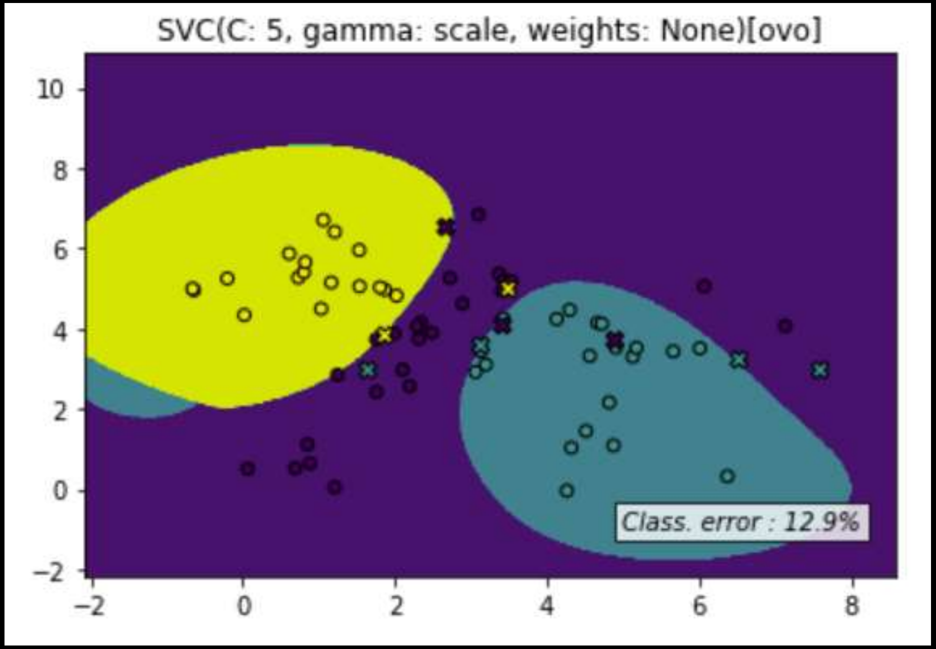
RESULTS

We see that we generate a grid search for some different hyperparameter's values , in order to find and then plot the best svc model, with discrete symbolism for the train sets , the support vectors , the test sets and the improperly classified datapoints as asked.



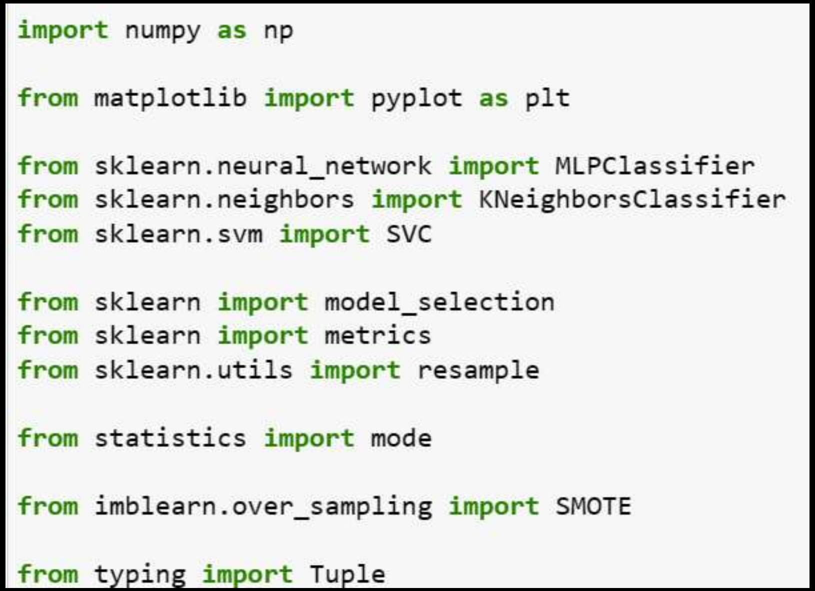
```
X_test_star, X_validate, y_test_star, y_validate = model_selection.train_test_split(
    X_test, y_test, test_size=0.5, shuffle=True, random_state=0
)

best_svc, info = grid_search(
    X_train, y_train, X_validate, y_validate,
    C=[1, 5, 10, 20, 50, 100],
    gamma=["scale", "auto"],
    weights=[None, "balanced"],
    decision_function_shape=["ovo", "ovr"],
    verbose=False
)
plot_rbf_svc(
    best_svc, np.vstack((X_train, X_validate)), np.concatenate((y_train, y_validate)),
    X_test_star, y_test_star, info, 500
)
```





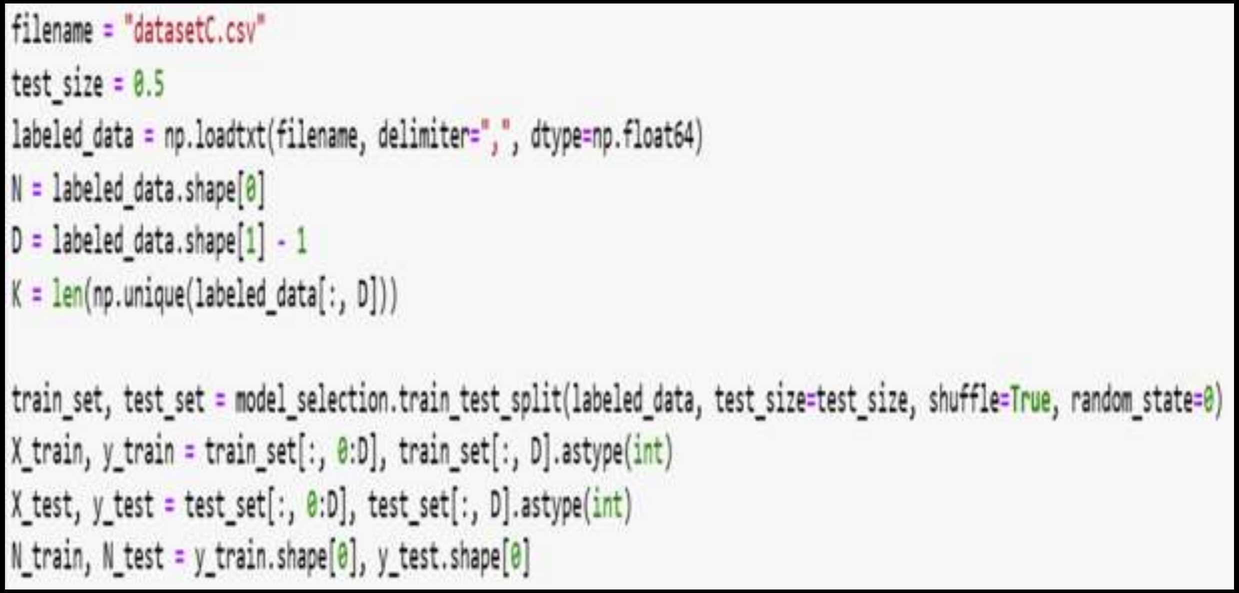
QUESTION D
CUSTOM CLASSIFIER

First of all, we start again by importing all the necessary libraries together with some of their modules to help us : implement classification methods (MLPClassifier , SVC , KNeighborsClassifier) , plot (pyplot) and measure (metrics) classification results, split data into sets (model_selection) and deal with imbalanced datasets (resample, SMOTE) etc.



```
import numpy as np

from matplotlib import pyplot as plt

from sklearn.neural_network import MLPClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC

from sklearn import model_selection
from sklearn import metrics
from sklearn.utils import resample

from statistics import mode

from imblearn.over_sampling import SMOTE

from typing import Tuple
```

After that, we load out data from the "datasetC.csv" file and we split them into the training set (consisted of N samples) and the testing-validating set.



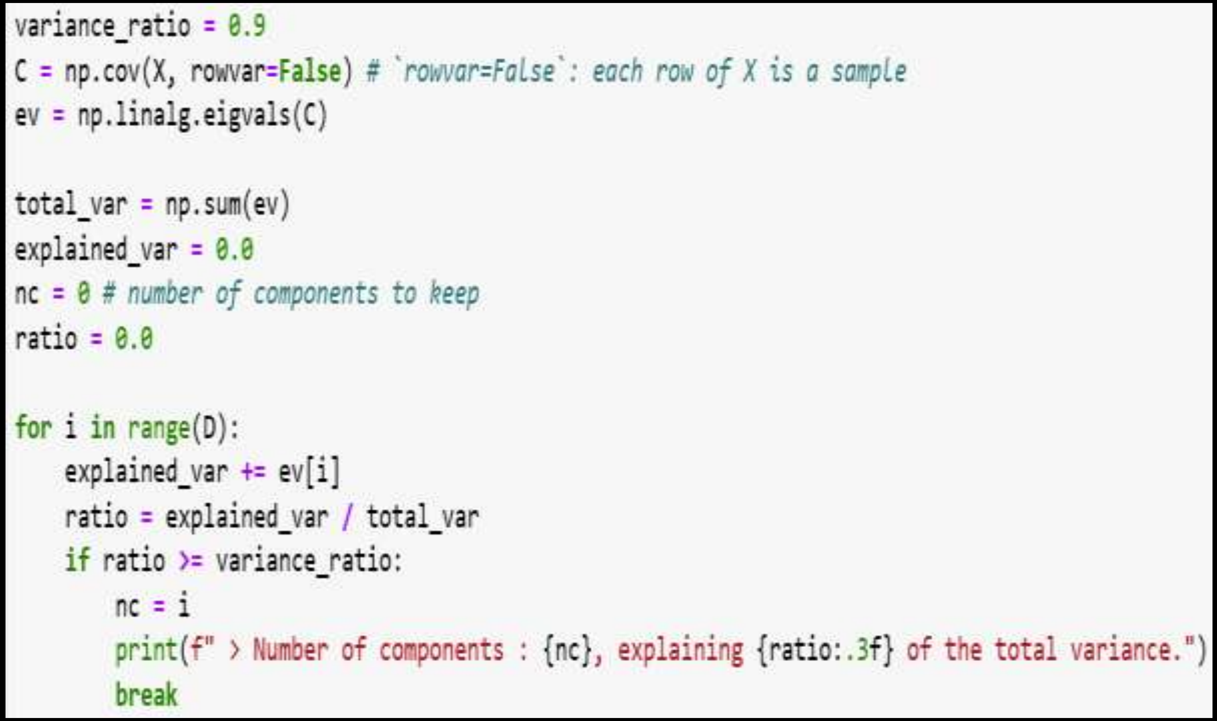
```
filename = "datasetC.csv"
test_size = 0.5
labeled_data = np.loadtxt(filename, delimiter=",", dtype=np.float64)
N = labeled_data.shape[0]
D = labeled_data.shape[1] - 1
K = len(np.unique(labeled_data[:, D]))

train_set, test_set = model_selection.train_test_split(labeled_data, test_size=test_size, shuffle=True, random_state=0)
X_train, y_train = train_set[:, 0:D], train_set[:, D].astype(int)
X_test, y_test = test_set[:, 0:D], test_set[:, D].astype(int)
N_train, N_test = y_train.shape[0], y_test.shape[0]
```

In this question , we are asked to create our own classification algorithm , to achieve the best classification ratio. For this reason , we are going to experiment on multiple classification techniques , in order to find the best one. The techniques we ended up testing are :

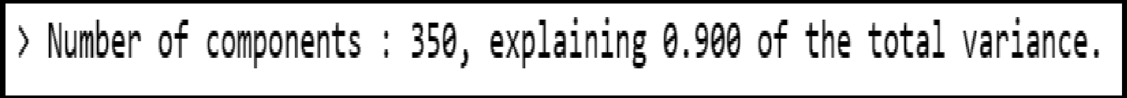
1) MLP
2) K Neighbors Classifier
3) SVC

PCA TESTING

We start our tests , by checking if we could take advantage of the Principal Component analysis , in order to reduce the size of our dataset without losing any important information about it. We chose to check how many components we have to keep , in order to maintain 90% of the explained variance. The number that comes up is too high (350 out of 400) to bother transforming our data. The reason behind this big number , is that the explained variance is equally distributed amongst almost all features.
So, we have no concrete way to visualize the data.



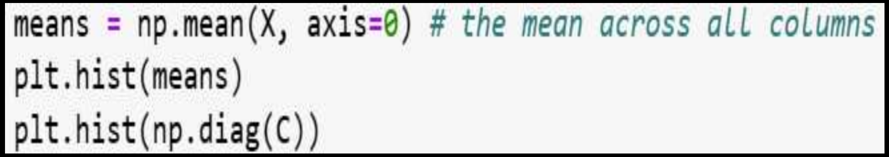
```
variance_ratio = 0.9
C = np.cov(X, rowvar=False) # `rowvar=False`: each row of X is a sample
ev = np.linalg.eigvals(C)

total_var = np.sum(ev)
explained_var = 0.0
nc = 0 # number of components to keep
ratio = 0.0

for i in range(D):
    explained_var += ev[i]
    ratio = explained_var / total_var
    if ratio >= variance_ratio:
        nc = i
        print(f" > Number of components : {nc}, explaining {ratio:.3f} of the total variance.")
        break
```



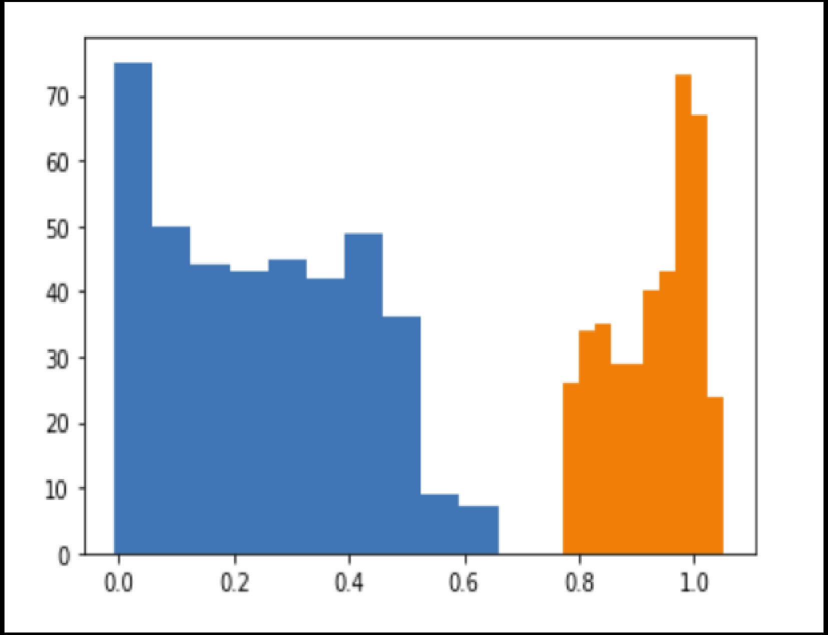
```
> Number of components : 350, explaining 0.900 of the total variance.
```

SCALING TESTING

Now , we are going to check whether we can improve our data by scaling or normalizing it. So , we will check the means and variances of all features by using the hist function of the matplotlib library , to create a histogram. The histogram that comes up and can be seen in this slide , shows that all features are in the same order of magnitude and thus no scaling needs to be done.

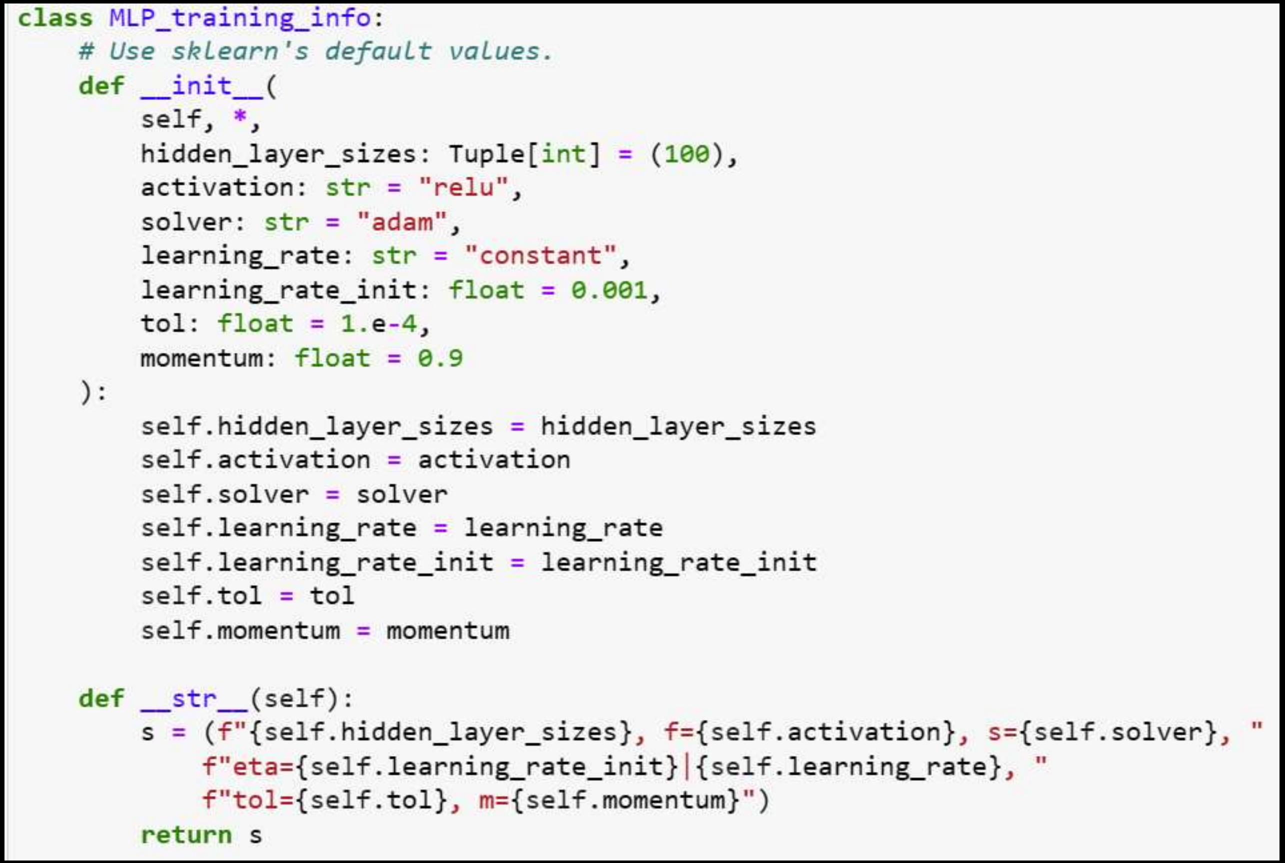


```
means = np.mean(X, axis=0) # the mean across all columns
plt.hist(means)
plt.hist(np.diag(C))
```



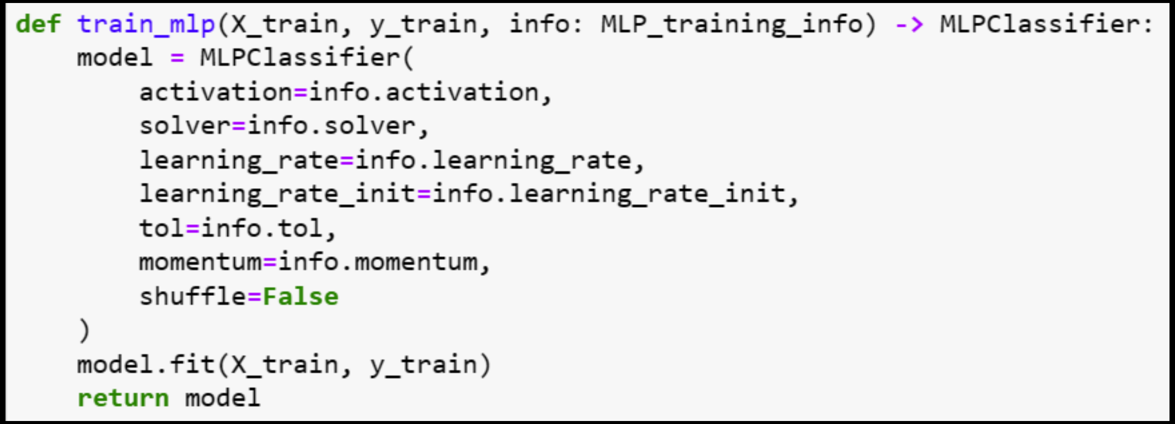
MLP TESTING

Moving on , we are going to test the performance of an MLP Classifier to our task. We start by creating a class like the one we used in question C , in order to store information on the hyperparameters that will prove to be the best after a typical grid search.

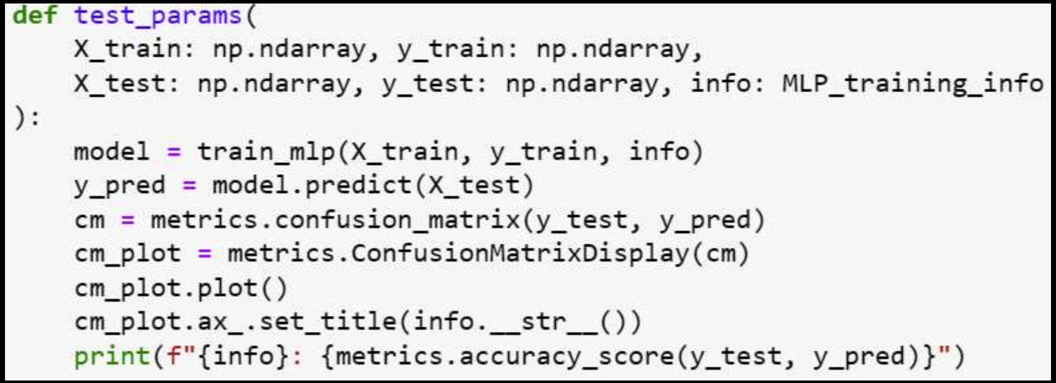


```
class MLP_training_info:
    # Use sklearn's default values.
    def __init__(
        self, *,
        hidden_layer_sizes: Tuple[int] = (100),
        activation: str = "relu",
        solver: str = "adam",
        learning_rate: str = "constant",
        learning_rate_init: float = 0.001,
        tol: float = 1.e-4,
        momentum: float = 0.9
    ):
        self.hidden_layer_sizes = hidden_layer_sizes
        self.activation = activation
        self.solver = solver
        self.learning_rate = learning_rate
        self.learning_rate_init = learning_rate_init
        self.tol = tol
        self.momentum = momentum

    def __str__(self):
        s = (f"{self.hidden_layer_sizes}, f={self.activation}, s={self.solver}, "
            f"eta={self.learning_rate_init}|{self.learning_rate}, "
            f"tol={self.tol}, m={self.momentum}")
        return s
```

Then , we continue by creating two functions : the train_mlp and the test_params , that will help us carry on the MLP tests. The first one , uses the custom MLP training parameter class in order to pass the values
to sklearn's MLPClassifier module. The second one , trains an MLP using the parameters stored in info (object of the class we use in our tests) , displays the confusion matrix and the accuracy of the model,
along with the utilized parameters.



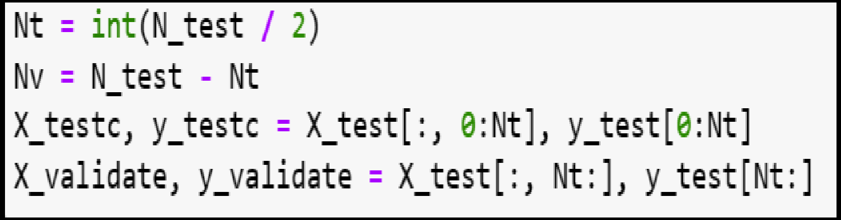
```
def train_mlp(X_train, y_train, info: MLP_training_info) -> MLPClassifier:
    model = MLPClassifier(
        activation=info.activation,
        solver=info.solver,
        learning_rate=info.learning_rate,
        learning_rate_init=info.learning_rate_init,
        tol=info.tol,
        momentum=info.momentum,
        shuffle=False
    )
    model.fit(X_train, y_train)
    return model
```



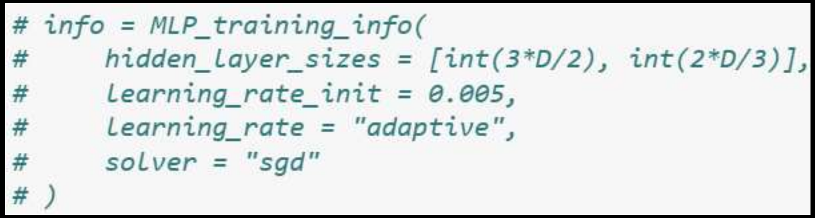
```
def test_params(
    X_train: np.ndarray, y_train: np.ndarray,
    X_test: np.ndarray, y_test: np.ndarray, info: MLP_training_info
):
    model = train_mlp(X_train, y_train, info)
    y_pred = model.predict(X_test)
    cm = metrics.confusion_matrix(y_test, y_pred)
    cm_plot = metrics.ConfusionMatrixDisplay(cm)
    cm_plot.plot()
    cm_plot.ax_.set_title(info.__str__())
    print(f"{info}: {metrics.accuracy_score(y_test, y_pred)}")
```

Finally , before starting our tests , we split the testing dataset into two equal parts. The first half , remains the testing data and the second half will be used for validation.

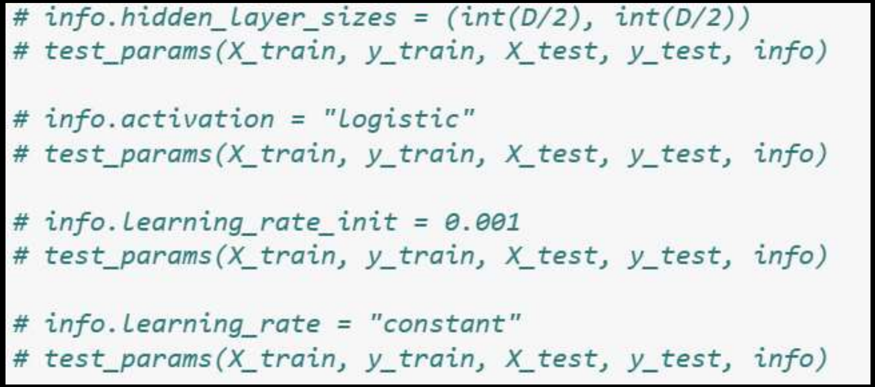
Then , we create an object of the MLP_training_info class called info (as we mentioned in the previous slide) and we experiment by changing some of the hyperparameter's values to see how the classification goes.



```
Nt = int(N_test / 2)
Nv = N_test - Nt
X_testc, y_testc = X_test[:, 0:Nt], y_test[0:Nt]
X_validate, y_validate = X_test[:, Nt:], y_test[Nt:]
```



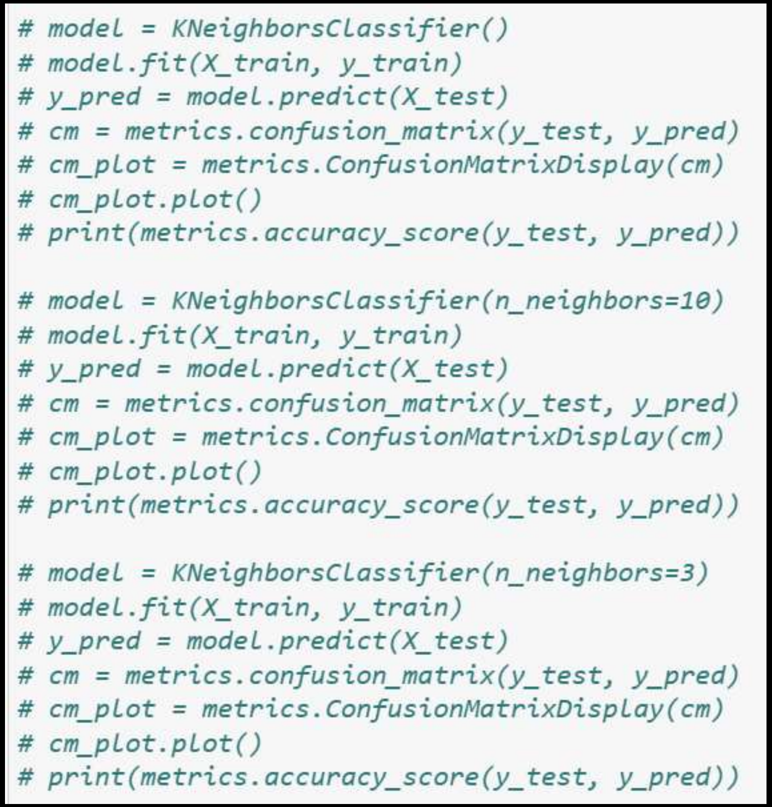
```
# info = MLP_training_info(
#     hidden_layer_sizes = [int(3*D/2), int(2*D/3)],
#     learning_rate_init = 0.005,
#     learning_rate = "adaptive",
#     solver = "sgd"
# )
```



```
# info.hidden_layer_sizes = (int(D/2), int(D/2))
# test_params(X_train, y_train, X_test, y_test, info)

# info.activation = "logistic"
# test_params(X_train, y_train, X_test, y_test, info)

# info.learning_rate_init = 0.001
# test_params(X_train, y_train, X_test, y_test, info)

# info.learning_rate = "constant"
# test_params(X_train, y_train, X_test, y_test, info)
```

K NEIGBORS CLASSIFIER TESTING

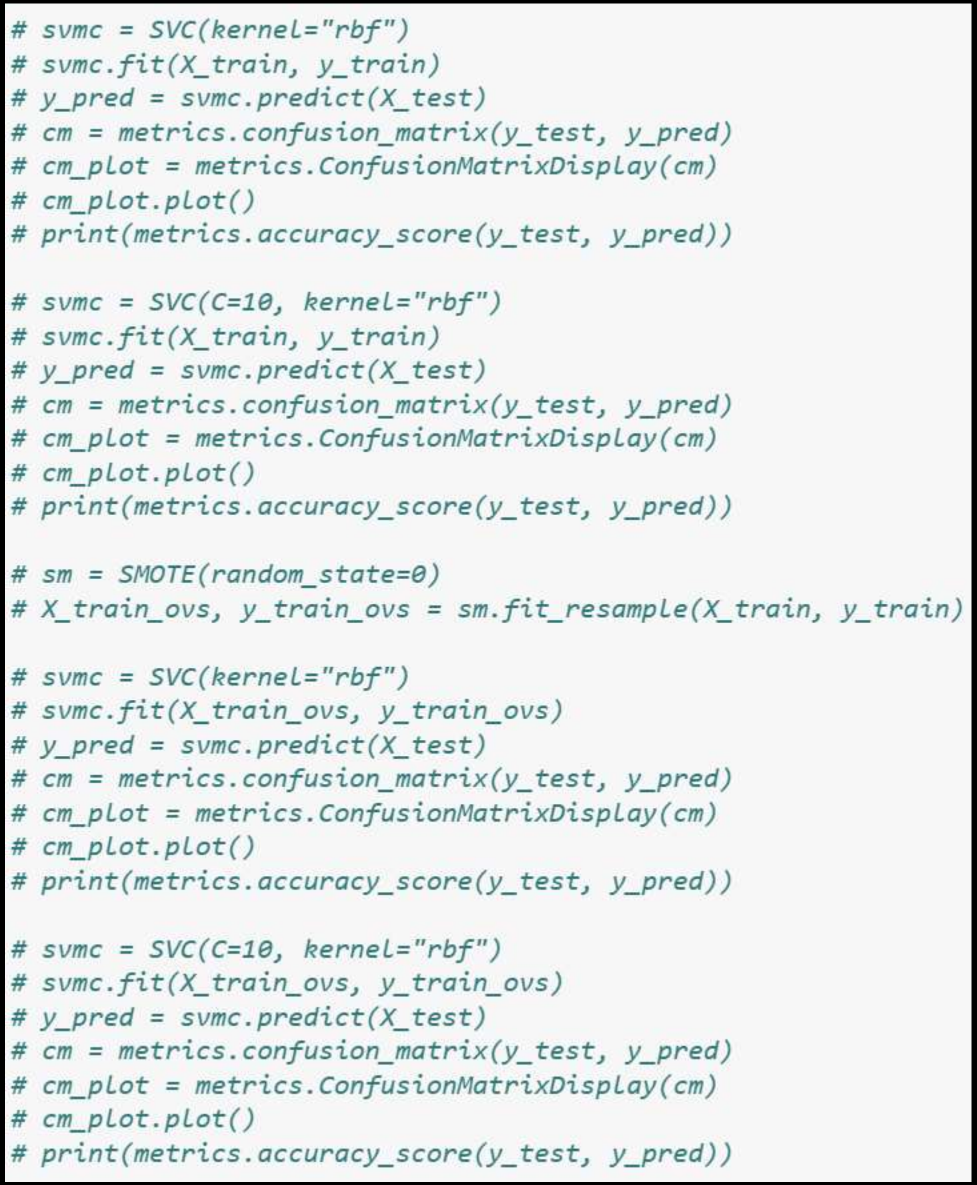
The next classification approach that we applied , is the K-Neighbors Classifier. As we see in this picture , we implemented this method for various values of K and we gathered the confusion matrices and accuracy scores for each case.



```
# model = KNeighborsClassifier()
# model.fit(X_train, y_train)
# y_pred = model.predict(X_test)
# cm = metrics.confusion_matrix(y_test, y_pred)
# cm_plot = metrics.ConfusionMatrixDisplay(cm)
# cm_plot.plot()
# print(metrics.accuracy_score(y_test, y_pred))

# model = KNeighborsClassifier(n_neighbors=10)
# model.fit(X_train, y_train)
# y_pred = model.predict(X_test)
# cm = metrics.confusion_matrix(y_test, y_pred)
# cm_plot = metrics.ConfusionMatrixDisplay(cm)
# cm_plot.plot()
# print(metrics.accuracy_score(y_test, y_pred))

# model = KNeighborsClassifier(n_neighbors=3)
# model.fit(X_train, y_train)
# y_pred = model.predict(X_test)
# cm = metrics.confusion_matrix(y_test, y_pred)
# cm_plot = metrics.ConfusionMatrixDisplay(cm)
# cm_plot.plot()
# print(metrics.accuracy_score(y_test, y_pred))
```

SVM TESTING

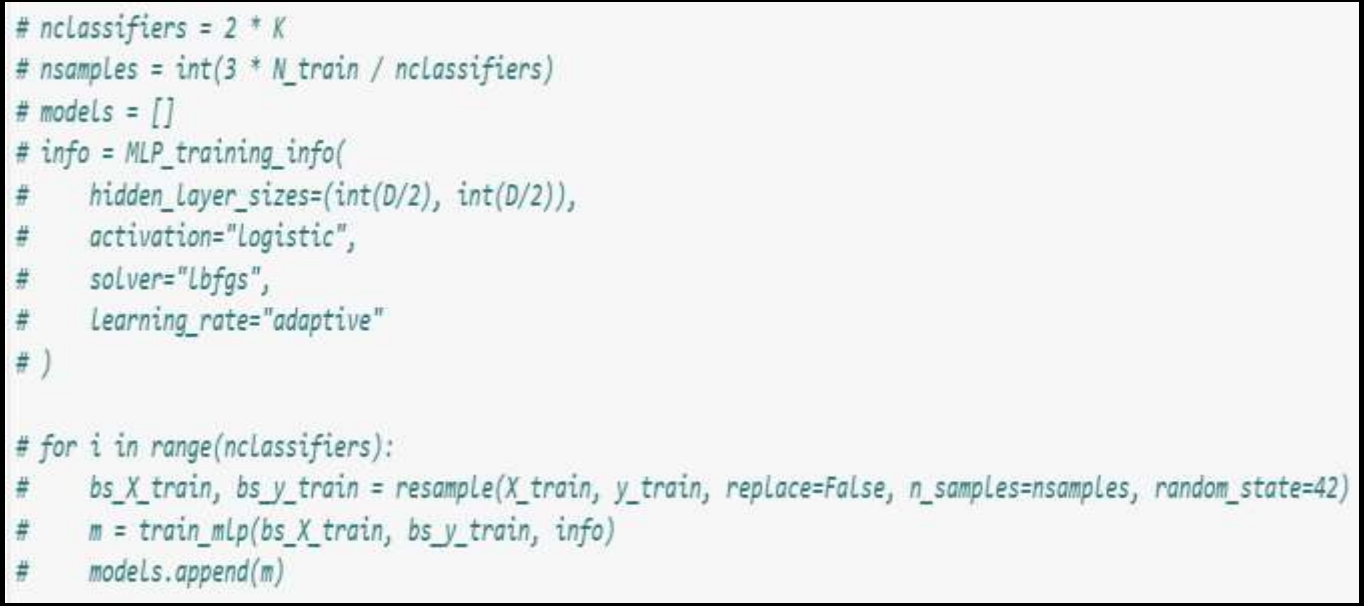
Progressing , we are also trying to use an SVC Classifier on our data. We try several SVC cases , all of them with an rbf kernel and some of them after implementing SMOTE to tacke any possible imbalance in our dataset. We generate and compare the confusion matrices and accuracy scores of each case.



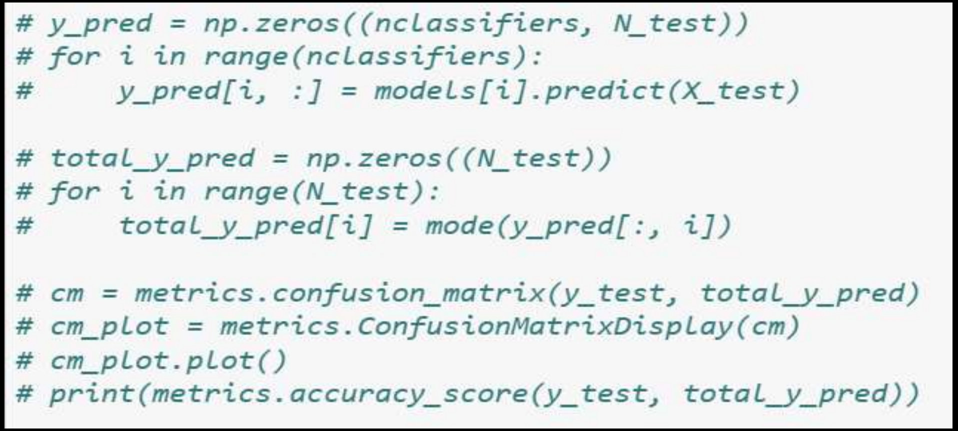
```
# svmc = SVC(kernel="rbf")
# svmc.fit(X_train, y_train)
# y_pred = svmc.predict(X_test)
# cm = metrics.confusion_matrix(y_test, y_pred)
# cm_plot = metrics.ConfusionMatrixDisplay(cm)
# cm_plot.plot()
# print(metrics.accuracy_score(y_test, y_pred))

# svmc = SVC(C=10, kernel="rbf")
# svmc.fit(X_train, y_train)
# y_pred = svmc.predict(X_test)
# cm = metrics.confusion_matrix(y_test, y_pred)
# cm_plot = metrics.ConfusionMatrixDisplay(cm)
# cm_plot.plot()
# print(metrics.accuracy_score(y_test, y_pred))

# sm = SMOTE(random_state=0)
# X_train_ovs, y_train_ovs = sm.fit_resample(X_train, y_train)

# svmc = SVC(kernel="rbf")
# svmc.fit(X_train_ovs, y_train_ovs)
# y_pred = svmc.predict(X_test)
# cm = metrics.confusion_matrix(y_test, y_pred)
# cm_plot = metrics.ConfusionMatrixDisplay(cm)
# cm_plot.plot()
# print(metrics.accuracy_score(y_test, y_pred))

# svmc = SVC(C=10, kernel="rbf")
# svmc.fit(X_train_ovs, y_train_ovs)
# y_pred = svmc.predict(X_test)
# cm = metrics.confusion_matrix(y_test, y_pred)
# cm_plot = metrics.ConfusionMatrixDisplay(cm)
# cm_plot.plot()
# print(metrics.accuracy_score(y_test, y_pred))
```

RESAMPLE TESTING

The last test we are conducting , is to train a number of 2*K (K is the number of classes in the dataset) classifiers using resampling.



```
# nclassifiers = 2 * K
# nsamples = int(3 * N_train / nclassifiers)
# models = []
# info = MLP_training_info(
#     hidden_layer_sizes=(int(D/2), int(D/2)),
#     activation="logistic",
#     solver="lbfgs",
#     learning_rate="adaptive"
# )

# for i in range(nclassifiers):
#     bs_X_train, bs_y_train = resample(X_train, y_train, replace=False, n_samples=nsamples, random_state=42)
#     m = train_mlp(bs_X_train, bs_y_train, info)
#     models.append(m)
```

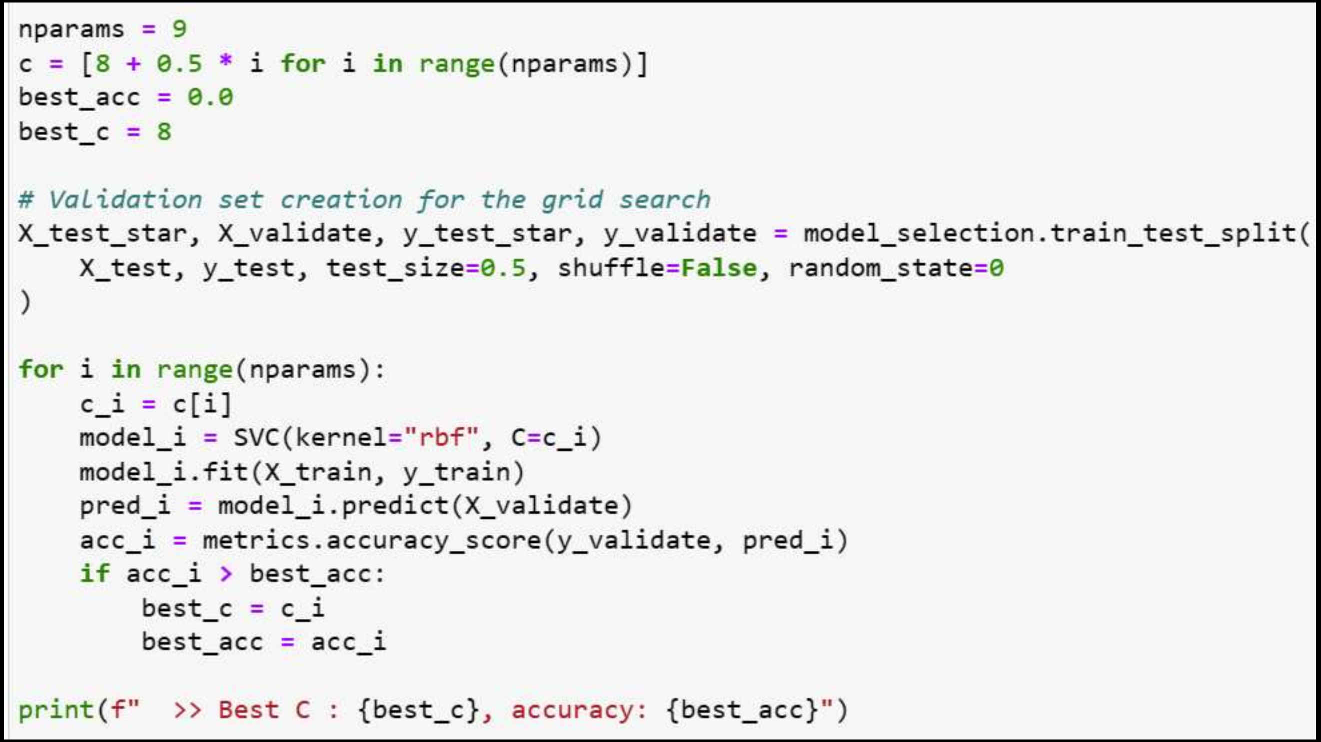


```
# y_pred = np.zeros((nclassifiers, N_test))
# for i in range(nclassifiers):
#     y_pred[i, :] = models[i].predict(X_test)

# total_y_pred = np.zeros((N_test))
# for i in range(N_test):
#     total_y_pred[i] = mode(y_pred[:, i])

# cm = metrics.confusion_matrix(y_test, total_y_pred)
# cm_plot = metrics.ConfusionMatrixDisplay(cm)
# cm_plot.plot()
# print(metrics.accuracy_score(y_test, total_y_pred))
```

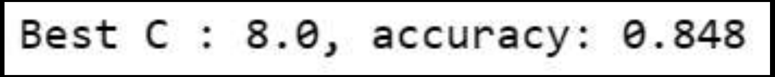
RESULTS

After all those different methods that we tried , the best accuracy score we get is 0.8456 and it comes with an SVC Classifier with rbf kernel and c = 10.

However , in order to be sure that we achieve the maximum accuracy score possible , we will conduct a final grid search around the neighborhood of the best value for C. After using the validation set for our search , we conclude that the best C is 8.0 with an accuracy score of 0.848.

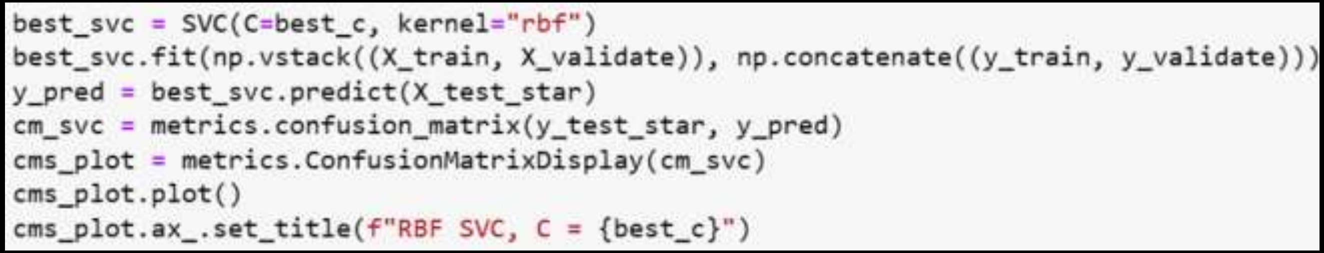


```
nparams = 9
c = [8 + 0.5 * i for i in range(nparams)]
best_acc = 0.0
best_c = 8

# Validation set creation for the grid search
X_test_star, X_validate, y_test_star, y_validate = model_selection.train_test_split(
    X_test, y_test, test_size=0.5, shuffle=False, random_state=0
)

for i in range(nparams):
    c_i = c[i]
    model_i = SVC(kernel="rbf", C=c_i)
    model_i.fit(X_train, y_train)
    pred_i = model_i.predict(X_validate)
    acc_i = metrics.accuracy_score(y_validate, pred_i)
    if acc_i > best_acc:
        best_c = c_i
        best_acc = acc_i

print(f"  >> Best C : {best_c}, accuracy: {best_acc}")
```

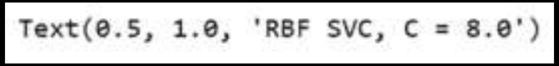


```
Best C : 8.0, accuracy: 0.848
```

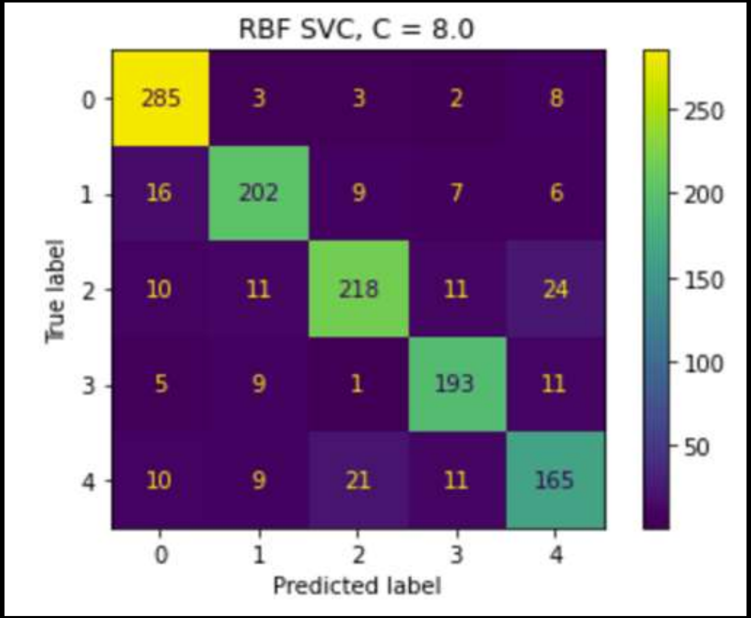
Now , we just have to create this optimal SVC Classifier (kernel = ‘rbf’ , C = 8) and generate its confusion matrix for the validation data.



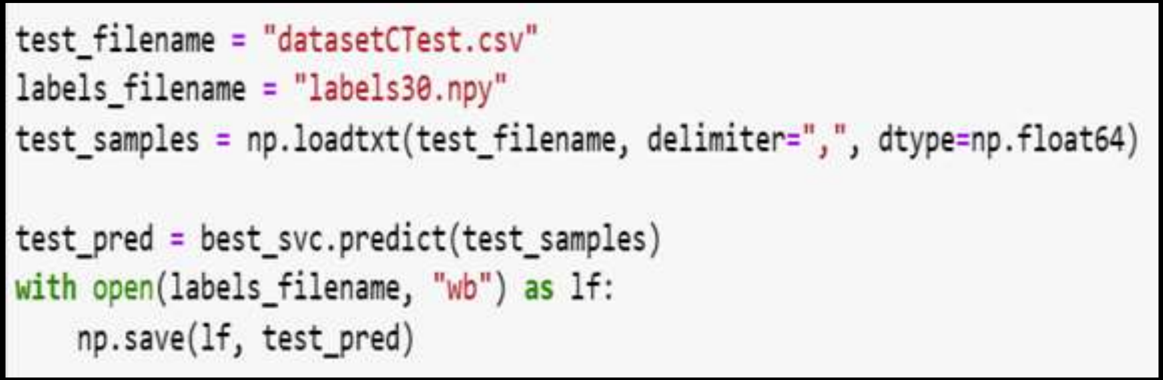
```
best_svc = SVC(C=best_c, kernel="rbf")
best_svc.fit(np.vstack((X_train, X_validate)), np.concatenate((y_train, y_validate)))
y_pred = best_svc.predict(X_test_star)
cm_svc = metrics.confusion_matrix(y_test_star, y_pred)
cms_plot = metrics.ConfusionMatrixDisplay(cm_svc)
cms_plot.plot()
cms_plot.ax_.set_title(f"RBF SVC, C = {best_c}")
```



```
Text(0.5, 1.0, 'RBF SVC, C = 8.0')
```



Last but not least , we predict the values for datasetCTest.csv and store them in a .npy file , names “labels30.npy” , as asked in this question.



```
test_filename = "datasetCTest.csv"
labels_filename = "labels30.npy"
test_samples = np.loadtxt(test_filename, delimiter=",", dtype=np.float64)

test_pred = best_svc.predict(test_samples)
with open(labels_filename, "wb") as lf:
    np.save(lf, test_pred)
```

THANK YOU SO MUCH FOR YOUR ATTENTION