Pràctica 2 - Clasificació

github grup 305

En aquesta pràctica haurem d'analitzar una base de dades i intentra veure quin o quins poden ser els millors classificadors després d'haver fet un anàlisi de les dades.

Apartat (B): Classificació numèrica

EDA expolratory data analysis

En aquest apartat farem un anàlisi de la nostra base de dates, la qual consisteix en, apartir de un seguit de sensors ubicats a diferents músculs, dir quin és el gest que esta articulant amb la mà.

```
import pandas as pd
import numpy as np

dataset = []

# llegim els quatre fitxes que tenim amb totes les dades
for i in range(4):
    dataset.append(pd.read_csv(f"dataset/{i}.csv", header=None))

dataset = pd.concat(dataset, axis=0)
data = dataset.values

print("Forma de la base de dades:", dataset.shape)
```

```
 Forma de la base de dades: (11678, 65)
```

Veiem que el nostre dataset es força gran, ja que no és només que compti amb 11.678 mostres, sinó que també té 65 atribust, dels quals 64 son sensors (8 sensors per cada múscul) i l'últim és el gest que fa la mà. Aquest és un atribut categòric que està composat per 4 classes: 0 : pedra, 1 : paper, 2 : tisores i 3 : ok

```
# afegim noms a les columnes per ubicar millor els valors
cols = {}

for i in range(64):
    cols[i] = f"m{int(i/8)+1}_s{i%8+1}"
cols[64] = "gesture_class"

dataset = dataset.rename(columns=cols)
dataset.describe()
```

	m1_s1	m1_s2	m1_s3	m1_s4	m1_s5	m1_s6	m1_s7	m1_s8
count	11678.000000	11678.000000	11678.000000	11678.000000	11678.000000	11678.000000	11678.000000	11678.0
mean	-0.520380	-0.726837	-0.739082	-0.729748	-0.159103	-0.554890	-1.272649	-0.661
std	18.566709	11.766878	4.989944	7.441675	17.850402	25.809528	25.089972	15.408
min	-116.000000	-104.000000	-33.000000	-75.000000	-121.000000	-122.000000	-128.000000	-128.0
25%	-9.000000	-4.000000	-3.000000	-4.000000	-10.000000	-15.000000	-6.000000	-8.000
50%	-1.000000	-1.000000	-1.000000	-1.000000	0.000000	-1.000000	-1.000000	-1.000
75%	7.000000	3.000000	2.000000	3.000000	10.000000	13.000000	4.000000	6.000

	m1_s1	m1_s2	m1_s3	m1_s4	m1_s5	m1_s6	m1_s7	m1_s8
max	111.000000	90.000000	34.000000	55.000000	92.000000	127.000000	127.000000	126.00

8 rows × 65 columns

Podem observar que la distribució dels valors és bastant similar entre els atributs. Tots els sensors tenen mitjanes molt similars i els percentils bastant semblants en quant als valors a més de que tots tenen la centralització en els mateixos valors.

En quant als gestes de la mà, aquest també semblen que tenim casi la mateixa quantitat de cada gest. Per assegurar-nos, ho comprovarem:

```
np.unique(data[:,-1], return_counts=True)
```

```
(array([0., 1., 2., 3.]), array([2910, 2903, 2943, 2922], dtype=int64))
```

Podem dir doncs que cada gest és representat en la mateixa mesura que els altres, tot i que hi hagi una diferencia màxima de 40 mostres més en la classe que menys i més en té.

```
for i in range(65):
    if dataset.isnull().sum()[i] != 0:
        print(f"{i} column is null with {dataset.isnull().sum()[i]} nulls")
```

Veiem que la nostra dataset tampoc té valors nulls, cosa que facilita el seu tractament.

```
type(data[0,0])
```

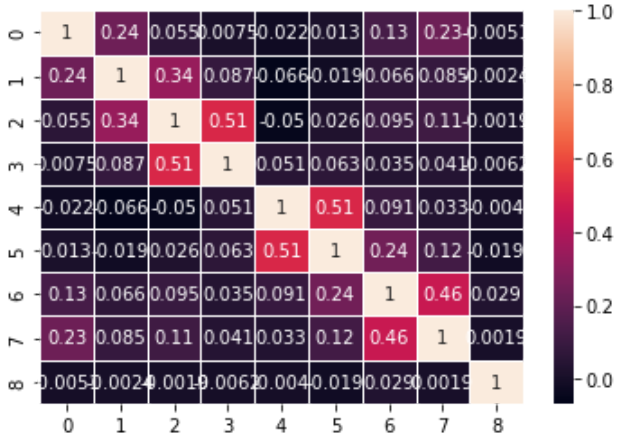
```
numpy.float64
```

```
type(data[-1, -1])
```

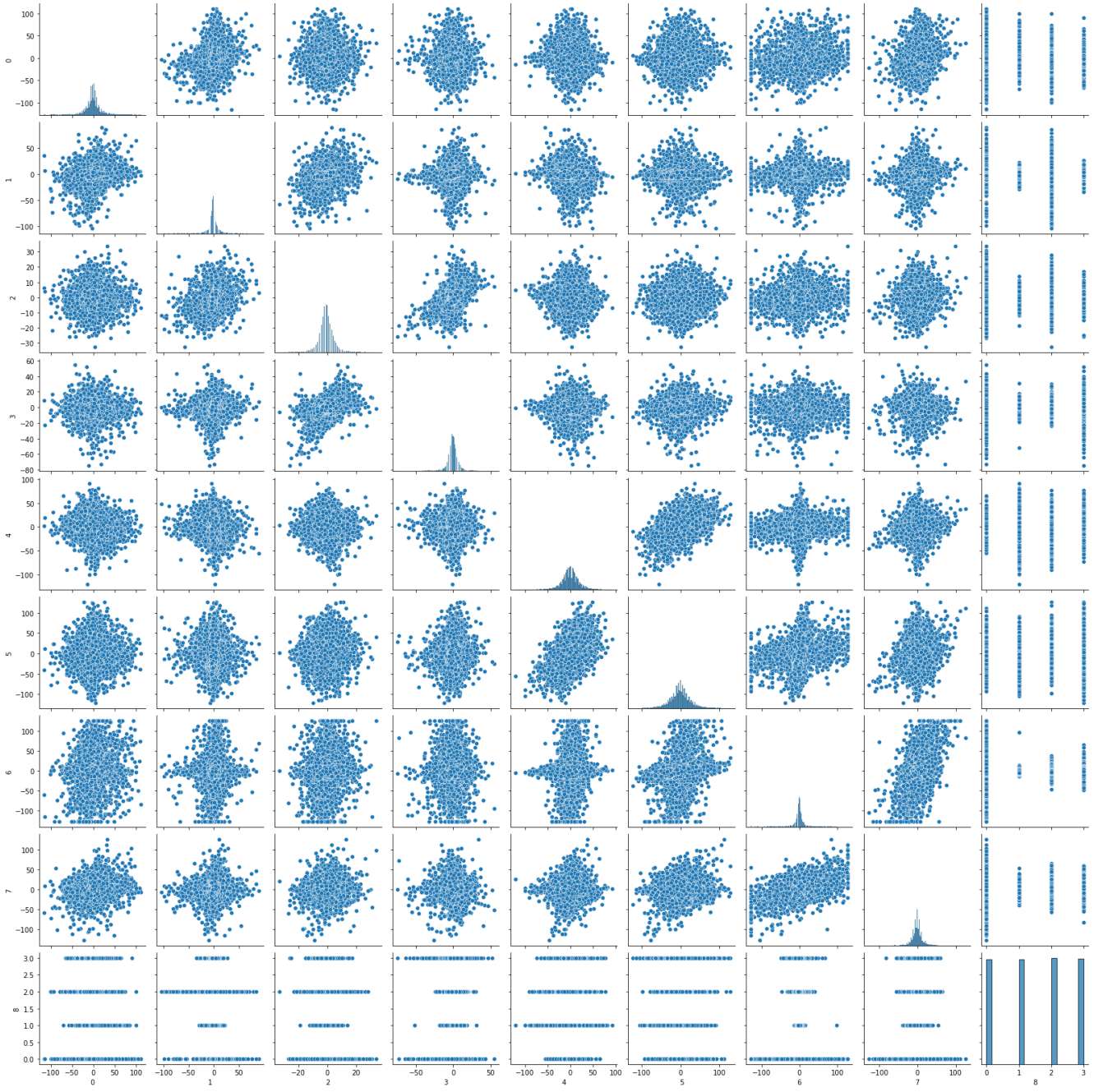
```
numpy.float64
```

Tots els atributs són float64

```
import seaborn as sns
from matplotlib import pyplot as plt

# data del primer múcul (els 8 sensors que el composen)
data1 = np.c_[dataset.values[:, :8], dataset.values[:, -1]]

data1 = pd.DataFrame(data1)

plt.figure()

ax = sns.heatmap(data1.corr(), annot=True, linewidths=.5)
```

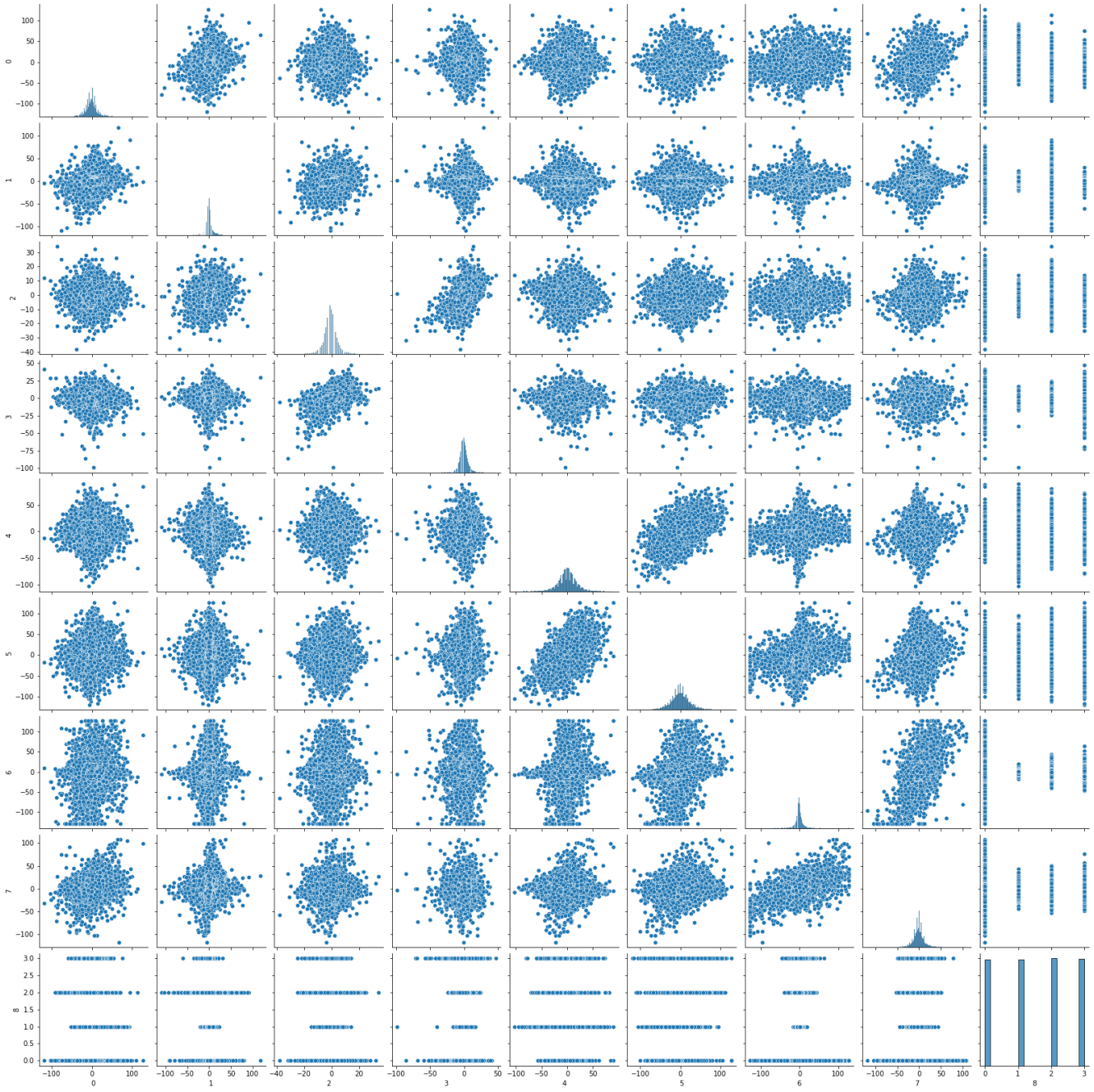



```
relacio = sns.pairplot(data1)
```



```
data3 = np.c_[dataset.values[:, 16:24], dataset.values[:, -1]]

data3 = pd.DataFrame(data3)
```

```
relacio = sns.pairplot(data3)
```



Em escollit dos músculs al atzar ja que considerem que per la resta s'aplicarien les mateixes normes. Aquest dos representen de forma general la resta del dataset.

Podem veure, com s'intuia a l'anàlisi de taula de descripció, que els valors dels sensors tenen forma gaussian, tant de forma indivisual com en la relació que representen entre ells.

També podem observar que aquest semblan ja normalitzats per la distrubució que presenten.

2 Preprocessing normalitzation, outlier removal, feature selection...

En aquesta secció tractarem les dades per tal de tenir més fàcilitats al treballar amb aquestes.

```
dataset.drop_duplicates()

X = data[:, :-1]
y = data[:, -1]
```

```
print("Forma de la base de dades:", dataset.shape)
print("Dimensions de la entrada x:", X.shape)
print("Dimensió dels atributs y:", y.shape)
```

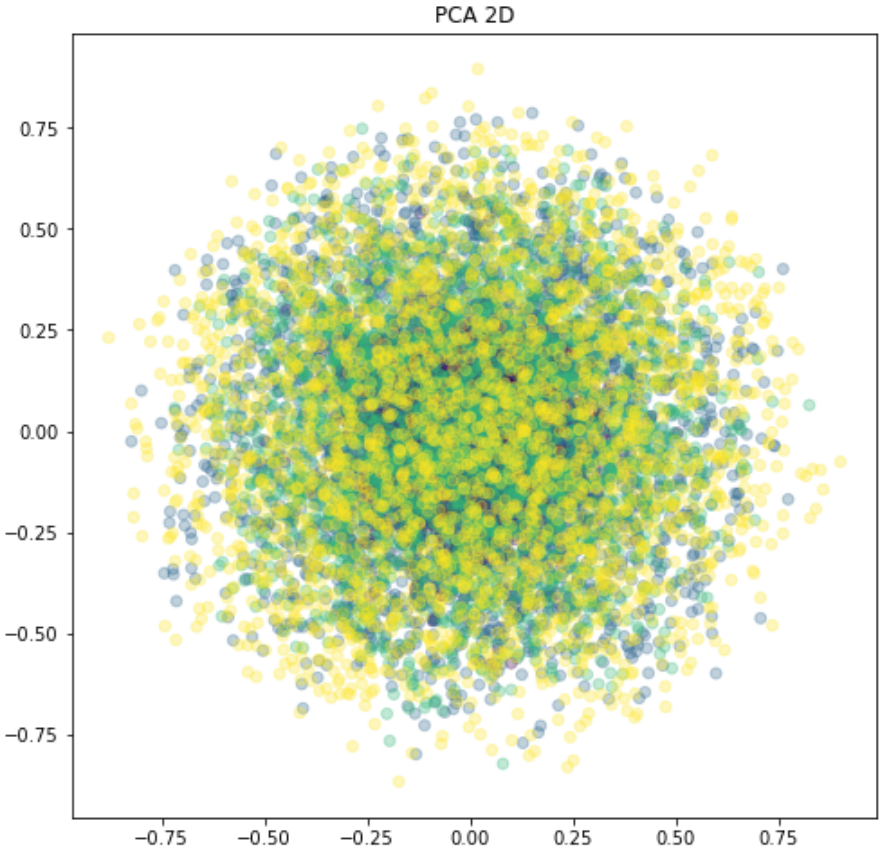
```
 Forma de la base de dades: (11678, 65)
Dimensions de la entrada x: (11678, 64)
Dimensió dels atributs y: (11678,)
```

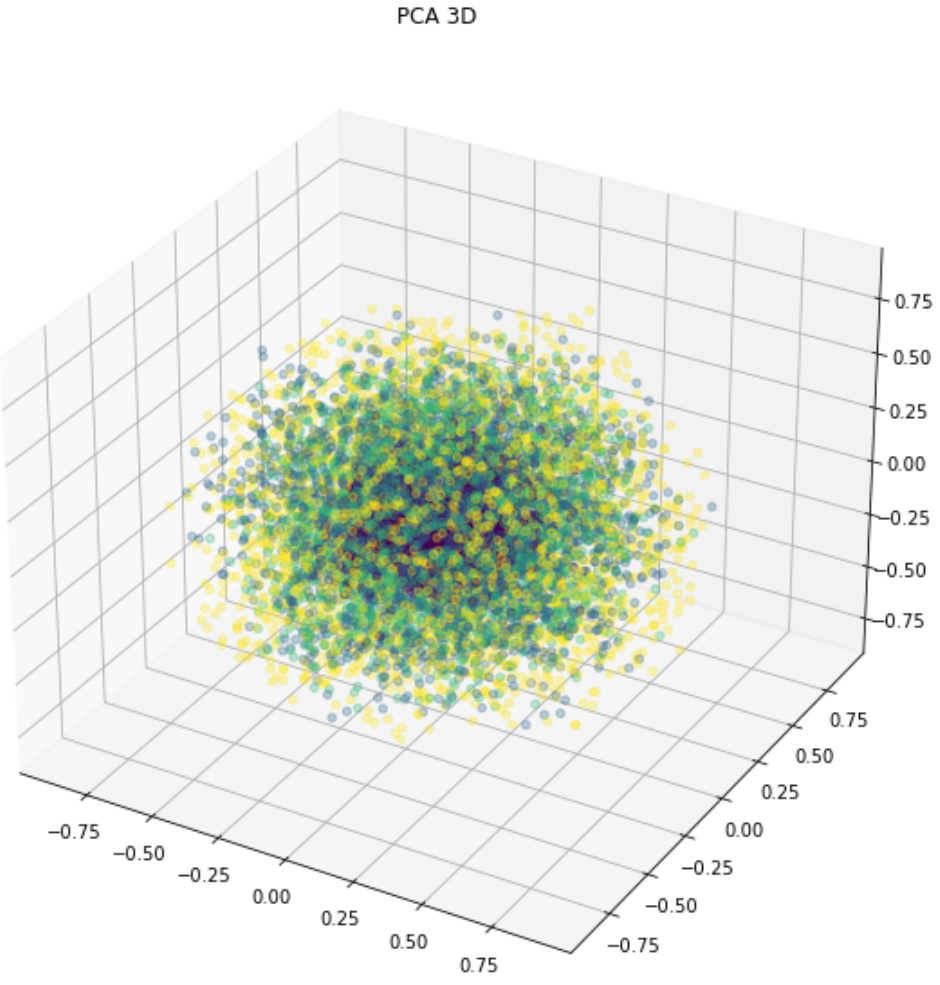
Normalitzem les dades per tal de que la distribució sigui homogenia en tots els atributs que tenim i així facilitar la classificació.

```
from sklearn.preprocessing import normalize

X = normalize(X)
```

L'anàlisi de components principals (*PCA*) s'utilitza per veure com es relacionen les dades dins d'un conjunt d'atributs.

```
from sklearn.decomposition import PCA
from sklearn.datasets import load_digits
digits = load_digits()

# PCA
pca = PCA(n_components=3)
projected = pca.fit_transform(X)

# 2D
plt.scatter(projected[:,0],projected[:,1], c= y, alpha = 0.3)
plt.gcf().set_size_inches((8, 8))
plt.title("PCA 2D")
plt.show()

# 3D
fig = plt.figure()
ax = fig.add_subplot(projection='3d')
ax.scatter(projected[:,0], projected[:,1], projected[:,2], c= y, alpha = 0.3)
fig.set_figheight(10)
fig.set_figwidth(10)
plt.title("PCA 3D")
plt.show()
```





En aquest cas veiem que no és util aplicar un PCA ja que al tenir unes dades tan homogenies, no podem reduir el nombre de components. Si ho fessim, perdriem massa informació necessaria per classificar.

3 Model Selection

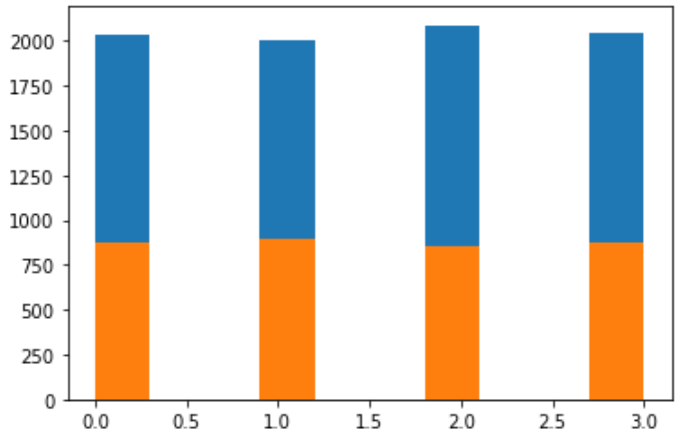
Una vegada fet tot això, entenar el dataset dins d'un model de selecció. Per això provarem varis i els comprovarem entre ells per veure quin és el que podria encaixar millor amb aquest.

Primer de tot dividim les dades de forma aleatoria en test i train. També, fem un histograma per tal de veure si la distribució en cada un, esta equilibrat i no hi ha descopensació de cara a una categoria.

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
```

```
plt.hist(y_train)
plt.hist(y_test)
```

```
(array([875.,   0.,   0., 896.,   0.,   0., 856.,   0.,   0., 877.]),
array([0. , 0.3, 0.6, 0.9, 1.2, 1.5, 1.8, 2.1, 2.4, 2.7, 3. ]),
<BarContainer object of 10 artists>)
```



Així, ja podem començar a provar models que puguin encaixar amb la nostra base de dades.

Super Vector Machine

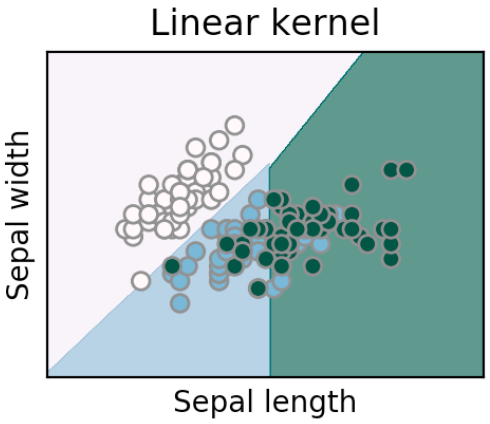
El SVM és un metode utilitzat per classificació entre d'altres, que consisteix en separar les classes a través de diferents algorismes que crean hiperplans juntament amb uns vectors de suport.

Acostuma a ser útil en espais de gans dimensions, aquest vectors de suport són els punts de entrenament en la desició, per tant és més eficient en temes de memòria i és molt versàtil ja que té diferenst algorismes (com ja hem mencionat) pel kernel que el fa més adaptable a les diferents situacions.

Al tenir tantes possibilitats, hem escollit diferents kernels per fer proves d'eficiència, com ara lineal, sigmoid i polinomic.

Lineal

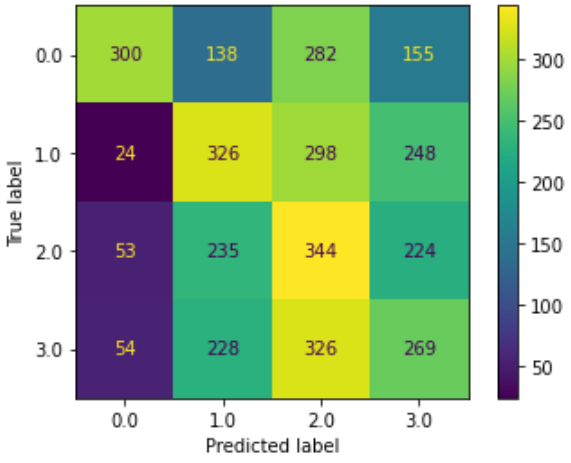
Aquest model consisteix en dividir les classes a través de hiperplans lineals com es pot veure a la imatge a continuació:



L'apliquem al nostre dataset per veure com funciona.

```
from sklearn.metrics import ConfusionMatrixDisplay
from sklearn import svm
from sklearn.metrics import accuracy_score, recall_score
import time

t0 = time.process_time()

linear_svc = svm.SVC(kernel='linear', probability=True)

linear_svc.fit(X_train, y_train)

y_pred_svc = linear_svc.predict(X_test)

print(f"Temps per generar el classificador: {time.process_time() - t0} s \n")

error =  ConfusionMatrixDisplay.from_predictions(y_test, y_pred_svc)
error = 1-(sum(np.diag(error.confusion_matrix)) / sum(error.confusion_matrix.ravel()))

accuracy = accuracy_score(y_test, y_pred_svc)
recall = recall_score(y_test, y_pred_svc, average='macro')

print(f"La taxa d'error és de: {error*100}%")
print(f"L' accuracy és de: {accuracy*100}%")
print(f"El recall és de: {recall*100}%")
```

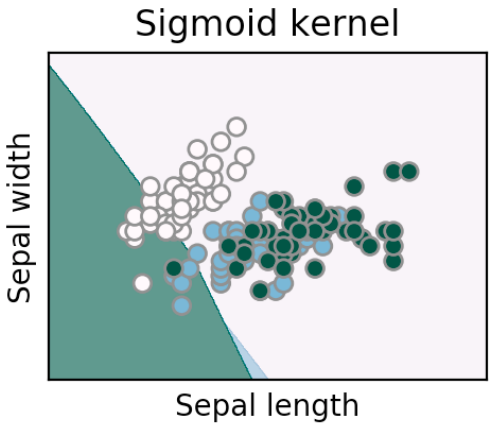
```
 Temps per generar el classificador: 20.8125 s


La taxa d'error és de: 64.640411095890411%
L' accuracy és de: 35.35958904109589%
El recall és de: 35.38232668738858%
```



Podem veure, que per fer la classificació no ha sigut molt efeicient, ja que té un error del 64%, això implica que és casi aleatoria la classificació.

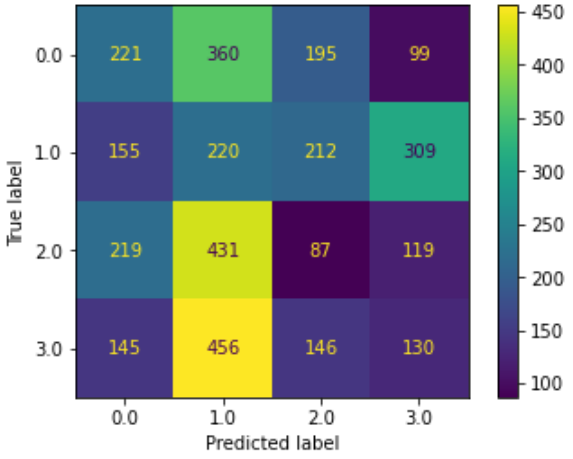
Sigmoid

En aquest classificador, el que fa és a través de la funció sigmoid crear els hiperplans:




```
t0 = time.process_time()

sigmoid_svc = svm.SVC(C=10.0, kernel='sigmoid', probability=True)

sigmoid_svc.fit(X_train, y_train)

y_pred_svc = sigmoid_svc.predict(X_test)


print(f"Temps per generar el classificador: {time.process_time() - t0} s \n")

error =  ConfusionMatrixDisplay.from_predictions(y_test, y_pred_svc)
error = 1-(sum(np.diag(error.confusion_matrix)) / sum(error.confusion_matrix.ravel()))

accuracy = accuracy_score(y_test, y_pred_svc)
recall = recall_score(y_test, y_pred_svc, average='macro')

print(f"La taxa d'error és de: {error*100}%")
print(f"L' accuracy és de: {accuracy*100}%")
print(f"El recall és de: {recall*100}%")
```

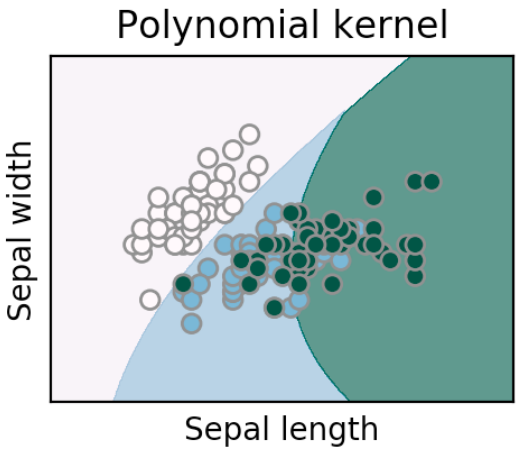
```
 Temps per generar el classificador: 24.71875 s

La taxa d'error és de: 81.22146118721460%
L' accuracy és de: 18.77853881278539%
El recall és de: 18.69938170125732%
```



Veiem que en aquest cas encara empitjora més la classificació, sent molt menys adaptable a la distribució que tenim.

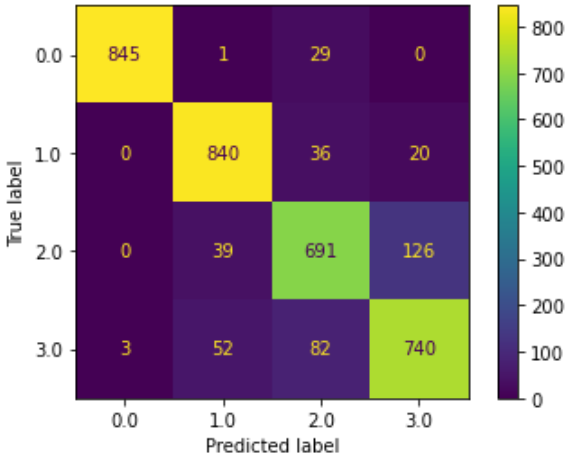
Polinòmica

La polinòmica, a través de un polinomi del grau que determinem nosaltres o la propia funció, crea la separació a través d'aquest:




```
t0 = time.process_time()

poly_svc = svm.SVC(C=10.0, kernel='poly', probability=True, degree=4)

poly_svc.fit(X_train, y_train)

y_pred_svc = poly_svc.predict(X_test)

print(f"Temps per generar el classificador: {time.process_time() - t0} s \n")

error =  ConfusionMatrixDisplay.from_predictions(y_test, y_pred_svc)
error = 1-(sum(np.diag(error.confusion_matrix)) / sum(error.confusion_matrix.ravel()))

accuracy = accuracy_score(y_test, y_pred_svc)
recall = recall_score(y_test, y_pred_svc, average='macro')

print(f"La taxa d'error és de: {error*100}%")
print(f"L' accuracy és de: {accuracy*100}%")
print(f"El recall és de: {recall*100}%")
```

```
 Temps per generar el classificador: 19.6875 s

La taxa d'error és de: 11.073059360730596%
L' accuracy és de: 88.9269406392694%
El recall és de: 88.8560727301929%
```



Dins dels SVM que hem escollit, aquest és el que dona un millor resultat, tot i que podria ser millorable, possiblement amb un canvi dels parametres podria arribar ha millorar força. Dins del temps, esta dins del que hem vist en els SVC, però molt per sota respecta els models que veurem a continuació.

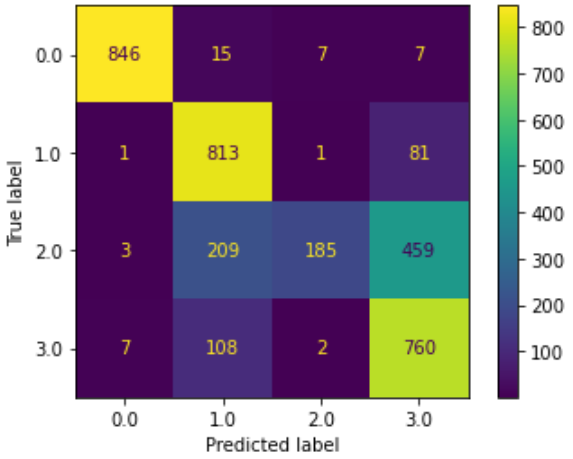
A més, veiem que la classe 'pedra' l'encerta sempre i no la confon mai amb altres. Potser, el que dona més error és que a vegades confon la classe 'tisores' amb el 'ok'.

K Nearest Neighbors

En aquest model la classificació es genera a través de quines són les classes dels punts més propers al qual es vol classificar. La classe que tingui més bots, és la que acaba sent assignada.

```
from sklearn.neighbors import KNeighborsClassifier

t0 = time.process_time()

neigh = KNeighborsClassifier(n_neighbors=100)

neigh.fit(X_train, y_train)

y_pred_neigh = neigh.predict(X_test)

print(f"Temps per generar el classificador: {time.process_time() - t0} s \n")

error =  ConfusionMatrixDisplay.from_predictions(y_test, y_pred_neigh)
error = 1-(sum(np.diag(error.confusion_matrix)) / sum(error.confusion_matrix.ravel()))

accuracy = accuracy_score(y_test, y_pred_neigh)
recall = recall_score(y_test, y_pred_neigh, average='macro')

print(f"La taxa d'error és de: {error*100}%")
print(f"L' accuracy és de: {accuracy*100}%")
print(f"El recall és de: {recall*100}%")
```

```
 Temps per generar el classificador: 1.75 s

La taxa d'error és de: 25.684931506849317%
L' accuracy és de: 74.31506849315068%
El recall és de: 73.92338398889511%
```



Aquest model com podem veure és molt bo, ja que té una taxa d'error del 25% però té un temps molt bo, aproximadament un segon, cosa que el deixa per sobre dels altres models.
A més, sembla que el que porta aquesta taxa d'error és en la seva majoria en la classe de les 'tisores' amb la classe del 'ok' o 'paper'. La resta sembla que més o menys cuadra bastant, potser destaca que té una tendencia també a la classe 'ok' classificar-la com a 'paper'.

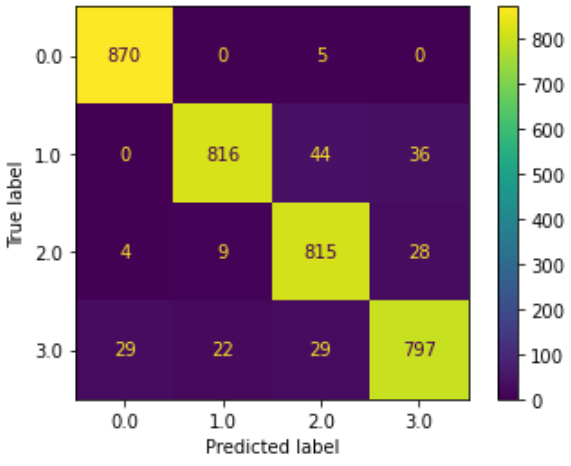
El que també veiem es que per la classe 'pedra' no té casi errors.

Random Forest

Random Forest és un tipus d'ensemble basat en arbres de desició formats de forma aleatoria que generen un bosc de desicions. Una vegada tots els arbres han pres la desició de a quina classe pertany la mostra, la classe més votada és la escollida.

```
from sklearn.ensemble import RandomForestClassifier

t0 = time.process_time()

forest = RandomForestClassifier(n_estimators=150, max_features=0.1, random_state=42)

forest.fit(X_train, y_train)

y_pred_forest = forest.predict(X_test)

print(f"Temps per generar el classificador: {time.process_time() - t0} s \n")

error =  ConfusionMatrixDisplay.from_predictions(y_test, y_pred_forest)
error = 1-(sum(np.diag(error.confusion_matrix)) / sum(error.confusion_matrix.ravel()))

accuracy = accuracy_score(y_test, y_pred_forest)
recall = recall_score(y_test, y_pred_forest, average='macro')

print(f"La taxa d'error és de: {error*100}%")
print(f"L' accuracy és de: {accuracy*100}%")
print(f"El recall és de: {recall*100}%")
```

```
 Temps per generar el classificador: 6.375 s

La taxa d'error és de: 5.8789954337899575%
L' accuracy és de: 94.12100456621005%
El recall és de: 94.14706838308166%
```



En 6 segons, veim que a conseguit una taxa d'error del 5% cosa molt satisfactoria. Podriem dir que és aquest el millor model dels que hem provat amb força diferència.

Potser podem dir que té una petita tendencia a confondre el 'ok' i el 'paper' amb altres classes, però són valors molt petits.

4 Cross Validation

Un cop tenim els models seleccionats, hem d'evaluar-los a través de Cross Validation per saber quin será el millor a l'hora de testejar sonre les dades.

Aprendre dels paràmetres d'una funció de predicció i provar-les amb les mateixes dades sempre es un error metodològic, un model que simplement repeteix les etiquetes que acaba de veure, sempre tindria una precisió perfecte i no podria predir quan arribi el moment, aquesta situació s'anomena overfitting, per poder evitar-ho separem el data set en K folds per tenir un training i un testing set diferents en cada fold.

Un cop tenim els models seleccionats, hem d'avaluar-los a través de Cross Validation amb la tècnica explicada anteriorment per saber quin serà el millor a l'hora de testejar sobre les dades.

```
from sklearn.model_selection import cross_val_score

for k in [3,5,10]:
    print(f"K-FOLDS: {k}")
    # poly
    scorespoly = cross_val_score(poly_svc, X, y, cv = k)
    print(f"Polinomica: {scorespoly.mean()} accuracy amb una desviació estandar {scorespoly.std()}")
    # KNearestNeighbour
    scoresknn = cross_val_score(neigh, X, y, cv = k)
    print(f"KNearestNeighbour: {scoresknn.mean()} accuracy amb una desviació estandar {scoresknn.std()}")
    # RandomForest
    scoresrandomforest = cross_val_score(forest, X, y, cv = k)
    print(f"Random Forest: {scoresrandomforest.mean()} accuracy amb una desviació estandar
{scoresrandomforest.std()} \n")
```

```
 K-FOLDS: 3
Polinomica: 0.8738646600608765 accuracy amb una desviació estandar 0.00908770124280652
KNearestNeighbour: 0.7337717877512602 accuracy amb una desviació estandar 0.009757751913479079
Random Forest: 0.9322675066068022 accuracy amb una desviació estandar 0.014804677758716883

K-FOLDS: 5
Polinomica: 0.8821712475433398 accuracy amb una desviació estandar 0.029448648442230008
KNearestNeighbour: 0.7401929028189258 accuracy amb una desviació estandar 0.017070615324519218
Random Forest: 0.9356088483764043 accuracy amb una desviació estandar 0.03516821617919262

K-FOLDS: 10
Polinomica: 0.8890248089586927 accuracy amb una desviació estandar 0.037622074374619 21
KNearestNeighbour: 0.7437917444330974 accuracy amb una desviació estandar 0.023617575780299754
Random Forest: 0.9381792090713807 accuracy amb una desviació estandar 0.03884493117659359
```

Amb la manera més simple d'implementar un cross validation, hem decidit provar amb 3 valors de K-fold diferents, 3,5 i 10 així podem veure quina variació de resultats hi ha, en els 3 diferents classificadors veiem que hi ha una millora mínima en cada tipus de K-fold el salt de k-fold 3 a 5 es major que de 5 a 10 pel lo que podem assegurar que a k-folds molt altes no millorà pràcticament el rendiment i ens podríem trobar en situació d'overfitting.

LeaveOneOut

Hem vist que amb la funció LeaveOneOut podem aplicar la funció utilitzada en el cross validation anomenada cross_val_score, un cop intentat ha passat molt de temps on no ha executat ni la part de la regressió lineal per lo que podem afirmar que no es viable aplicar LeaveOneOut en aquest cas.

```
from sklearn.model_selection import LeaveOneOut

def leaveOneOut():

    k = LeaveOneOut()
    print("LeaveOneOut")

    scoreslinlou = cross_val_score(linear_svc, X, y, cv = k)
    print(f"Lineal Regression: {scoreslinlou.mean()} accuracy with a standard deviation of
{scoreslinlou.std()}")
```

```
    scoresknnlou = cross_val_score(neigh, X, y, cv = k)
    print(f"KNearest Neighbour: {scoresknnlou.mean()} accuracy with a standard deviation of
{scoresknnlou.std()}")

    scoresrandomforestlou = cross_val_score(forest, X, y, cv = k)
    print(f"Random Forest: {scoresrandomforestlou.mean()} accuracy with a standard deviation of
{scoresrandomforestlou.std()}")
```

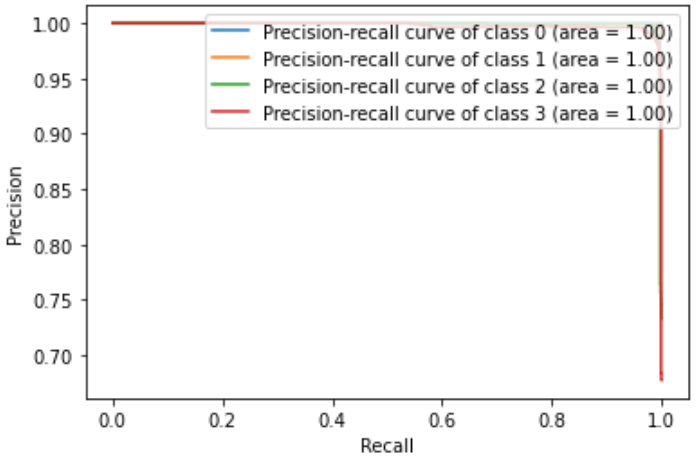
5 Metric Analysis

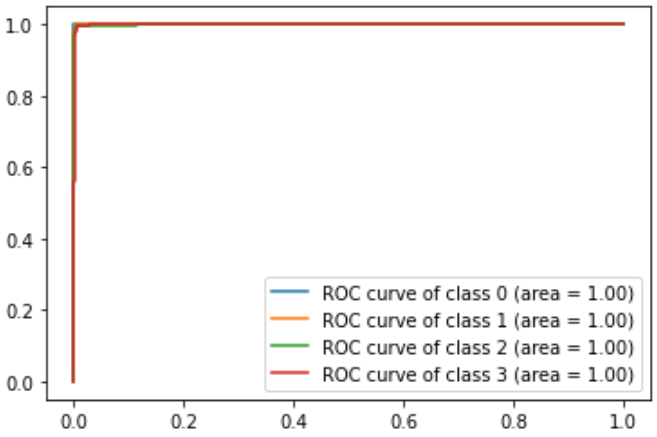
Curva Precisió-Recall i Curva ROC

SVM Polinòmica

```
from sklearn.metrics import f1_score, precision_recall_curve, average_precision_score, roc_curve, auc

# Compute Precision-Recall and plot curve
probs = poly_svc.predict_proba(X_train)
precision = {}
recall = {}
average_precision = {}
plt.figure()
n_classes = 4

for i in range(n_classes):
    precision[i], recall[i], _ = precision_recall_curve(y_train == i, probs[:, i])
    average_precision[i] = average_precision_score(y_train == i, probs[:, i])

    plt.plot(recall[i], precision[i],
    label='Precision-recall curve of class {0} (area = {1:0.2f})'
                          ''.format(i, average_precision[i]))
    plt.xlabel('Recall')
    plt.ylabel('Precision')
    plt.legend(loc="upper right")


# Compute ROC curve and ROC area for each class
fpr = {}
tpr = {}
roc_auc = {}
for i in range(n_classes):
    fpr[i], tpr[i], _ = roc_curve(y_train == i, probs[:, i])
    roc_auc[i] = auc(fpr[i], tpr[i])

# Compute micro-average ROC curve and ROC area
# Plot ROC curve
plt.figure()
for i in range(n_classes):
    plt.plot(fpr[i], tpr[i], label='ROC curve of class {0} (area = {1:0.2f})' ''.format(i, roc_auc[i]))
plt.legend()
```

```
<matplotlib.legend.Legend at 0x1cf8c03cc70>
```





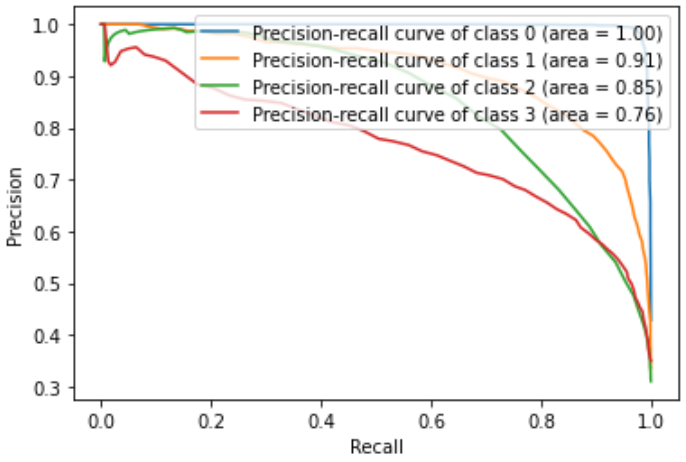
Com podem observar, tant la corva de ROC com la precision-recalltene una area mitjana de 100%, això indica que el nostre model és capaç de predir les classes amb una fiabilitat molt elevada, té una percisió del 100% excepte quan el recall ha d'augmentar fins al 100%, llavors arriba a una precisió màxima de 70%

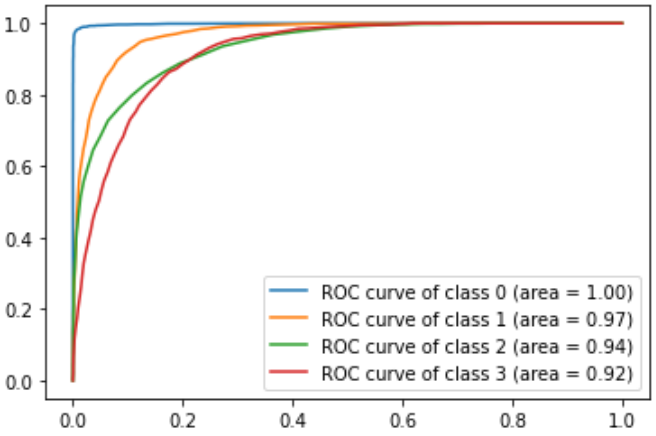
KNearestNeighbour

```
# Compute Precision-Recall and plot curve
probs = neigh.predict_proba(X_train)
precision = {}
recall = {}
average_precision = {}
plt.figure()

for i in range(n_classes):
    precision[i], recall[i], _ = precision_recall_curve(y_train == i, probs[:, i])
    average_precision[i] = average_precision_score(y_train == i, probs[:, i])

    plt.plot(recall[i], precision[i],
    label='Precision-recall curve of class {0} (area = {1:0.2f})'
                          ''.format(i, average_precision[i]))
    plt.xlabel('Recall')
    plt.ylabel('Precision')
    plt.legend(loc="upper right")


# Compute ROC curve and ROC area for each class
fpr = {}
tpr = {}
roc_auc = {}
for i in range(n_classes):
    fpr[i], tpr[i], _ = roc_curve(y_train == i, probs[:, i])
    roc_auc[i] = auc(fpr[i], tpr[i])

# Compute micro-average ROC curve and ROC area
# Plot ROC curve
```

```
plt.figure()
for i in range(n_classes):
    plt.plot(fpr[i], tpr[i], label='ROC curve of class {0} (area = {1:0.2f})' ''.format(i, roc_auc[i]))
plt.legend()
```

```
<matplotlib.legend.Legend at 0x1cfa28ce100>
```





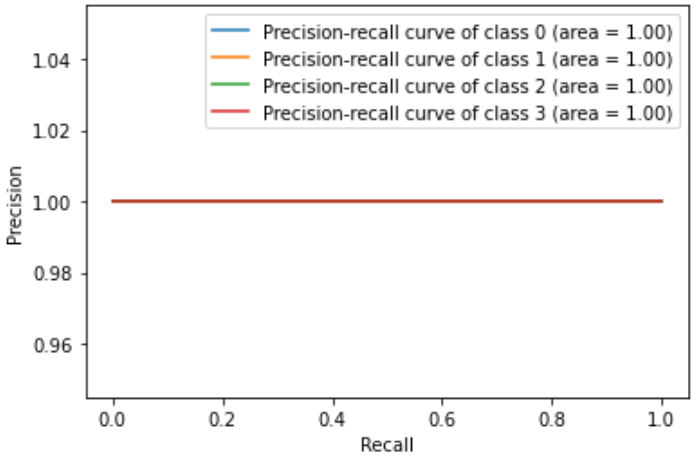
Aquí es pot veure com les corves ja no són tan perfectes, la classe 0 és la que més faci es pot predir i la classe 3 és la més complicada. Encara que els resultats anteriors siguin millors, cal tenir en compte que aquests resultats no són dolents, ja que a partir d'un 10% en les X totes les classe passen del 80% en les Y.

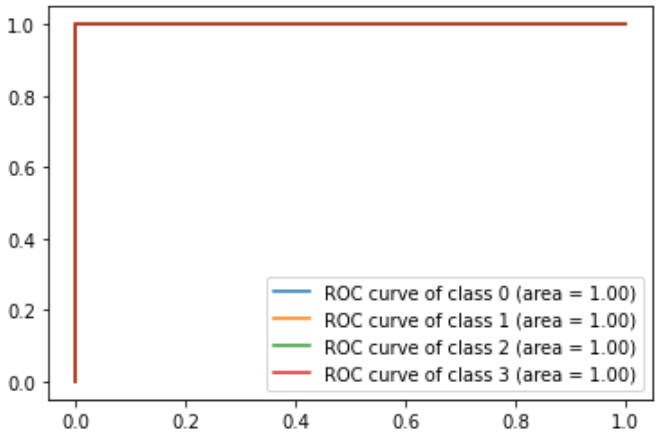
Random Forest

```
from sklearn.metrics import f1_score, precision_recall_curve, average_precision_score, roc_curve, auc

# Compute Precision-Recall and plot curve
probs = forest.predict_proba(X_train)
precision = {}
recall = {}
average_precision = {}
plt.figure()

for i in range(n_classes):
    precision[i], recall[i], _ = precision_recall_curve(y_train == i, probs[:, i])
    average_precision[i] = average_precision_score(y_train == i, probs[:, i])

    plt.plot(recall[i], precision[i],
    label='Precision-recall curve of class {0} (area = {1:0.2f})'
                          ''.format(i, average_precision[i]))
    plt.xlabel('Recall')
    plt.ylabel('Precision')
    plt.legend(loc="upper right")
```

```
# Compute ROC curve and ROC area for each class
fpr = {}
tpr = {}
roc_auc = {}
for i in range(n_classes):
    fpr[i], tpr[i], _ = roc_curve(y_train == i, probs[:, i])
    roc_auc[i] = auc(fpr[i], tpr[i])

# Compute micro-average ROC curve and ROC area
# Plot ROC curve
plt.figure()
for i in range(n_classes):
    plt.plot(fpr[i], tpr[i], label='ROC curve of class {0} (area = {1:0.2f})' ''.format(i, roc_auc[i]))
plt.legend()
```

```
<matplotlib.legend.Legend at 0x1cf8befdca0>
```





Aquí podem veure com les dues corves donen uns resultats massa perfectes, això pot ser degut a que s'ha generat overfitting, el qual intentarem resoldre més endavant.

6 Hyperparameter Search

La millor manera que hem trobat ha sigut a traves de la funció de HalvingGridSearchCV, que a través d'uns paràmetres aplica una estratègia de cerca on avalua tots el candidats amb pocs recursos i els itera triant els millors assignant poc a poc mes recursos. Creiem que es la millor ja que fer-ho amb força bruta tardaria una quantitat de temps que no seria viable computacionalment parlant, per lo que si tenim un ordinador amb recursos limitats no ens donaria temps ni a obtenir resultats.

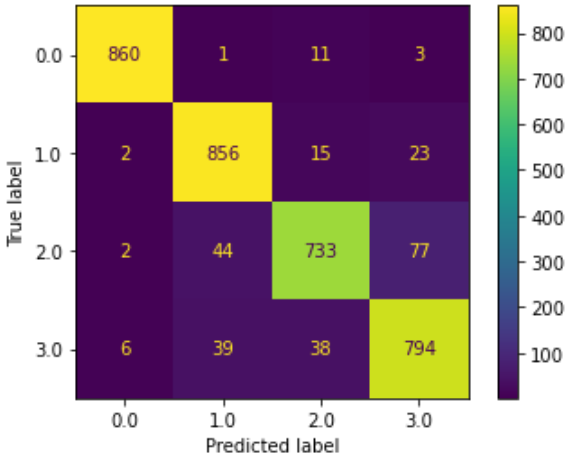
SVM Polinòmica

```
from sklearn.experimental import enable_halving_search_cv
from sklearn.model_selection import HalvingGridSearchCV

param_grid = param_grid = {'C': [1, 10, 100, 1000],
                           'gamma': ['scale', 'auto'], 'degree': range(1,5)}
poly_svc = svm.SVC(C=10.0, kernel='poly',probability = True)
search = HalvingGridSearchCV(poly_svc, param_grid,random_state=42).fit(X_train, y_train)
search.best_params_
```

```
 {'C': 10, 'degree': 2, 'gamma': 'scale'}
```

```
t0 = time.process_time()

poly_svc = svm.SVC(C=10.0, kernel='poly',probability = True, degree=2, gamma='scale')

poly_svc.fit(X_train, y_train)

y_pred_svc = poly_svc.predict(X_test)

print(f"Temps per generar el classificador: {time.process_time() - t0} s \n")

error =  ConfusionMatrixDisplay.from_predictions(y_test, y_pred_svc)
error = 1-(sum(np.diag(error.confusion_matrix)) / sum(error.confusion_matrix.ravel()))

accuracy = accuracy_score(y_test, y_pred_svc)
recall = recall_score(y_test, y_pred_svc, average='macro')

print(f"La taxa d'error és de: {error*100}%")
print(f"L' accuracy és de: {accuracy*100}%")
print(f"El recall és de: {recall*100}%")
```

```
 Temps per generar el classificador: 7.0 s

La taxa d'error és de: 7.448630136986301%
L' accuracy és de: 92.5513698630137%
El recall és de: 92.49704689871558%
```



En la regressió polinòmica observem que els millors paràmetres son: C=10, degree que li vam posar una possibilitat entre 1 i 5 la millor era 2 i en el paràmetre gamma el que obté un rendiment major es scale amb tot això obtenim un accuracy de 92,5% que es molt millor que l'anterior millorant-ho en un 8%.

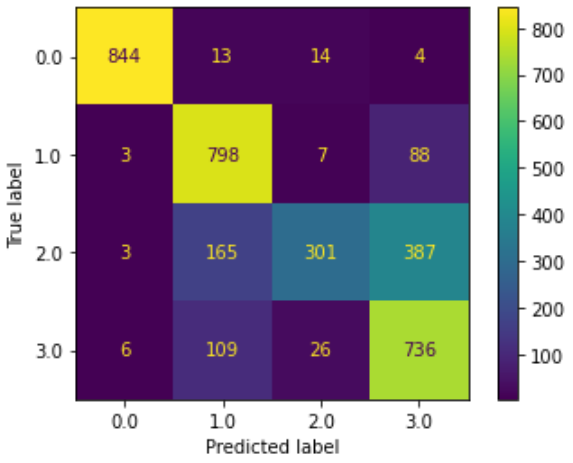
KNearestNeighbors

```
param_grid = {'n_neighbors': range(1,10), 'weights': ['uniform', 'distance'],
              'algorithm': ['auto', 'ball_tree', 'kd_tree', 'brute']}
neigh = KNeighborsClassifier()
search = HalvingGridSearchCV(neigh, param_grid,random_state=42).fit(X_train, y_train)
search.best_params_
```

```
 {'algorithm': 'auto', 'n_neighbors': 5, 'weights': 'distance'}
```

```
t0 = time.process_time()

neigh = KNeighborsClassifier(n_neighbors=5, weights='distance', algorithm = 'auto')

neigh.fit(X_train, y_train)

y_pred_neigh = neigh.predict(X_test)

print(f"Temps per generar el classificador: {time.process_time() - t0} s \n")

error =  ConfusionMatrixDisplay.from_predictions(y_test, y_pred_neigh)
error = 1-(sum(np.diag(error.confusion_matrix)) / sum(error.confusion_matrix.ravel()))

accuracy = accuracy_score(y_test, y_pred_neigh)
recall = recall_score(y_test, y_pred_neigh, average='macro')

print(f"La taxa d'error és de: {error*100}%")
print(f"L' accuracy és de: {accuracy*100}%")
print(f"El recall és de: {recall*100}%")
```

```
 Temps per generar el classificador: 1.640625 s

La taxa d'error és de: 23.544520547945204%
L' accuracy és de: 76.45547945205479%
El recall és de: 76.15141430021481%
```



En KNearestNeighbors donem la possibilitat al paràmetre n_neighbors de tenir un valor del 1 al 10, el millor sent 5, en weights veiem que la millor opció es distance i en el tipus de algoritme la millor opció es auto, la millora es d'un 2% solament.

Random Forest

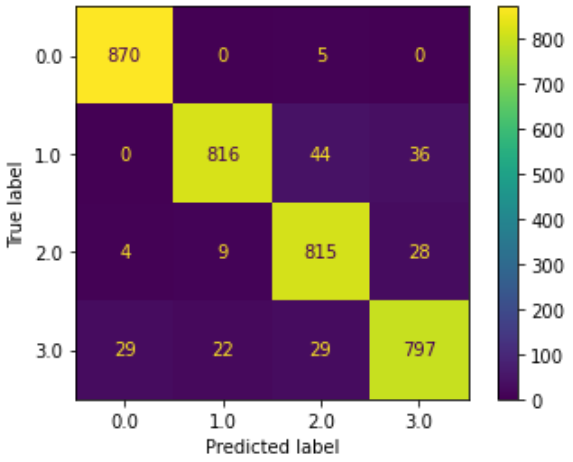
```
param_grid = {'n_estimators': range(1,200,50), 'max_features': ['sqrt', 'log2', None]},
forest = RandomForestClassifier(random_state=42)
```

```
search = HalvingGridSearchCV(forest, param_grid,random_state=42).fit(X_train, y_train)
search.best_params_
```

```
{'max_features': 'log2', 'n_estimators': 151}
```

```
t0 = time.process_time()

forest = RandomForestClassifier(n_estimators=150, max_features='log2', random_state=42)

forest.fit(X_train, y_train)

y_pred_forest = forest.predict(X_test)

print(f"Temps per generar el classificador: {time.process_time() - t0} s \n")

error =  ConfusionMatrixDisplay.from_predictions(y_test, y_pred_forest)
error = 1-(sum(np.diag(error.confusion_matrix)) / sum(error.confusion_matrix.ravel()))

accuracy = accuracy_score(y_test, y_pred_forest)
recall = recall_score(y_test, y_pred_forest, average='macro')

print(f"La taxa d'error és de: {error*100}%")
print(f"L' accuracy és de: {accuracy*100}%")
print(f"El recall és de: {recall*100}%")
```

```
Temps per generar el classificador: 6.34375 s

La taxa d'error és de: 5.8789954337899575%
L' accuracy és de: 94.12100456621005%
El recall és de: 94.14706838308166%
```



Els millors paràmetres en aquest últim classificador son: n_estimators = 151 imax_features amb l'opció log2 on observem una millora mínima.

```
for k in [3,5,10]:
    print(f"K-FOLDS: {k}")
    # poly
    scorespoly = cross_val_score(poly_svc, X, y, cv = k)
    print(f"Polinomica: {scorespoly.mean()} accuracy amb una desviació estandar {scorespoly.std()}")
    # KNearestNeighbour
    scoresknn = cross_val_score(neigh, X, y, cv = k)
    print(f"KNearestNeighbour: {scoresknn.mean()} accuracy amb una desviació estandar {scoresknn.std()}")
    # RandomForest
    scoresrandomforest = cross_val_score(forest, X, y, cv = k)
```

```
    print(f"Random Forest: {scoresrandomforest.mean()} accuracy amb una desviació estandar
{scoresrandomforest.std()} \n")
```

```
 K-FOLDS: 3
Polinomica: 0.9100020046345954 accuracy amb una desviació estandar 0.004657718950415292
KNearestNeighbour: 0.7561214834964805 accuracy amb una desviació estandar 0.010350869904555587
Random Forest: 0.9322675066068022 accuracy amb una desviació estandar 0.01480467758716883

K-FOLDS: 5
Polinomica: 0.9155685884837641 accuracy amb una desviació estandar 0.028076472040000574
KNearestNeighbour: 0.7612579566454489 accuracy amb una desviació estandar 0.021198492342409954
Random Forest: 0.9356088483764043 accuracy amb una desviació estandar 0.03516821617919262

K-FOLDS: 10
Polinomica: 0.9235335892289092 accuracy amb una desviació estandar 0.030258352163677987
KNearestNeighbour: 0.7657133676092545 accuracy amb una desviació estandar 0.027543963207177155
Random Forest: 0.9381792090713807 accuracy amb una desviació estandar 0.03884493117659359
```

Un cop fet el cross-validation veiem que el rendiment a través dels diferents K-folds no varia però, els valors de accuracy si milloren com hem dit en els apartats anteriors.

Poly	precisión (%)	C	degree	gamma
Bàsica				
50%(T) - 50%(V)	87,66%	10	4	scale
80%(T) - 20%(V)	89,38%	10	4	scale
70%(T) - 30%(V)	88,92%	10	4	scale
K-fold				
K = 3	87,38%	10	4	scale
K = 5	88,22%	10	4	scale
K = 10	88,90%	10	4	scale
Hyperparametros				
Bàsica				
50%(T) - 50%(V)	91,37%	10	2	scale
80%(T) - 20%(V)	92,77%	10	2	scale
70%(T) - 30%(V)	92,55%	10	2	scale
K-fold				
K = 3	91%	10	2	scale
K = 5	91,55%	10	2	scale
K = 10	92,35%	10	2	scale

KNN	precisión (%)	algorithm	n_neighbors	weights
Bàsica				
50%(T) - 50%(V)	74,12%	auto	5	uniform
80%(T) - 20%(V)	74,48%	auto	5	uniform
70%(T) - 30%(V)	74,32%	auto	5	uniform

KNN	precisión (%)	algorithm	n_neighbors	weights
K-fold				
K = 3	73,38%	auto	5	uniform
K = 5	74,02%	auto	5	uniform
K = 10	74,37%	auto	5	uniform
Hyperparametros				
Bàsica				
50%(T) - 50%(V)	75,83%	auto	5	distance
80%(T) - 20%(V)	76,97%	auto	5	distance
70%(T) - 30%(V)	76,45%	auto	5	distance
K-fold				
K = 3	76%	auto	5	distance
K = 5	76,12%	auto	5	distance
K = 10	76,57%	auto	5	distance

RandomForest	precisión (%)	max_features	n_estimators
Bàsica			
50%(T) - 50%(V)	93,93%	0.1	150
80%(T) - 20%(V)	94,60%	0.1	150
70%(T) - 30%(V)	94,12%	0.1	150
K-fold			
K = 3	93,22%	0.1	150
K = 5	93,56%	0.1	150
K = 10	93,81%	0.1	150
Hyperparametros			
Bàsica			
50%(T) - 50%(V)	93,93%	log2	151
80%(T) - 20%(V)	94,60%	log2	151
70%(T) - 30%(V)	<	log2	151
K-fold			
K = 3	93,22%	log2	151
K = 5	93,56%	log2	151
K = 10	93,81%	log2	151

Apartat (A): Comparativa de Models

En aquest apartat farem una comparativa dels tres models que hem escollit com a millors, per les característiques que creiem que són millors com el temps d'execució i un millor recall i acuraccy. Aquests models són regressió polinòmica, KNearestNeighbors i Random Forest.

Així, primer de tot, mirarem si és necessari fer més o menys entrenament per a cada model:

```python
from sklearn.linear_model import LogisticRegression

# Particions per fer el test i el train
particions = [0.3, 0.5, 0.7, 0.8]

# Creació dels classificadors
poly_svc = svm.SVC(C=10.0, kernel='poly',probability = True, degree=2, gamma='scale')
neigh = KNeighborsClassifier(n_neighbors=5, weights='distance', algorithm='auto')
forest = RandomForestClassifier(n_estimators=150, max_features='log2', random_state=42)

for part in particions:
    # fer les particions
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=part, random_state=42)

    # Poly
    t0 = time.process_time()
    poly_svc.fit(X_train, y_train)
    probspoly=poly_svc.predict_proba(X_test)
    print(f"Temps: {time.process_time() - t0} s")
    print (f"Correct classification polynomical  {part}% of the data: {poly_svc.score(X_test, y_test)}")

    # KNearestNeighbors
    t0 = time.process_time()
    neigh.fit(X_train, y_train)
    probsknn = neigh.predict_proba(X_test)
    print(f"Temps: {time.process_time() - t0} s")
    print (f"Correct classification KNN          {part}% of the data: {neigh.score(X_test, y_test)}")

    # Random Forest
    t0 = time.process_time()
    forest.fit(X_train, y_train)
    probsRF = forest.predict_proba(X_test)
    print(f"Temps: {time.process_time() - t0} s")
    print (f"Correct classification RandomForest {part}% of the data: {forest.score(X_test, y_test)}\n")
```

```
 Temps: 7.046875 s
Correct classification polynomical  0.3% of the data: 0.925513698630137
Temps: 1.609375 s
Correct classification KNN          0.3% of the data: 0.7645547945205 48
Temps: 6.4375 s
Correct classification RandomForest 0.3% of the data: 0.9412100456621004

Temps: 4.390625 s
Correct classification polynomical  0.5% of the data: 0.9136838499743106
Temps: 1.765625 s
Correct classification KNN          0.5% of the data: 0.7583490323685562
Temps: 4.34375 s
Correct classification RandomForest 0.5% of the data: 0.9393731803390991

Temps: 2.171875 s
Correct classification polynomical  0.7% of the data: 0.8870948012232416
Temps: 1.640625 s
Correct classification KNN          0.7% of the data: 0.7443425076452599
Temps: 2.390625 s
Correct classification RandomForest 0.7% of the data: 0.9363914373088685

Temps: 1.328125 s
Correct classification polynomical  0.8% of the data: 0.8663170287916087
Temps: 1.53125 s
Correct classification KNN          0.8% of the data: 0.7300652895215669
```

```
Temps: 1.5625 s
Correct classification RandomForest 0.8% of the data: 0.9273252702558065
```

Podem veure que cada classificador a reaccionat de formes diferents a aquest rang de test/train.

Així doncs, en la polinòmica veiem que quan menys train a de fer, menys triga, però això afecta a la seva predicció. Podem dir que en aquest cas que el 50% és potser el més òptim, ja que hi ha una disminució del 1% de presició a canvi de 3s.

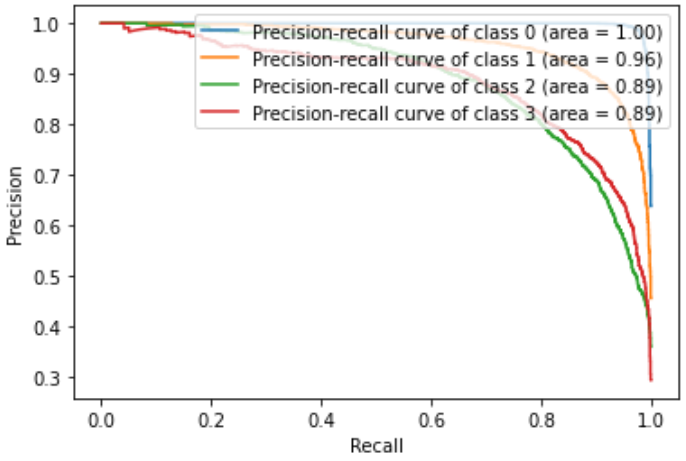
En el classificador KNN veiem que els canvis no afecten al seu temps i casi que tampoc a la seva acuraccy. Del millor al pitjor cas només va un 3% de diferencia. Per això, creiem que el millor és el de 30% de test, que per molt poc, té una millor acuraccy, però podriem dir que és la que té un millor resultat.

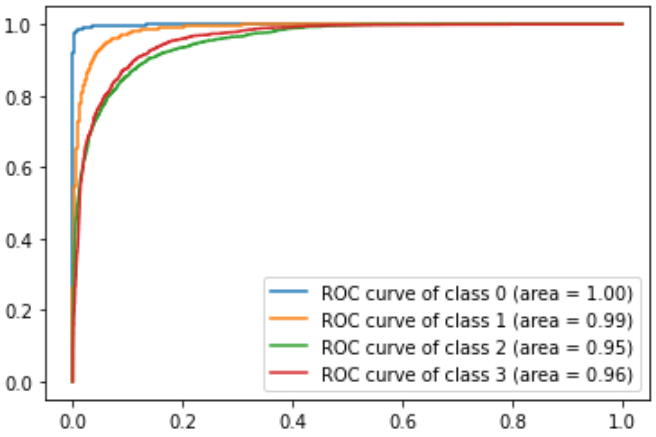
En el Random Forest veiem una millora del temps quan menys train és fa i el canvi de el percentatge d'accuracy no varia ni un 1%, per tant, creiem que el millor seria el de 70% de test ja que té molta millora de temps i poca pèrdua d'eficiència.

ROC SVM Polinòmica

```
from sklearn.metrics import f1_score, precision_recall_curve, average_precision_score, roc_curve, auc

# Compute Precision-Recall and plot curve
precision = {}
recall = {}
average_precision = {}
plt.figure()

for i in range(n_classes):
    precision[i], recall[i], _ = precision_recall_curve(y_test == i, probspoly[:,i])
    average_precision[i] = average_precision_score(y_test == i, probspoly[:,i])

    plt.plot(recall[i], precision[i],
    label='Precision-recall curve of class {0} (area = {1:0.2f})'
                          ''.format(i, average_precision[i]))
    plt.xlabel('Recall')
    plt.ylabel('Precision')
    plt.legend(loc="upper right")


# Compute ROC curve and ROC area for each class
fpr = {}
tpr = {}
roc_auc = {}
for i in range(n_classes):
    fpr[i], tpr[i], _ = roc_curve(y_test == i, probspoly[:,i])
    roc_auc[i] = auc(fpr[i], tpr[i])

# Compute micro-average ROC curve and ROC area
# Plot ROC curve
plt.figure()
for i in range(n_classes):
    plt.plot(fpr[i], tpr[i], label='ROC curve of class {0} (area = {1:0.2f})' ''.format(i, roc_auc[i]))
plt.legend()
```

```
<matplotlib.legend.Legend at 0x1cf8bce1670>
```

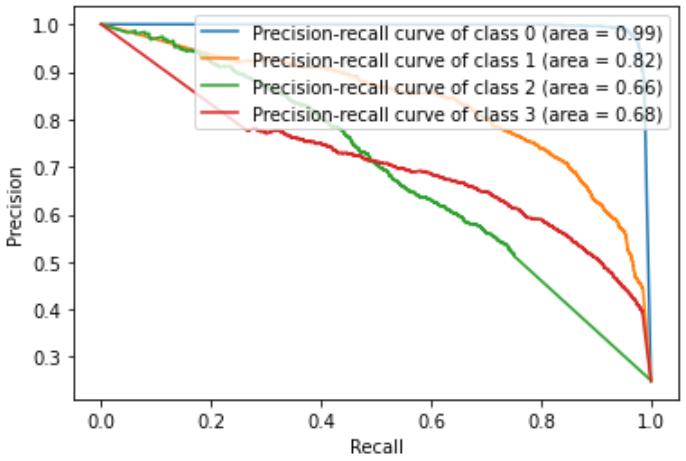


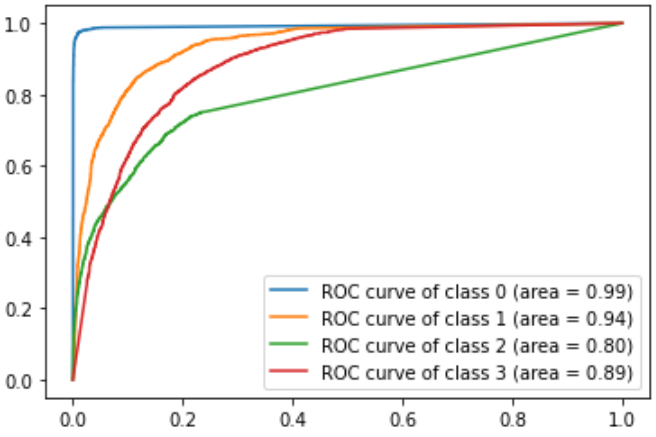


ROC KNearestNeighbors

```
# Compute Precision-Recall and plot curve
precision = {}
recall = {}
average_precision = {}
plt.figure()

for i in range(n_classes):
    precision[i], recall[i], _ = precision_recall_curve(y_test == i, probsknn[:,i])
    average_precision[i] = average_precision_score(y_test == i, probsknn[:,i])

    plt.plot(recall[i], precision[i],
    label='Precision-recall curve of class {0} (area = {1:0.2f})'
                          ''.format(i, average_precision[i]))
    plt.xlabel('Recall')
    plt.ylabel('Precision')
    plt.legend(loc="upper right")


# Compute ROC curve and ROC area for each class
fpr = {}
tpr = {}
roc_auc = {}
for i in range(n_classes):
    fpr[i], tpr[i], _ = roc_curve(y_test == i, probsknn[:,i])
    roc_auc[i] = auc(fpr[i], tpr[i])

# Compute micro-average ROC curve and ROC area
# Plot ROC curve
plt.figure()
for i in range(n_classes):
    plt.plot(fpr[i], tpr[i], label='ROC curve of class {0} (area = {1:0.2f})' ''.format(i, roc_auc[i]))
plt.legend()
```

```
<matplotlib.legend.Legend at 0x1cf8bb8c130>
```



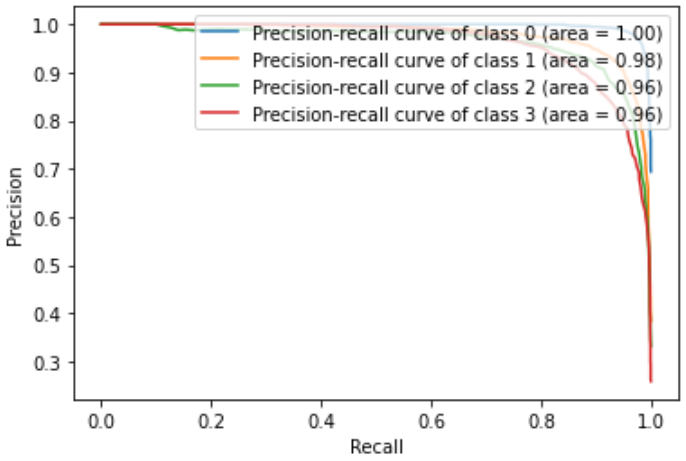


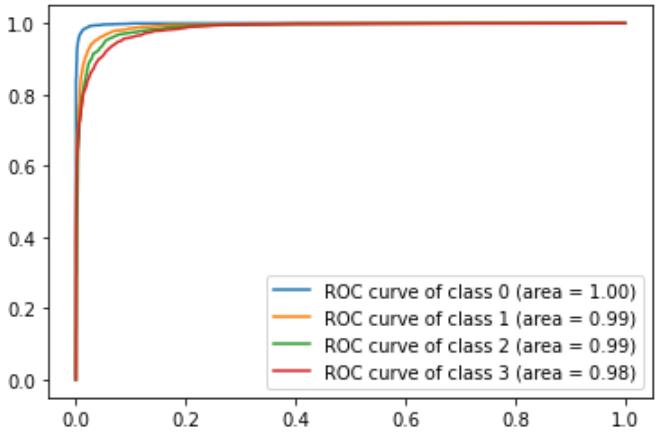
ROC Random Forest

```
# Compute Precision-Recall and plot curve
precision = {}
recall = {}
average_precision = {}
plt.figure()

for i in range(n_classes):
    precision[i], recall[i], _ = precision_recall_curve(y_test == i, probsRF[:,i])
    average_precision[i] = average_precision_score(y_test == i, probsRF[:,i])

    plt.plot(recall[i], precision[i],
    label='Precision-recall curve of class {0} (area = {1:0.2f})'
                          ''.format(i, average_precision[i]))
    plt.xlabel('Recall')
    plt.ylabel('Precision')
    plt.legend(loc="upper right")


# Compute ROC curve and ROC area for each class
fpr = {}
tpr = {}
roc_auc = {}
for i in range(n_classes):
    fpr[i], tpr[i], _ = roc_curve(y_test == i, probsRF[:,i])
    roc_auc[i] = auc(fpr[i], tpr[i])

# Compute micro-average ROC curve and ROC area
# Plot ROC curve
plt.figure()
```

```
for i in range(n_classes):
    plt.plot(fpr[i], tpr[i], label='ROC curve of class {0} (area = {1:0.2f})' ''.format(i, roc_auc[i]))
plt.legend()
```

```
<matplotlib.legend.Legend at 0x1cf8b7eb520>
```





Tal i com podem veure amb les corves precision-recall i ROC, l'ordre dels nostres models per rendiment seria:

3r - KNN - Assolim una area mijana de 79% en precision-recall i 90% en la ROC
2n - SVM polinmoial - Assolim una area mijana de 93% en precision-recall i 97% en la ROC
1r - Random forest - Assolim una area mijana de 98% en precision-recall i 100% en la ROC

Cal destacar que amb tots els models, la classe més ben predita és la class 0 (pedra) amb quasi un 100% d'area en tots els casos. Això suposem que és degut a que com que els sensors mesuren l'activitat muscular, la perda és la que activa més sensor, i, per tant, la més facil d'identificar

La segona més ben predita és la class 1 (tisores) i, finalment les classes 2 (paper) i 3 (ok), les quals se solen confondre entre elles

S'ha de tenir en compte també que encara que el KNN no doni els millor resultat en questió de precision-recall, és la que dona millors resultats en temps d'entrenament i precicció, amb una diferencia de 1-6 amb el random forest i 1-20 amb el SVM

Tenint totes aquestes consideracions en compte, creiem que en el cas de necessitar un model ràpid en el que no importi tant si s'equivoca, s'ha de fer servir el KNN i en cas de necessitar un model que s'equivoqui el minim possible, s'ha de fer servir el RandomForest. En cap cas creiem que el SVM sigui una bola elecció, ja que tarda 3 vegades més que el RandomFores i te una precisió menor.

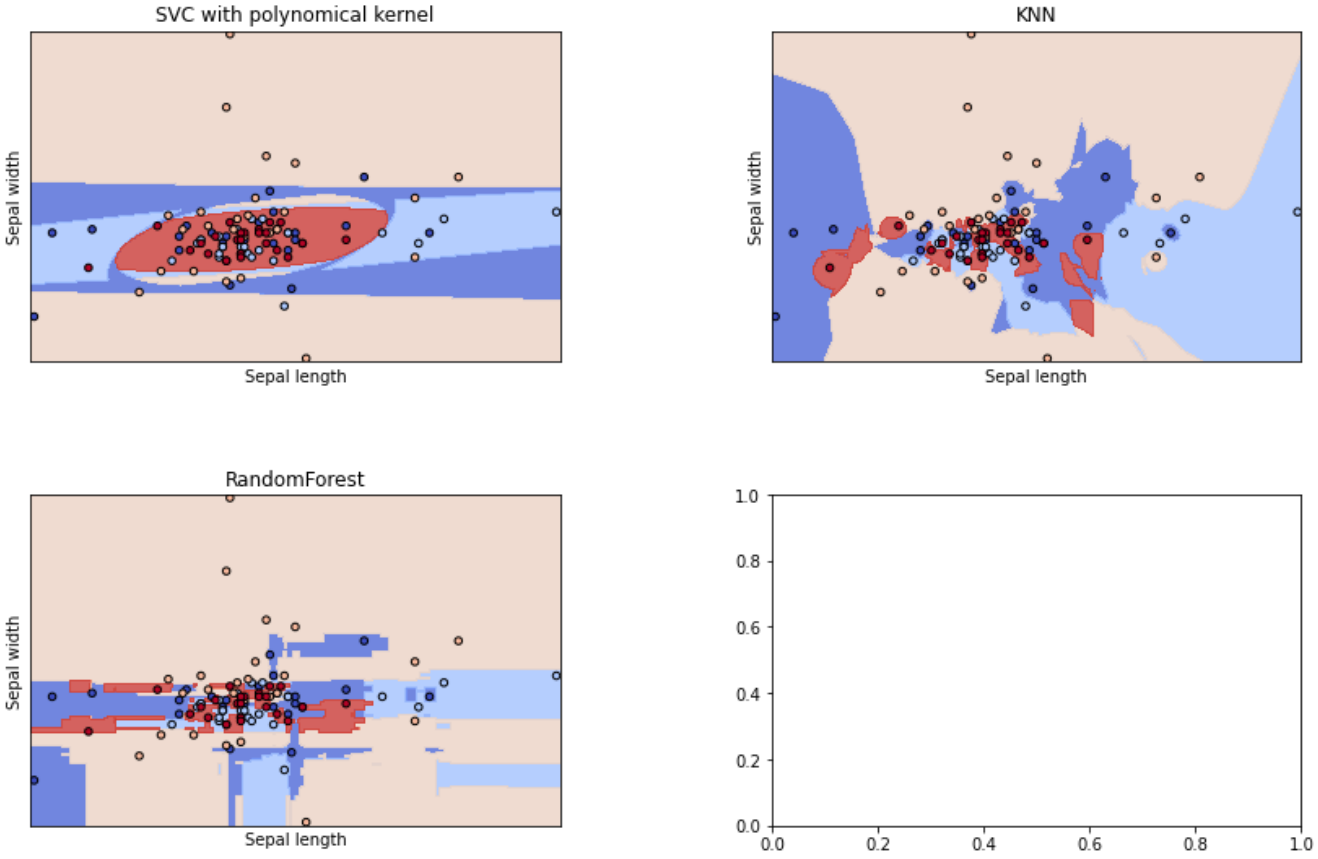
Efecte en C

Ara, veurem com per a diferents valors de C, gamma i degree va variant la distribució de les classes de forma gràfica de la següent manera:

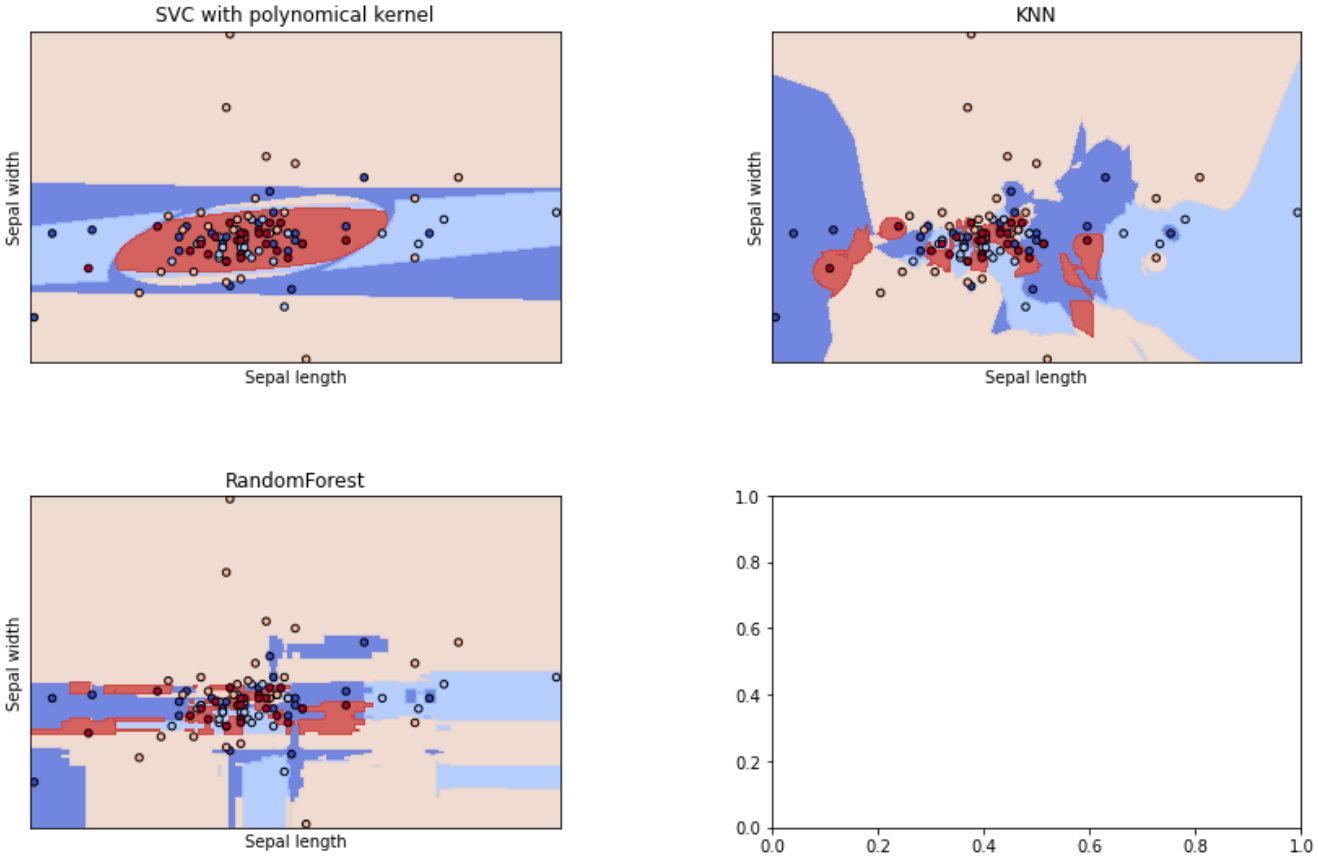
```
def make_meshgrid(x, y, h=.02):
    """Create a mesh of points to plot in

    Parameters
    ----------
    x: data to base x-axis meshgrid on
    y: data to base y-axis meshgrid on
    h: stepsize for meshgrid, optional

    Returns
    -------
    xx, yy : ndarray
    """
    x_min, x_max = x.min() - 1, x.max() + 1
    y_min, y_max = y.min() - 1, y.max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                         np.arange(y_min, y_max, h))

    return xx, yy


def plot_contours(ax, clf, xx, yy, **params):
    """Plot the decision boundaries for a classifier.

    Parameters
    ----------
    ax: matplotlib axes object
    clf: a classifier
    xx: meshgrid ndarray
    yy: meshgrid ndarray
    params: dictionary of params to pass to contourf, optional
    """
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
    out = ax.contourf(xx, yy, Z, **params)
    return out

def show_C_effect(C=1.0, gamma=0.7, degree=3):
    y = data[:, -1]
    X = data[:, :2]
    # we create an instance of SVM and fit out data. We do not scale our
    # data since we want to plot the support vectors
    # title for the plots
    arr = np.linspace(0,11450,100, dtype=np.int16)
    X = X[arr,:]
    y = y[arr]
    titles = ('SVC with polynomical kernel',
              'KNN',
              'RandomForest')

    #C = 1.0  # SVM regularization parameter
    models = (poly_svc, neigh, forest)
    models = (clf.fit(X, y) for clf in models)

    plt.close('all')
    fig, sub = plt.subplots(2, 2, figsize=(14,9))
    plt.subplots_adjust(wspace=0.4, hspace=0.4)

    X0, X1 = X[:, 0], X[:, 1]
    xx, yy = make_meshgrid(X0, X1)

    for clf, title, ax in zip(models, titles, sub.flatten()):
        plot_contours(ax, clf, xx, yy,
                      cmap=plt.cm.coolwarm, alpha=0.8)
        ax.scatter(X0, X1, c=y, cmap=plt.cm.coolwarm, s=20, edgecolors='k')
        ax.set_xlim(xx.min(), xx.max())
        ax.set_ylim(yy.min(), yy.max())
```

```
        ax.set_xlabel('Sepal length')
        ax.set_ylabel('Sepal width')
        ax.set_xticks(())
        ax.set_yticks(())
        ax.set_title(title)

    plt.show()
```

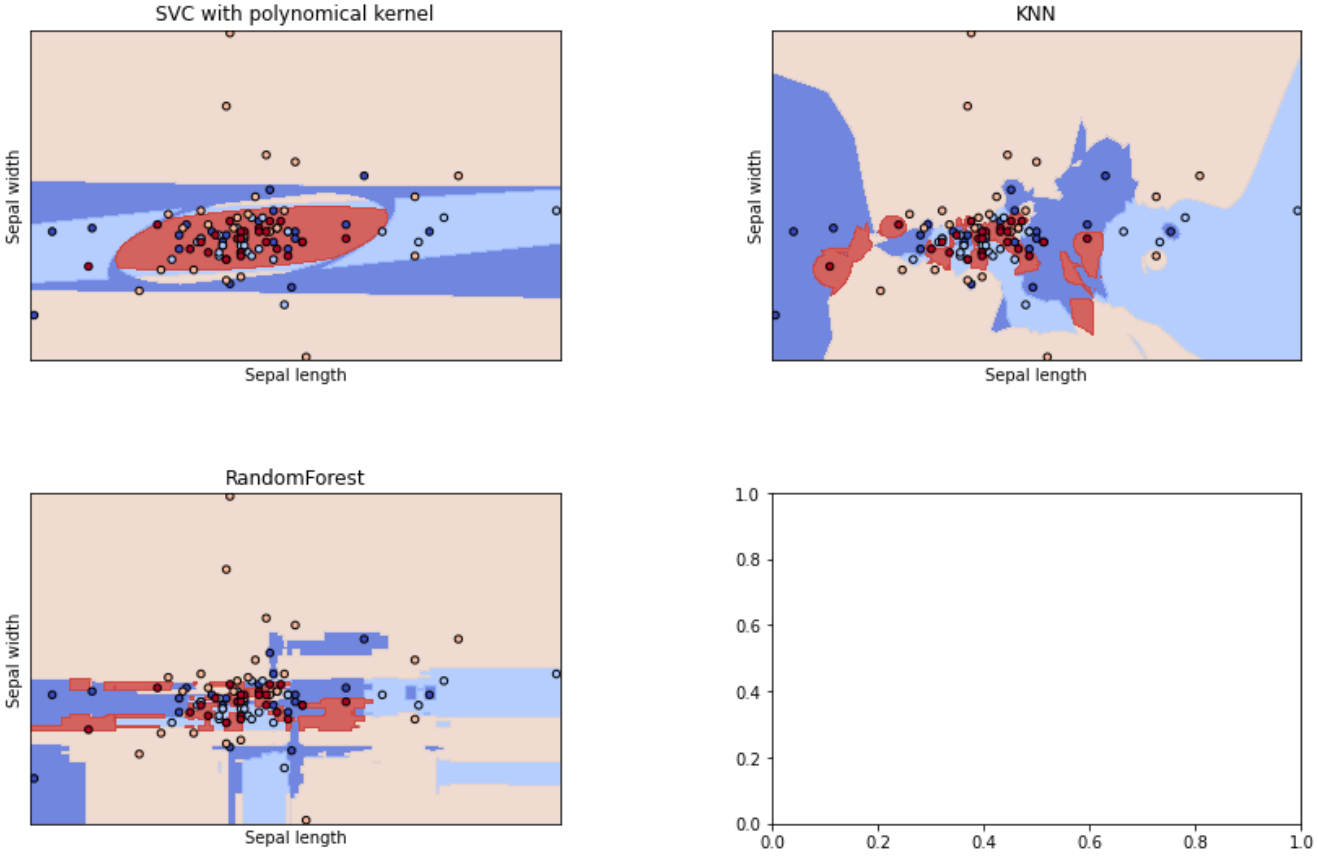
```
show_C_effect(C=0.1)
```



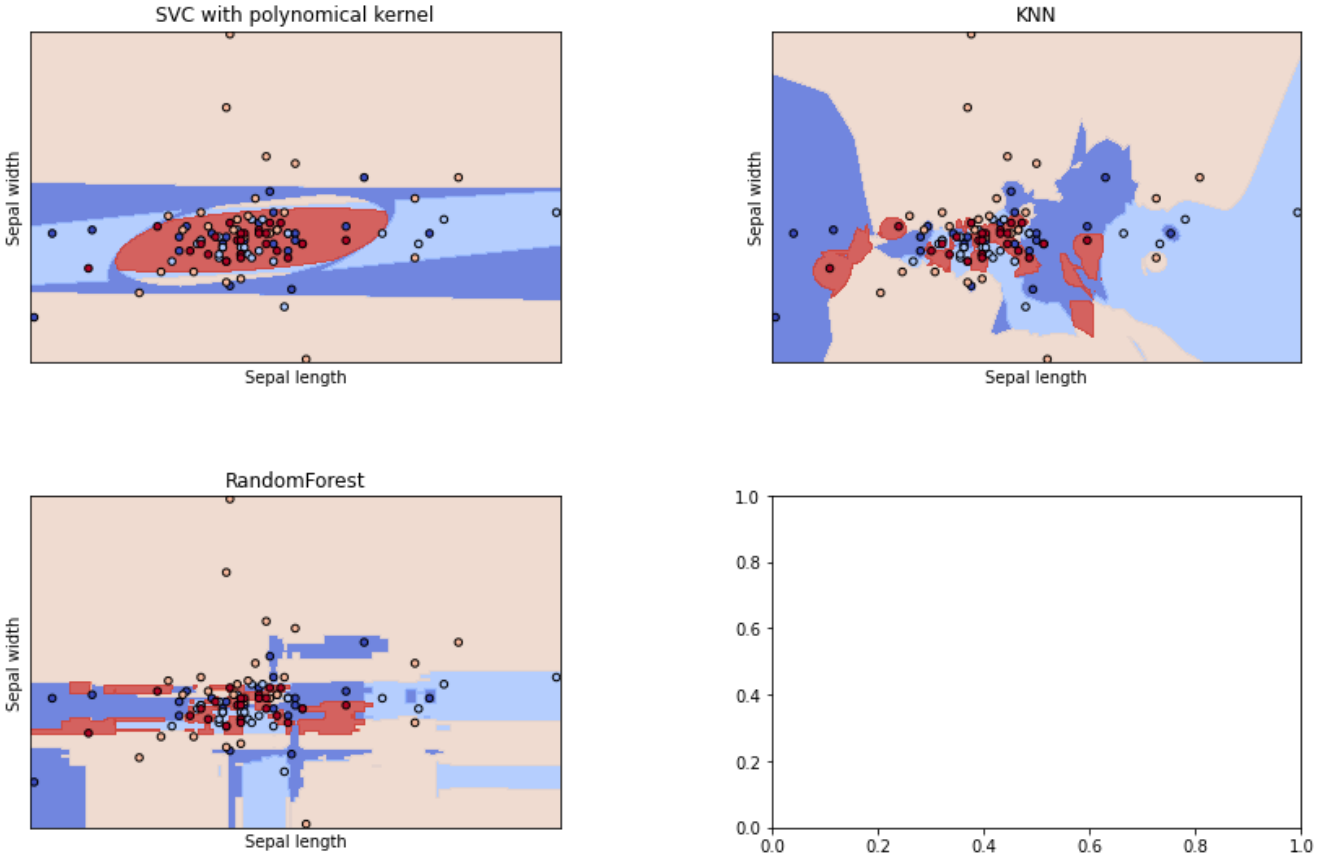

```
show_C_effect(C=0.00001)
```



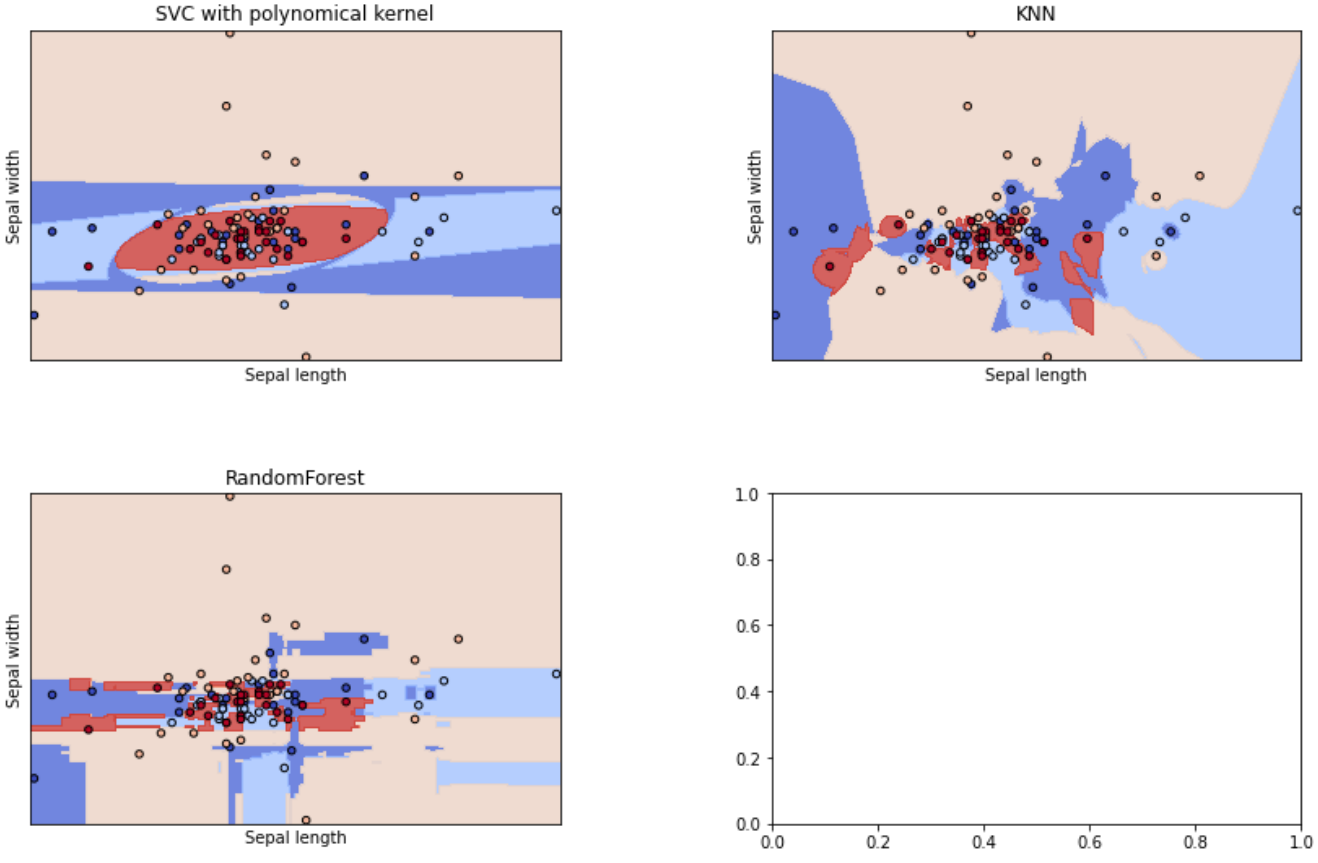

```
show_C_effect(C=1000.0)
```



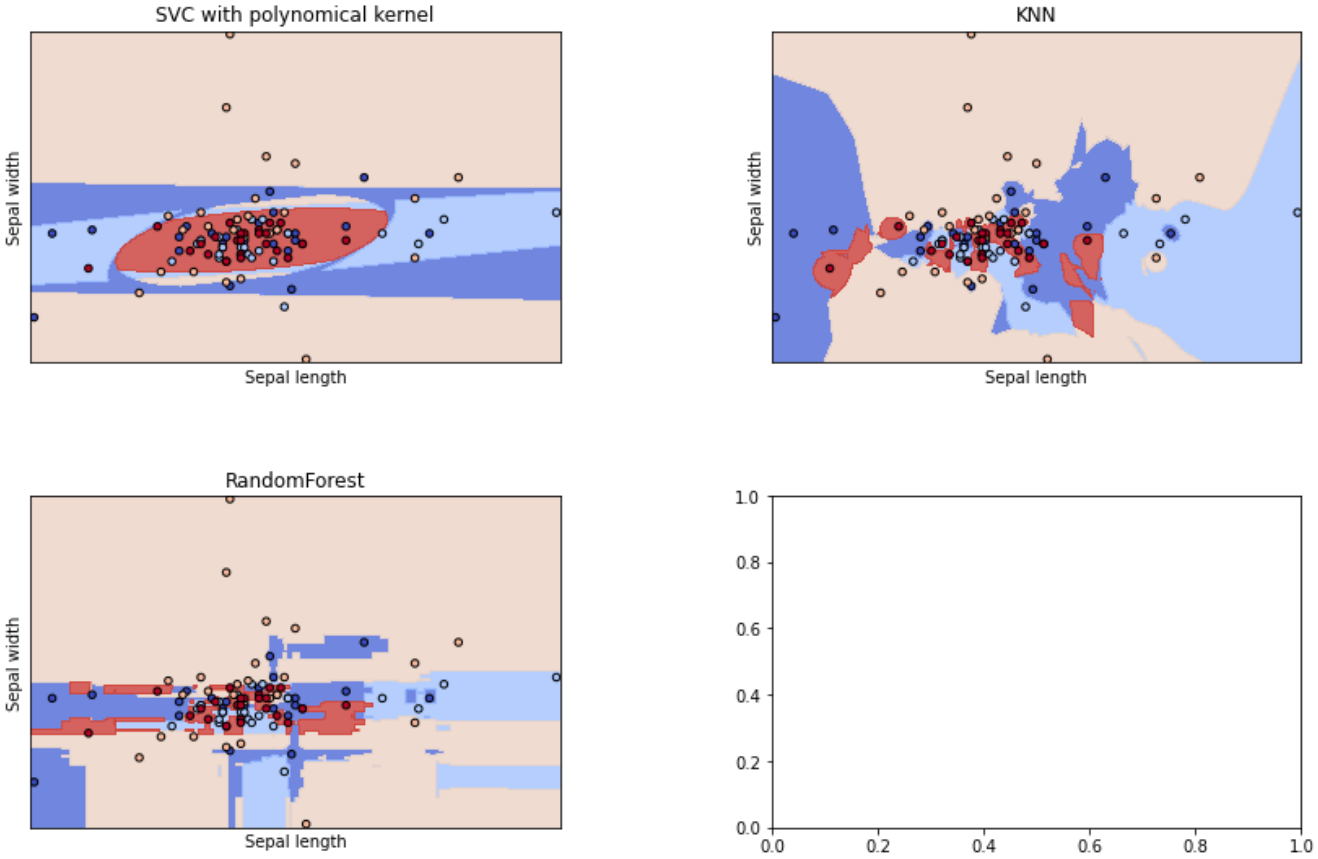

```
show_C_effect(gamma=1)
```




```
show_C_effect(gamma=100)
```




```
show_C_effect(degree=50)
```



Com podem veure, en cap model hi ha una variació per molt que canviem els paràmetres. Això creiem que és degut a que hem hagut d'agafar una mostra de 100 valors ja que sinó triga massa i no podem fer diferents proves perquè em el temps que triga és inviable.