Actividad: Problemas de clasificación

Link del Colab: https://colab.research.google.com/drive/1PLPKWQOA4Lb-Q3oSOPiK2n3af6Z7o0-6?usp=sharing

Instrucciones

Resuelve de manera individual los siguientes ejercicios en un cuaderno de Jupyter Notebook y responde a los planteamientos indicados en cada uno. Los conjuntos de datos con los que trabajarás así como algunos incisos de los ejercicios dependen de tu número de matrícula.

Ejercicio 1 (50 puntos)

En este ejercicio trabajarás con el conjunto de datos que se te asignó de acuerdo al último número de tu matrícula (ver las notas del ejercicio). En estos archivos se tienen datos procesados de un experimento de psicología en el que se mide la respuesta cerebral cuando un sujeto presta atención a un estímulo visual que aparece de manera repentina y cuando no presta atención a dicho estímulo visual. Los datos están en archivos de texto, los cuales se cargan con la función *loadtxt de numpy*. La primera columna corresponde a la clase (1 o 2). La clase 1 representa cuando el sujeto está prestando atención, y la clase 2 cuando no lo hace. La segunda columna se ignora, mientras que el resto de las columnas indican las variables que se calcularon de la respuesta cerebral medida con la técnicas de Electroencefaolografía para cada caso.

Nota: El conjunto de datos con el que trabarás en este ejercicio depende del último número de tu matrícula de acuerdo a la siguiente lista:

- 0 y 1 - P1_1.txt
- 2 y 3 - P1_2.txt
- 4 y 5 - P1_3.txt <---------- Este es el mío.
- 6 y 7 - P1_4.txt
- 8 y 9 - P1_5.txt

```
import random
import numpy as np
import pandas as pd
import numpy.linalg as ln
import matplotlib.pyplot as plt

from sklearn.svm import SVC
from sklearn.tree import DecisionTreeClassifier
```

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import StratifiedKFold
from sklearn.metrics import classification_report, accuracy_score
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.feature_selection import SelectKBest, f_classif, SequentialFeatureSelector, R


data = np.loadtxt('/content/drive/MyDrive/7mo Semestre/Colab Notebooks/DataSources/P1_3.tx
df = pd.DataFrame(data)
df
```

	0	1	2	3	4	5	6	7	8
0	1.0	1.0	0.392507	0.676570	0.601804	0.183421	-0.349747	-0.860932	-1.170066
1	1.0	1.0	-1.314876	-0.732874	0.414225	0.993659	0.827121	0.456878	0.163193
2	1.0	1.0	-1.093450	-0.689312	0.070827	0.593414	0.638874	0.450695	0.351006
3	1.0	1.0	0.467519	0.107122	-0.012376	-0.153833	-0.486099	-0.732346	-0.591912
4	1.0	1.0	0.807108	0.592778	-0.223927	-0.521974	0.147150	0.869226	0.760317
...	...	...	...	...	...	...	...	...	...
1789	2.0	1.0	0.784337	1.221407	1.425340	0.986192	-0.099067	-1.469195	-2.522046
1790	2.0	1.0	0.269954	0.582897	1.291684	2.033241	1.734560	0.210820	-1.163406
1791	2.0	1.0	-0.728536	-0.784221	0.023509	1.035219	1.233705	0.679446	0.508779
1792	2.0	1.0	1.771475	0.837355	0.181846	0.443217	0.887551	0.741103	0.326102
1793	2.0	1.0	0.479969	-0.544330	-0.752496	0.128591	1.245104	1.466514	0.620078

1794 rows × 155 columns

```
data
```

```
array([[ 1.        ,  1.        ,  0.3925073 , ...,  1.24975793,
         1.03738802,  1.05531121],
       [ 1.        ,  1.        , -1.31487611, ..., -0.61989557,
        -1.05137325, -1.19338103],
       [ 1.        ,  1.        , -1.09345032, ..., -0.29226011,
        -0.2000579 ,  0.28090627],
       ...,
       [ 2.        ,  1.        , -0.72853565, ..., -0.00920863,
         0.12140923,  0.39523656],
       [ 2.        ,  1.        ,  1.77147543, ..., -0.45705499,
        -1.52412392, -1.73872657],
       [ 2.        ,  1.        ,  0.47996947, ..., -0.18374824,
        -0.69901401, -1.41618733]])
```

```
data = np.delete(data, 1, axis=1)
df = pd.DataFrame(data)
df
```

	0	1	2	3	4	5	6	7	
0	1.0	0.392507	0.676570	0.601804	0.183421	-0.349747	-0.860932	-1.170066	-0.9
1	1.0	-1.314876	-0.732874	0.414225	0.993659	0.827121	0.456878	0.163193	-0.1
2	1.0	-1.093450	-0.689312	0.070827	0.593414	0.638874	0.450695	0.351006	0.4
3	1.0	0.467519	0.107122	-0.012376	-0.153833	-0.486099	-0.732346	-0.591912	-0.1
4	1.0	0.807108	0.592778	-0.223927	-0.521974	0.147150	0.869226	0.760317	0.2
...	...	...	...	...	...	...	...	...	
1789	2.0	0.784337	1.221407	1.425340	0.986192	-0.099067	-1.469195	-2.522046	-2.7
1790	2.0	0.269954	0.582897	1.291684	2.033241	1.734560	0.210820	-1.163406	-1.0
1791	2.0	-0.728536	-0.784221	0.023509	1.035219	1.233705	0.679446	0.508779	1.2
1792	2.0	1.771475	0.837355	0.181846	0.443217	0.887551	0.741103	0.326102	0.3
1793	2.0	0.479969	-0.544330	-0.752496	0.128591	1.245104	1.466514	0.620078	-0.3

1794 rows × 154 columns

1. Determina si es necesario balancear los datos. En caso de que sea afirmativo, en todo este ejercicio tendrás que utilizar alguna estrategia para mitigar el problema de tener una muestra desbalanceada.

```
df[0].value_counts()
```

```
2.0    1496
1.0     298
Name: 0, dtype: int64
```

Como podemos observar, tenemos mas datos de una clase que de otra, especificamente tenemos muchos mas datos de la clase 2 que de la clase 1. Así que tenemos que balancear.

```
x = data[:,1:]
y = data[:,0]
```

Utilizaremos la estrategia de balance 'Subsampling'.

2. Evalúa al menos 5 modelos de clasificación distintos utilizando validación cruzada, y determina cuál de ellos es el más efectivo.

Linear SVM Classifier

```
clf = SVC(kernel = 'linear')
kf = StratifiedKFold(n_splits=5, shuffle = True)

cv_y_test = []
cv_y_pred = []

for train_index, test_index in kf.split(x, y):

    # Training phase
    x_train = x[train_index, :]
    y_train = y[train_index]

    x1 = x_train[y_train==1, :]
    y1 = y_train[y_train==1]
    n1 = len(y1)

    x2 = x_train[y_train==2, :]
    y2 = y_train[y_train==2]
    n2 = len(y2)

    ind = random.sample([i for i in range(n2)], n1)

    x_sub = np.concatenate((x1, x2[ind,:]), axis=0)
    y_sub = np.concatenate((y1, y2[ind]), axis=0)

    clf.fit(x_sub, y_sub)

    # Test phase
    x_test = x[test_index, :]
    y_test = y[test_index]
    y_pred = clf.predict(x_test)

    cv_y_test.append(y_test)
    cv_y_pred.append(y_pred)

svm_report = classification_report(np.concatenate(cv_y_test), np.concatenate(cv_y_pred))
```

KNN

```
clf = KNeighborsClassifier(n_neighbors=3)
kf = StratifiedKFold(n_splits=5, shuffle = True)

cv_y_test = []
cv_y_pred = []

for train_index, test_index in kf.split(x, y):

    x_train = x[train_index, :]
    y_train = y[train_index]

    x1 = x_train[y_train==1, :]
    y1 = y_train[y_train==1]
```

```
        n1 = len(y1)

        x2 = x_train[y_train==2, :]
        y2 = y_train[y_train==2]
        n2 = len(y2)

        ind = random.sample([i for i in range(n2)], n1)

        x_sub = np.concatenate((x1, x2[ind,:]), axis=0)
        y_sub = np.concatenate((y1, y2[ind]), axis=0)

        clf.fit(x_sub, y_sub)

        x_test = x[test_index, :]
        y_test = y[test_index]
        y_pred = clf.predict(x_test)

        cv_y_test.append(y_test)
        cv_y_pred.append(y_pred)

    knn_report = classification_report(np.concatenate(cv_y_test), np.concatenate(cv_y_pred))
```

Decision tree

```
    clf = DecisionTreeClassifier()
    kf = StratifiedKFold(n_splits=5, shuffle = True)

    cv_y_test = []
    cv_y_pred = []

    for train_index, test_index in kf.split(x, y):

        x_train = x[train_index, :]
        y_train = y[train_index]

        x1 = x_train[y_train==1, :]
        y1 = y_train[y_train==1]
        n1 = len(y1)

        x2 = x_train[y_train==2, :]
        y2 = y_train[y_train==2]
        n2 = len(y2)

        ind = random.sample([i for i in range(n2)], n1)

        x_sub = np.concatenate((x1, x2[ind,:]), axis=0)
        y_sub = np.concatenate((y1, y2[ind]), axis=0)

        clf.fit(x_sub, y_sub)

        x_test = x[test_index, :]
        y_test = y[test_index]
        y_pred = clf.predict(x_test)
```

```
    cv_y_test.append(y_test)
    cv_y_pred.append(y_pred)

tree_report = classification_report(np.concatenate(cv_y_test), np.concatenate(cv_y_pred))
```

Linear Discriminant Analysis

```
clf = LinearDiscriminantAnalysis()
kf = StratifiedKFold(n_splits=5, shuffle = True)

cv_y_test = []
cv_y_pred = []

for train_index, test_index in kf.split(x, y):

    x_train = x[train_index, :]
    y_train = y[train_index]

    x1 = x_train[y_train==1, :]
    y1 = y_train[y_train==1]
    n1 = len(y1)

    x2 = x_train[y_train==2, :]
    y2 = y_train[y_train==2]
    n2 = len(y2)

    ind = random.sample([i for i in range(n2)], n1)

    x_sub = np.concatenate((x1, x2[ind,:]), axis=0)
    y_sub = np.concatenate((y1, y2[ind]), axis=0)

    clf.fit(x_sub, y_sub)

    x_test = x[test_index, :]
    y_test = y[test_index]
    y_pred = clf.predict(x_test)

    cv_y_test.append(y_test)
    cv_y_pred.append(y_pred)

lda_report = classification_report(np.concatenate(cv_y_test), np.concatenate(cv_y_pred))
```

RBF-SVM

```
clf = SVC(kernel = 'rbf')
kf = StratifiedKFold(n_splits=5, shuffle = True)

cv_y_test = []
cv_y_pred = []
```

```
for train_index, test_index in kf.split(x, y):

    x_train = x[train_index, :]
    y_train = y[train_index]

    x1 = x_train[y_train==1, :]
    y1 = y_train[y_train==1]
    n1 = len(y1)

    x2 = x_train[y_train==2, :]
    y2 = y_train[y_train==2]
    n2 = len(y2)

    ind = random.sample([i for i in range(n2)], n1)

    x_sub = np.concatenate((x1, x2[ind,:]), axis=0)
    y_sub = np.concatenate((y1, y2[ind]), axis=0)

    clf.fit(x_sub, y_sub)

    x_test = x[test_index, :]
    y_test = y[test_index]
    y_pred = clf.predict(x_test)

    cv_y_test.append(y_test)
    cv_y_pred.append(y_pred)

rbf_report = classification_report(np.concatenate(cv_y_test), np.concatenate(cv_y_pred))
```

Ya que hicimos los 5 modelos, vamos a evaluar los resultados:

```
print('Linear SVM:\n', svm_report, '\n')
print('KNN:\n', knn_report, '\n')
print('Decision Tree:\n', tree_report, '\n')
print('Linear Discriminant Analysis:\n', lda_report, '\n')
print('RBF-SVM:\n', rbf_report, '\n')
```

```
Linear SVM:
                precision    recall  f1-score   support

           1.0       0.50      0.86      0.64       298
           2.0       0.97      0.83      0.89      1496

      accuracy                           0.84      1794
     macro avg       0.74      0.84      0.76      1794
  weighted avg       0.89      0.84      0.85      1794


KNN:
                precision    recall  f1-score   support

           1.0       0.34      0.82      0.48       298
           2.0       0.95      0.69      0.80      1496

      accuracy                           0.71      1794
```

```
   macro avg       0.65      0.75      0.64      1794
weighted avg       0.85      0.71      0.75      1794


Decision Tree:
              precision    recall  f1-score   support

         1.0       0.31      0.66      0.42       298
         2.0       0.91      0.71      0.80      1496

    accuracy                           0.70      1794
   macro avg       0.61      0.69      0.61      1794
weighted avg       0.81      0.70      0.73      1794


Linear Discriminant Analysis:
              precision    recall  f1-score   support

         1.0       0.47      0.82      0.60       298
         2.0       0.96      0.82      0.88      1496

    accuracy                           0.82      1794
   macro avg       0.71      0.82      0.74      1794
weighted avg       0.88      0.82      0.83      1794


RBF-SVM:
              precision    recall  f1-score   support

         1.0       0.55      0.86      0.67       298
         2.0       0.97      0.86      0.91      1496

    accuracy                           0.86      1794
   macro avg       0.76      0.86      0.79      1794
weighted avg       0.90      0.86      0.87      1794
```

Si vemos estos resultados, nos podemos dar cuenta en base a la precisión y el recall que tan buenos son los modelos.

La precisión nos dice cuántos de los casos clasificados como positivos realmente lo son.

El recall nos dice cuántos casos positivos reales el modelo logró identificar, o sea, cuán bien atrapa todos los casos positivos. Este es muy importante que se mantenga similar para todas las clases, porque si no, puede significar que nuestros modelo está desbalanceado.

Así que buscaremos el modelo que nos ofrezca el mayor balance de las clases en su recall y la mayor precision promedio entre clases.

Y entre estos modelos, el ganador es el RBF-SVM, con un recall de 0.86 en cada clase y una precision promedio de 0.76.

3. Implementa desde cero el método de regresión logística, y evalúalo con el conjunto de datos.

Funciones para el modelo de regresión logistica:

```
def grad(X, y, beta):
    n = len(y)
    xbeta = X @ beta
    exp = np.exp(-xbeta)
    res = y - 1/(1+exp)
    tmp = (exp/((1+exp)**2)) * res*X.transpose()
    return -(2/n)*tmp.sum(axis = 1)

def predict(X, beta):
    xbeta = X @ beta
    tmp = 1./(1.+np.exp(-xbeta))
    return (tmp > 0.5).astype("int32")

def fit_model(X, y, alpha = 0.005, maxit = 10000):

    # Number of predictors
    npredictors = X.shape[1]

    # Initialize beta
    beta = 2*np.random.rand(npredictors) - 1.0

    # Optimization algorithm
    it = 0
    while (ln.norm(grad(X, y, beta)) > 1e-4) and (it < maxit):
        beta = beta - alpha*grad(X, y, beta)
        it = it + 1
        #print(beta)

    return beta


X = np.column_stack((np.ones(x.shape[0]), x))

kf = StratifiedKFold(n_splits=5, shuffle = True)

cv_y_test = []
cv_y_pred = []

for train_index, test_index in kf.split(x, y):

    x_train = x[train_index, :]
    y_train = y[train_index]

    x1 = x_train[y_train==1, :]
    y1 = y_train[y_train==1]
    n1 = len(y1)
```

```
    x2 = x_train[y_train==2, :]
    y2 = y_train[y_train==2]
    n2 = len(y2)

    ind = random.sample([i for i in range(n2)], n1)

    x_sub = np.concatenate((x1, x2[ind,:]), axis=0)
    y_sub = np.concatenate((y1, y2[ind]), axis=0)

    beta_cv = fit_model(x_sub, y_sub)

    x_test = x[test_index, :]
    y_test = y[test_index]
    y_pred = predict(x_test, beta_cv)

    cv_y_test.append(y_test)
    cv_y_pred.append(y_pred)

print(classification_report(np.concatenate(cv_y_test), np.concatenate(cv_y_pred)))
```

```
              precision    recall  f1-score   support

         0.0       0.00      0.00      0.00         0
         1.0       0.19      0.86      0.31       298
         2.0       0.00      0.00      0.00      1496

    accuracy                           0.14      1794
   macro avg       0.06      0.29      0.10      1794
weighted avg       0.03      0.14      0.05      1794

/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344: Unde
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344: Unde
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344: Unde
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344: Unde
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344: Unde
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344: Unde
  _warn_prf(average, modifier, msg_start, len(result))
```

Como podemos observar, no es un muy buen modelo, nos esta dando una precision y un recall bastante malos. De hecho hasta nos creó una tercera clase.

4. Con alguno de los clasificadores que probaste en los pasos anteriores, determina el número óptimo de características utilizando un método tipo Filter.

Arbol de decisión con metodo Filter:

```
clf = DecisionTreeClassifier()
acc_nfeat = []
n_feats = []

for i in range(1, df.shape[1]): # Creating the list with the number of features
  n_feats.append(i)

for n_feat in n_feats:
    print('---- n features =', n_feat)

    acc_cv = []

    kf = StratifiedKFold(n_splits=5, shuffle = True)

    for train_index, test_index in kf.split(x, y):

        # Training phase
        x_train = x[train_index, :]
        y_train = y[train_index]

        x1 = x_train[y_train==1, :]
        y1 = y_train[y_train==1]
        n1 = len(y1)

        x2 = x_train[y_train==2, :]
        y2 = y_train[y_train==2]
        n2 = len(y2)

        ind = random.sample([i for i in range(n2)], n1)

        x_sub = np.concatenate((x1, x2[ind,:]), axis=0)
        y_sub = np.concatenate((y1, y2[ind]), axis=0)

        fselection_cv = SelectKBest(f_classif, k = n_feat)
        fselection_cv.fit(x_sub, y_sub)
        x_sub = fselection_cv.transform(x_sub)

        clf.fit(x_sub, y_sub)

        # Test phase
        x_test = fselection_cv.transform(x[test_index, :])
        y_test = y[test_index]
        y_pred = clf.predict(x_test)


        acc_i = accuracy_score(y_test, y_pred)
        acc_cv.append(acc_i)

    acc = np.average(acc_cv)
    acc_nfeat.append(acc)
```

```
        print('ACC:', acc)

    opt_index = np.argmax(acc_nfeat)
    opt_features = n_feats[opt_index]
    print("\nOptimal number of features: ", opt_features)

    plt.plot(n_feats, acc_nfeat)
    plt.xlabel("features")
    plt.ylabel("Accuracy")

    plt.show()
```

```
---- n features = 1
ACC: 0.5663263877001603
---- n features = 2
ACC: 0.608684894414964
---- n features = 3
ACC: 0.6147990227354072
---- n features = 4
ACC: 0.6376542537464402
---- n features = 5
ACC: 0.6610805932058325
---- n features = 6
ACC: 0.675036180576088
---- n features = 7
ACC: 0.7051244144971289
---- n features = 8
ACC: 0.6890073294844463
---- n features = 9
ACC: 0.7179517903549586
---- n features = 10
ACC: 0.7207466425981544
---- n features = 11
ACC: 0.7151429327274708
---- n features = 12
ACC: 0.7179284480478051
---- n features = 13
ACC: 0.7307822785204089
---- n features = 14
ACC: 0.7257745755590483
---- n features = 15
ACC: 0.7274272109055258
---- n features = 16
ACC: 0.7251770124959152
---- n features = 17
ACC: 0.73855059834114
---- n features = 18
ACC: 0.7502544311479747
---- n features = 19
ACC: 0.7262974432392898
---- n features = 20
ACC: 0.7330200276995379
---- n features = 21
ACC: 0.7263207855464434
---- n features = 22
ACC: 0.7123869843295311
---- n features = 23
ACC: 0.7329842361619022
---- n features = 24
ACC: 0.7045641991254415
---- n features = 25
ACC: 0.7491651234808049
---- n features = 26
ACC: 0.7335304461492973
---- n features = 27
ACC: 0.7558316864038842
---- n features = 28
ACC: 0.7251910178802073
---- n features = 29
ACC: 0.7302189508411011
---- n features = 30
ACC: 0.7357930937893901
```

```
---- n features = 31
ACC: 0.7397005960069093
---- n features = 32
ACC: 0.7564074633136739
---- n features = 33
ACC: 0.7402374690714432
---- n features = 34
ACC: 0.7173869065218407
---- n features = 35
ACC: 0.7407992405969407
---- n features = 36
ACC: 0.7408210267502839
---- n features = 37
ACC: 0.736343972238216
---- n features = 38
ACC: 0.7135011904576648
---- n features = 39
ACC: 0.7107172312911408
---- n features = 40
ACC: 0.7279874262772132
---- n features = 41
ACC: 0.7318964846485427
---- n features = 42
ACC: 0.7151662750346244
---- n features = 43
ACC: 0.732982680008092
---- n features = 44
ACC: 0.717396243444702
---- n features = 45
ACC: 0.7291171939434494
---- n features = 46
ACC: 0.719615318778108
---- n features = 47
ACC: 0.7263005555469102
---- n features = 48
ACC: 0.7184902195733025
---- n features = 49
ACC: 0.7039744168313596
---- n features = 50
ACC: 0.7212990772007906
---- n features = 51
ACC: 0.6923001509469197
---- n features = 52
ACC: 0.7235306017646784
---- n features = 53
ACC: 0.7112416551251926
---- n features = 54
ACC: 0.7296649600846548
---- n features = 55
ACC: 0.7168360280730148
---- n features = 56
ACC: 0.7318762546490095
---- n features = 57
ACC: 0.7252174724949815
---- n features = 58
ACC: 0.7318762546490095
---- n features = 59
ACC: 0.6923312740231243
---- n features = 60
ACC: 0.69454723704891
```

```
---- n features = 61
ACC: 0.7430478828527411
---- n features = 62
ACC: 0.6989853877157218
---- n features = 63
ACC: 0.6889730941006209
---- n features = 64
ACC: 0.7196168749319184
---- n features = 65
ACC: 0.7347037861222203
---- n features = 66
ACC: 0.7268716639952693
---- n features = 67
ACC: 0.7279500785857674
---- n features = 68
ACC: 0.7151476011889015
---- n features = 69
ACC: 0.7023264499463127
---- n features = 70
ACC: 0.7179237795863743
---- n features = 71
ACC: 0.7201957641493284
---- n features = 72
ACC: 0.6989978369462038
---- n features = 73
ACC: 0.7090428097913197
---- n features = 74
ACC: 0.7173837942142202
---- n features = 75
ACC: 0.7017864645741585
---- n features = 76
ACC: 0.7112774466628282
---- n features = 77
ACC: 0.7073683882914986
---- n features = 78
ACC: 0.6934112447674329
---- n features = 79
ACC: 0.6995518277026502
---- n features = 80
ACC: 0.7168095734582407
---- n features = 81
ACC: 0.6995705015483731
---- n features = 82
ACC: 0.7068081729198115
---- n features = 83
ACC: 0.7218592925724778
---- n features = 84
ACC: 0.7179377849706665
---- n features = 85
ACC: 0.7112447674328131
---- n features = 86
ACC: 0.7363361914691648
---- n features = 87
ACC: 0.7140411758298191
---- n features = 88
ACC: 0.7079021490484119
---- n features = 89
ACC: 0.7051321952661802
---- n features = 90
ACC: 0.7251972424954483
```

```
---- n features = 91
ACC: 0.7162851496241889
---- n features = 92
ACC: 0.7118158758811721
---- n features = 93
ACC: 0.6906459594466317
---- n features = 94
ACC: 706260406778606
---- n features = 95
ACC: 0.7090428097913197
---- n features = 96
ACC: 0.7212881841241188
---- n features = 97
ACC: 0.7006707022922145
---- n features = 98
ACC: 0.7207279687524315
---- n features = 99
ACC: 0.7101243366894383
---- n features = 100
ACC: 0.7157202657910708
---- n features = 101
ACC: 0.7073683882914988
---- n features = 102
ACC: 0.7006598092155428
---- n features = 103
ACC: 0.7040428875990103
---- n features = 104
ACC: 0.6950763293443925
---- n features = 105
ACC: 0.6995300415493066
---- n features = 106
ACC: 0.7012153561257995
---- n features = 107
ACC: 706811285227432
---- n features = 108
ACC: 0.7140738550598342
---- n features = 109
ACC: 0.7179595711240099
---- n features = 110
ACC: 0.7129176327788238
---- n features = 111
ACC: 0.7073715005991192
---- n features = 112
ACC: 721840618726755
---- n features = 113
ACC: 0.7201802026112262
---- n features = 114
ACC: 0.7251972424954483
---- n features = 115
ACC: 0.7045564183563903
---- n features = 116
ACC: 0.6928525855495555
---- n features = 117
ACC: 0.7296447300851217
---- n features = 118
ACC: 0.7151802804189166
---- n features = 119
ACC: 0.7307698292899271
---- n features = 120
ACC: 0.7213068579698417
---- n features = 121
```

```
ACC: 0.710160128227074
---- n features = 122
ACC: 0.6956427693313207
---- n features = 123
ACC: 0.7045673114330621
---- n features = 124
ACC: 0.6962418885482641
---- n features = 125
ACC: 0.6844960395885529
---- n features = 126
ACC: 0.7118143197273619
---- n features = 127
ACC: 0.7146029473553166
---- n features = 128
ACC: 0.691167270973063
---- n features = 129
ACC: 0.6989853877157218
---- n features = 130
ACC: 0.686170461088374
---- n features = 131
ACC: 0.7084748136505812
---- n features = 132
ACC: 0.7134856289195624
---- n features = 133
ACC: 0.7012153561257993
---- n features = 134
ACC: 0.7157062604067785
---- n features = 135
ACC: 0.7095781267020433
---- n features = 136
ACC: 0.7112603289709155
---- n features = 137
ACC: 0.7129363066245468
---- n features = 138
ACC: 0.6928556978571763
---- n features = 139
ACC: 0.7140598496755419
---- n features = 140
ACC: 0.6956536624079923
---- n features = 141
ACC: 0.7134996343038547
---- n features = 142
ACC: 0.7123900966371516
---- n features = 143
ACC: 0.691719705575699
```

5. Repite el paso anterior, pero para un método de selección de características de tipo Wrapper.

Por cuestiones de tiempo, a los metodos siguientes metodos se mostrará unicamente su código, no sus resultados, ya que con tantas caracteristicas, tarda mucho en poder procesar todas.

Arbol de decisión con metodo Wrapper:

```
Acc: 0.6833693842299373
```

```
clf = DecisionTreeClassifier()
acc_nfeat = []
n_feats = []

for i in range(1, 41): # Creating the list with the number of features
  n_feats.append(i)

for n_feat in n_feats:
    print('---- n features =', n_feat)

    acc_cv = []

    kf = StratifiedKFold(n_splits=5, shuffle = True)

    for train_index, test_index in kf.split(x, y):

        # Training phase
        x_train = x[train_index, :]
        y_train = y[train_index]

        x1 = x_train[y_train==1, :]
        y1 = y_train[y_train==1]
        n1 = len(y1)

        x2 = x_train[y_train==2, :]
        y2 = y_train[y_train==2]
        n2 = len(y2)

        ind = random.sample([i for i in range(n2)], n1)

        x_sub = np.concatenate((x1, x2[ind,:]), axis=0)
        y_sub = np.concatenate((y1, y2[ind]), axis=0)

        fselection_cv = SequentialFeatureSelector(clf, n_features_to_select=n_feat)
        fselection_cv.fit(x_sub, y_sub)
        x_sub = fselection_cv.transform(x_sub)

        clf.fit(x_sub, y_sub)

        # Test phase
        x_test = fselection_cv.transform(x[test_index, :])
        y_test = y[test_index]
        y_pred = clf.predict(x_test)


        acc_i = accuracy_score(y_test, y_pred)
        acc_cv.append(acc_i)

    acc = np.average(acc_cv)
    acc_nfeat.append(acc)
```

```
    print('ACC:', acc)

opt_index = np.argmax(acc_nfeat)
opt_features = n_feats[opt_index]
print("\nOptimal number of features: ", opt_features)

plt.plot(n_feats, acc_nfeat)
plt.xlabel("features")
plt.ylabel("Accuracy")

plt.show()
```

6. Repite el paso 4, pero para un método de selección de características de tipo Filter-Wrapper.

Arbol de decisión con metodo Filter-Wrapper:

```
clf = DecisionTreeClassifier()
acc_nfeat = []
n_feats = []

for i in range(1, 41): # Creating the list with the number of features
  n_feats.append(i)

for n_feat in n_feats:
    print('---- n features =', n_feat)

    acc_cv = []

    kf = StratifiedKFold(n_splits=5, shuffle = True)

    for train_index, test_index in kf.split(x, y):

        # Training phase
        x_train = x[train_index, :]
        y_train = y[train_index]

        x1 = x_train[y_train==1, :]
        y1 = y_train[y_train==1]
        n1 = len(y1)

        x2 = x_train[y_train==2, :]
        y2 = y_train[y_train==2]
        n2 = len(y2)

        ind = random.sample([i for i in range(n2)], n1)

        x_sub = np.concatenate((x1, x2[ind,:]), axis=0)
        y_sub = np.concatenate((y1, y2[ind]), axis=0)

        fselection_cv = RFE(clf, n_features_to_select=n_feat)
```

```
            fselection_cv.fit(x_sub, y_sub)
            x_sub = fselection_cv.transform(x_sub)

            clf.fit(x_sub, y_sub)

            # Test phase
            x_test = fselection_cv.transform(x[test_index, :])
            y_test = y[test_index]
            y_pred = clf.predict(x_test)


            acc_i = accuracy_score(y_test, y_pred)
            acc_cv.append(acc_i)

        acc = np.average(acc_cv)
        acc_nfeat.append(acc)

        print('ACC:', acc)

    opt_index = np.argmax(acc_nfeat)
    opt_features = n_feats[opt_index]
    print("\nOptimal number of features: ", opt_features)

    plt.plot(n_feats, acc_nfeat)
    plt.xlabel("features")
    plt.ylabel("Accuracy")

    plt.show()
```

7. Escoge alguna de las técnicas de selección de características que probaste con anterioridad, y con el número óptimo de características encontrado, prepara tu modelo para producción haciendo lo siguiente:

- Aplica el método de selección de características con todos los datos.
- Ajusta el modelo con las características encontradas.

Clasificador SVM-RBF con metodo Filter:

```
clf = SVC(kernel = 'rbf')

x1 = x[y==1, :]
y1 = y[y==1]
n1 = len(y1)

x2 = x[y==2, :]
y2 = y[y==2]
n2 = len(y2)

ind = random.sample([i for i in range(n2)], n1)
```

```
x_sub = np.concatenate((x1, x2[ind,:]), axis=0)
y_sub = np.concatenate((y1, y2[ind]), axis=0)

fselection = SelectKBest(f_classif, k = 32)
fselection.fit(x_sub, y_sub)

print("Selected features: ", fselection.get_feature_names_out())

x_transformed = fselection.transform(x)
clf.fit(x_transformed, y)
```

```
Selected features:  ['x10' 'x11' 'x12' 'x13' 'x15' 'x16' 'x17' 'x18' 'x19' 'x20' 'x21
 'x25' 'x26' 'x27' 'x28' 'x29' 'x30' 'x61' 'x64' 'x65' 'x66' 'x76' 'x77'
 'x78' 'x104' 'x105' 'x111' 'x112' 'x113' 'x114' 'x128']
```

▾ SVC

SVC()

8. Contesta las siguientes preguntas:

- ¿Qué pasa si no se considera el problema de tener datos desbalanceados para este caso? ¿Por qué?

  Nuestro modelo podría quedar desbalanceado, o sea, que si una clase tiene muchas muestras y la otra no tantas, nuestro modelo se volverá muy bueno prediciendo la clase con mas muestras y no tanto con la clase con menos muestras. Tambien el modelo será más propenso a dar falsos positivos.

  Y esto es porque el modelo al tener mas muestras de una clase, tiene mas oportunidad de entrenarse en dicha clase, y tiende a ignorar en menor o mayor medida a la clase con menos nuestras.

- De todos los clasificadores, ¿cuál o cuales consideras que son adecuados para los datos? ¿Qué propiedades tienen dichos modelos que los hacen apropiados para los datos? Argumenta tu respuesta.

  El clasificador que considero mejor para los datos es el SVM-RBF. La razón es que tiene un recall muy bueno, bastante alto y sobre todo, balanceado entre ambas clases, y su precisión promedio fue la más alta entre los 5 clasificadores.

- ¿Es posibles reducir la dimensionalidad del problema sin perder rendimiento en el modelo? ¿Por qué?

  Depende, cuando seleccionamos algunas caracteristicas para descartar otras, estamos optimizando el modelo, haciendolo que sea mejor en sus predicciones; pero tambien cuando quitamos algunos datos (filas) para balancear el conjunto de datos, estamos perdiendo información util que afectan el rendimiento del modelo. Pero al final de cuentas, todos estos procedimientos son para mejorar el modelo, así que podríamos decir que sí es posible.

- ¿Qué método de selección de características consideras el más adecuado para este caso? ¿Por qué?

  Por la eficiencia y tiempo con los datos, yo escogería el metodo Filter.

- Si quisieras mejorar el rendimiento de tus modelos, ¿qué más se podría hacer?

  Podríamos evaluar mas modelos, y el mejor modelo lo evaluamos con cada uno de los metodos de seleccion de caracteristicas y someterlos a metricas para ver cual es el mejor modelo con las mejores caracteristicas. Y dependiendo del modelo, podríamos calcular sus mejores hiperparametros.

Ejercicio 2 (50 puntos)

En este ejercicio trabajarás con datos que vienen de un experimento en el que se midió actividad muscular con la técnica de la Electromiografía en el brazo derecho de varios participantes cuando éstos realizaban un movimiento con la mano entre siete posible (Flexionar hacia arriba, Flexionar hacia abajo, Cerrar la mano, Estirar la mano, Abrir la mano, Coger un objeto, No moverse). A su vez, la primera columna corresponde a la clase (1, 2, 3, 4, 5, 6, y 7), la segunda columna se ignora, y el resto de las columnas indican las variables que se calcularon de la respuesta muscular. El archivo de datos con el que trabajarás depende de tu matrícula.

```
data = np.loadtxt('/content/drive/MyDrive/7mo Semestre/Colab Notebooks/DataSources/M_5.txt
df = pd.DataFrame(data)
df
```

	0	1	2	3	4	5	6	7	8
0	1.0	1.0	0.159910	0.829038	-0.236322	-1.137015	0.049065	-1.331090	0.081879
1	1.0	1.0	-1.039646	0.061581	-0.372804	-0.315868	0.351879	-1.399993	-0.981714
2	1.0	1.0	-1.411644	-1.090915	-1.164213	-1.041624	0.055639	-2.163669	-1.410827
3	1.0	1.0	-2.645974	0.129788	2.822250	-1.345104	-0.196804	3.303073	-3.014899
4	1.0	1.0	-1.692860	-1.597632	-2.690969	-0.413617	-1.163711	-2.757666	-1.719198
...	...	...	...	...	...	...	...	...	...
624	7.0	1.0	-7.042298	-6.180162	-4.875976	-7.048198	-7.099642	-4.785359	-6.666241
625	7.0	1.0	-6.826647	-5.981072	-4.807238	-7.361130	-6.843612	-4.523671	-6.360528
626	7.0	1.0	-7.717987	-5.548352	-4.399172	-6.311186	-7.481042	-4.227248	-6.762767
627	7.0	1.0	-7.447476	-5.223852	-4.073834	-6.553809	-7.143856	-3.894904	-6.507190
628	7.0	1.0	-8.059398	-5.362028	-4.429245	-6.694931	-7.674423	-4.078131	-6.847052

629 rows × 632 columns

```
data = np.delete(data, 1, axis=1)
df = pd.DataFrame(data)
df
```

	0	1	2	3	4	5	6	7	
0	1.0	0.159910	0.829038	-0.236322	-1.137015	0.049065	-1.331090	0.081879	0.71
1	1.0	-1.039646	0.061581	-0.372804	-0.315868	0.351879	-1.399993	-0.981714	-0.27
2	1.0	-1.411644	-1.090915	-1.164213	-1.041624	0.055639	-2.163669	-1.410827	-1.20
3	1.0	-2.645974	0.129788	2.822250	-1.345104	-0.196804	3.303073	-3.014899	0.29
4	1.0	-1.692860	-1.597632	-2.690969	-0.413617	-1.163711	-2.757666	-1.719198	-1.59
...	...	...	...	...	...	...	...	...	
624	7.0	-7.042298	-6.180162	-4.875976	-7.048198	-7.099642	-4.785359	-6.666241	-5.60
625	7.0	-6.826647	-5.981072	-4.807238	-7.361130	-6.843612	-4.523671	-6.360528	-5.46
626	7.0	-7.717987	-5.548352	-4.399172	-6.311186	-7.481042	-4.227248	-6.762767	-5.01
627	7.0	-7.447476	-5.223852	-4.073834	-6.553809	-7.143856	-3.894904	-6.507190	-4.69
628	7.0	-8.059398	-5.362028	-4.429245	-6.694931	-7.674423	-4.078131	-6.847052	-4.88

629 rows × 631 columns

1. Determina si es necesario balancear los datos. En caso de que sea afirmativo, en todo este ejercicio tendrás que utilizar alguna estrategia para mitigar el problema de tener una muestra desbalanceada.

```
df[0].value_counts()
```

```
1.0    90
2.0    90
3.0    90
4.0    90
5.0    90
6.0    90
7.0    89
Name: 0, dtype: int64
```

Nuestras muestras estan balanceadas por clases.

2. Evalúa al menos 5 modelos de clasificación distintos utilizando validación cruzada, y determina cuál de ellos es el más efectivo.

```
x = data[:,1:]
y = data[:,0]
```

Linear SVM Classifier

```
clf = SVC(kernel = 'linear')
kf = StratifiedKFold(n_splits=5, shuffle = True)

cv_y_test = []
cv_y_pred = []

for train_index, test_index in kf.split(x, y):

    # Training phase
    x_train = x[train_index, :]
    y_train = y[train_index]

    clf.fit(x_train, y_train)

    # Test phase
    x_test = x[test_index, :]
    y_test = y[test_index]
    y_pred = clf.predict(x_test)

    cv_y_test.append(y_test)
    cv_y_pred.append(y_pred)

svm_report = classification_report(np.concatenate(cv_y_test), np.concatenate(cv_y_pred))
```

KNN

```
clf = KNeighborsClassifier(n_neighbors=3)
kf = StratifiedKFold(n_splits=5, shuffle = True)

cv_y_test = []
cv_y_pred = []

for train_index, test_index in kf.split(x, y):

    x_train = x[train_index, :]
    y_train = y[train_index]

    clf.fit(x_train, y_train)

    x_test = x[test_index, :]
    y_test = y[test_index]
    y_pred = clf.predict(x_test)

    cv_y_test.append(y_test)
    cv_y_pred.append(y_pred)
```

```
knn_report = classification_report(np.concatenate(cv_y_test), np.concatenate(cv_y_pred))
```

Decision tree

```
clf = DecisionTreeClassifier()
kf = StratifiedKFold(n_splits=5, shuffle = True)

cv_y_test = []
cv_y_pred = []

for train_index, test_index in kf.split(x, y):

    x_train = x[train_index, :]
    y_train = y[train_index]

    clf.fit(x_train, y_train)

    x_test = x[test_index, :]
    y_test = y[test_index]
    y_pred = clf.predict(x_test)

    cv_y_test.append(y_test)
    cv_y_pred.append(y_pred)

tree_report = classification_report(np.concatenate(cv_y_test), np.concatenate(cv_y_pred))
```

Linear Discriminant Analysis

```
clf = LinearDiscriminantAnalysis()
kf = StratifiedKFold(n_splits=5, shuffle = True)

cv_y_test = []
cv_y_pred = []

for train_index, test_index in kf.split(x, y):

    x_train = x[train_index, :]
    y_train = y[train_index]

    clf.fit(x_train, y_train)

    x_test = x[test_index, :]
    y_test = y[test_index]
    y_pred = clf.predict(x_test)

    cv_y_test.append(y_test)
    cv_y_pred.append(y_pred)

lda_report = classification_report(np.concatenate(cv_y_test), np.concatenate(cv_y_pred))
```

RBF-SVM

```
clf = SVC(kernel = 'rbf')
kf = StratifiedKFold(n_splits=5, shuffle = True)

cv_y_test = []
cv_y_pred = []

for train_index, test_index in kf.split(x, y):

    x_train = x[train_index, :]
    y_train = y[train_index]

    clf.fit(x_train, y_train)

    x_test = x[test_index, :]
    y_test = y[test_index]
    y_pred = clf.predict(x_test)

    cv_y_test.append(y_test)
    cv_y_pred.append(y_pred)

rbf_report = classification_report(np.concatenate(cv_y_test), np.concatenate(cv_y_pred))
```

Ya que hicimos los 5 modelos, vamos a evaluar los resultados:

```
print('Linear SVM:\n', svm_report, '\n')
print('KNN:\n', knn_report, '\n')
print('Decision Tree:\n', tree_report, '\n')
print('Linear Discriminant Analysis:\n', lda_report, '\n')
print('RBF-SVM:\n', rbf_report, '\n')
```

```
accuracy                           0.72       629
   macro avg       0.73      0.72      0.72       629
weighted avg       0.73      0.72      0.72       629


Linear Discriminant Analysis:
              precision    recall  f1-score   support

         1.0       0.76      0.74      0.75        90
         2.0       0.55      0.51      0.53        90
         3.0       0.91      0.89      0.90        90
         4.0       0.87      0.81      0.84        90
         5.0       0.74      0.78      0.76        90
         6.0       0.48      0.57      0.52        90
         7.0       0.98      0.93      0.95        89

    accuracy                           0.75       629
   macro avg       0.76      0.75      0.75       629
weighted avg       0.76      0.75      0.75       629


RBF-SVM:
              precision    recall  f1-score   support

         1.0       0.95      0.91      0.93        90
         2.0       0.70      0.61      0.65        90
         3.0       1.00      0.94      0.97        90
         4.0       1.00      0.96      0.98        90
         5.0       0.91      0.97      0.94        90
         6.0       0.66      0.79      0.72        90
         7.0       0.98      0.99      0.98        89

    accuracy                           0.88       629
   macro avg       0.89      0.88      0.88       629
weighted avg       0.89      0.88      0.88       629
```

Entre estos 5 modelos, hay 3 que destacan, que son:

- Linear SVM
- KNN
- SVM-RBF

Estos 3 modelos tienen una muy buena precisión promedio, y un muy buen recall promedio. Pero yo me inclino más por el modelo Linear SVM porque, aunque es poca la diferencia, tiene mejor precisión y recall que los otros.

3. Escoge al menos dos clasificadores que hayas evaluado en el paso anterior e identifica sus hiperparámetros. Lleva a cabo el proceso de validación cruzada anidada para evaluar los dos modelos con la selección óptima de hiperparámetros.

Linear SVM:

```
cc = np.logspace(-3, 1, 100)

acc = []

for c in cc:
    print('---- C =', c)

    acc_cv = []

    kf = StratifiedKFold(n_splits=5, shuffle = True)

    for train_index, test_index in kf.split(x, y):

        # Training phase
        x_train = x[train_index, :]
        y_train = y[train_index]

        clf_cv = SVC(C=c, kernel = 'linear')

        clf_cv.fit(x_train, y_train)

        # Test phase
        x_test = x[test_index, :]
        y_test = y[test_index]
        y_pred = clf_cv.predict(x_test)

        acc_i = accuracy_score(y_test, y_pred)
        acc_cv.append(acc_i)

    acc_hyp = np.average(acc_cv)
    acc.append(acc_hyp)

    print('ACC:', acc_hyp)

opt_index = np.argmax(acc)
opt_hyperparameter = cc[opt_index]
print("\nOptimal C: ", opt_hyperparameter)

plt.plot(cc, acc)
plt.xscale('log')
plt.xlabel("c")
plt.ylabel("Accuracy")

plt.show()

# Fit model with optimal number of features
clf = SVC(C=opt_hyperparameter, kernel = 'linear')
clf.fit(x, y)
```

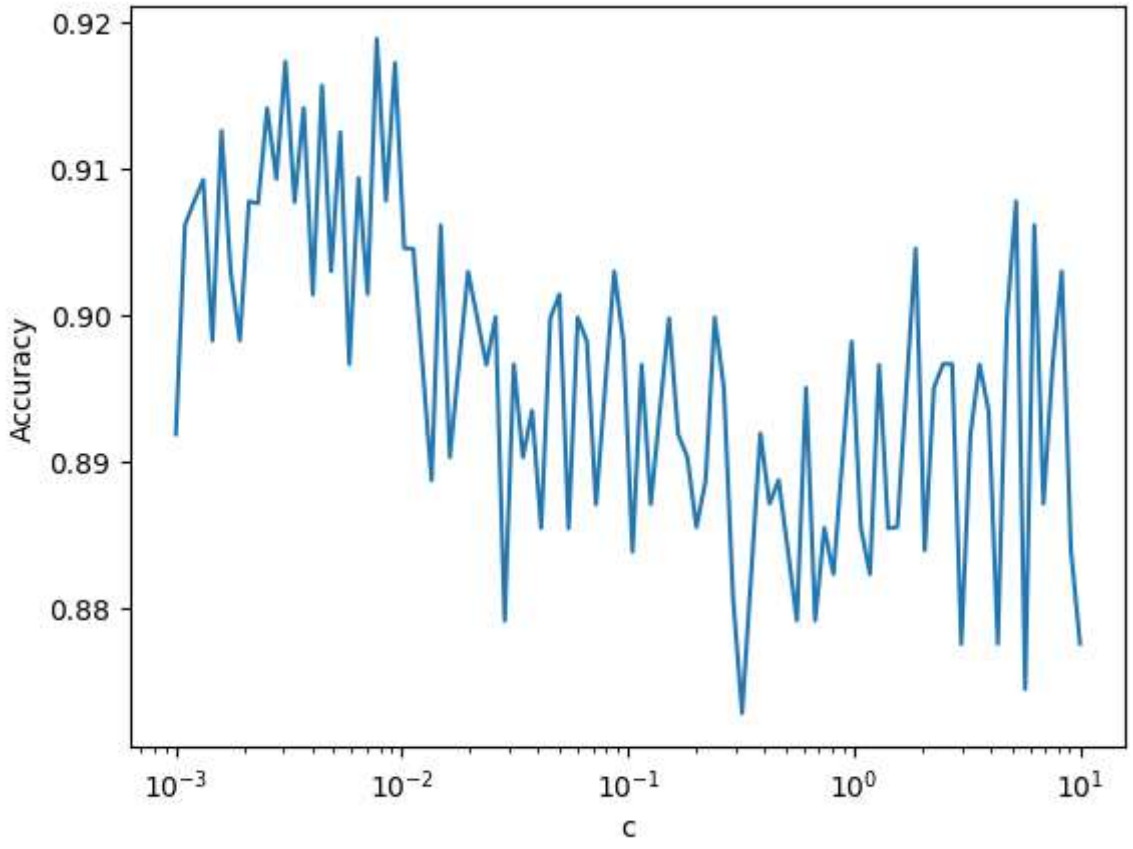
```
---- C = 0.001
ACC: 0.8918857142857144
---- C = 0.0010974987654930556
ACC: 0.9061968253968254
---- C = 0.0012045035402587824
ACC: 0.9077968253968255
---- C = 0.0013219411484660286
ACC: 0.9092825396825397
---- C = 0.0014508287784959402
ACC: 0.8982857142857142
---- C = 0.0015922827933410922
ACC: 0.9126222222222221
---- C = 0.0017474528400007683
ACC: 0.9030095238095237
---- C = 0.0019179102616724887
ACC: 0.8982984126984126
---- C = 0.00210490414451202
ACC: 0.9078095238095238
---- C = 0.0023101297000831605
ACC: 0.9076571428571428
---- C = 0.0025353644939701114
ACC: 0.9141714285714286
---- C = 0.0027825594022071257
ACC: 0.9093460317460318
---- C = 0.0030538555088334154
ACC: 0.9173587301587303
---- C = 0.003351602650938841
ACC: 0.9077587301587302
---- C = 0.0036783797718286343
ACC: 0.9141968253968255
---- C = 0.004037017258596553
ACC: 0.9014349206349207
---- C = 0.004430621457583882
ACC: 0.9157333333333334
---- C = 0.004862601580065354
ACC: 0.9030222222222223
---- C = 0.005336699231206312
ACC: 0.9125333333333334
---- C = 0.005857020818056668
ACC: 0.8966730158730158
---- C = 0.006428073117284319
ACC: 0.9093968253968254
---- C = 0.007054802310718645
ACC: 0.9014730158730158
---- C = 0.007742636826811269
ACC: 0.9189206349206349
---- C = 0.008497534359086447
ACC: 0.907847619047619
---- C = 0.009326033468832 2
ACC: 0.9172825396825397
---- C = 0.010235310218990263
ACC: 0.9046095238095238
---- C = 0.011233240329780276
ACC: 0.9045587301587302
---- C = 0.012328467394420665
ACC: 0.8966349206349206
---- C = 0.013530477745798075
ACC: 0.8887238095238095
---- C = 0.01484968262254465
ACC: 0.9061714285714286
```

```
---- C = 0.016297508346206444
ACC: 0.890311111111111
---- C = 0.01788649529057435
ACC: 0.8967238095238095
---- C = 0.019630406500402715
ACC: 0.9029968253968255
---- C = 0.021544346900318846
ACC: 0.8998222222222223
---- C = 0.023644894126454083
ACC: 0.8966222222222223
---- C = 0.025950242113997372
ACC: 0.8998857142857142
---- C = 0.02848035868435802
ACC: 0.8791238095238094
---- C = 0.03125715849688237
ACC: 0.8966603174603176
---- C = 0.03430469286314919
ACC: 0.890311111111111
---- C = 0.037649358067924694
ACC: 0.8934857142857142
---- C = 0.04132012400115339
ACC: 0.8854730158730157
---- C = 0.04534878508128584
ACC: 0.8998222222222223
---- C = 0.049770235643321115
ACC: 0.9014476190476192
---- C = 0.05462277217684343
ACC: 0.8854349206349207
---- C = 0.05994842503189412
ACC: 0.8998730158730158
---- C = 0.06579332246575682
ACC: 0.8982349206349205
---- C = 0.07220809018385467
ACC: 0.8870857142857143
---- C = 0.07924828983539177
ACC: 0.8950603174603173
---- C = 0.08697490026177834
ACC: 0.9030222222222223
---- C = 0.09545484566618342
ACC: 0.8982222222222221
---- C = 0.10476157527896651
ACC: 0.883847619047619
---- C = 0.11497569953977368
ACC: 0.8966349206349206
---- C = 0.1261856883066021
ACC: 0.8870857142857143
---- C = 0.1384886371393873
ACC: 0.8935238095238095
---- C = 0.15199110829529347
ACC: 0.8998095238095238
---- C = 0.1668100537200059
ACC: 0.8918730158730159
---- C = 0.18307382802953698
ACC: 0.8903238095238095
---- C = 0.2009233002565048
ACC: 0.8855365079365078
---- C = 0.22051307399030456
ACC: 0.8887111111111112
---- C = 0.24201282647943834
ACC: 0.8998730158730159
```

```
---- C = 0.26560877829466867
ACC: 0.8950857142857144
---- C = 0.29150530628251786
ACC: 0.8807873015873016
---- C = 0.31992671377973847
ACC: 0.8728126984126984
---- C = 0.35111917342151344
ACC: 0.8823619047619047
---- C = 0.3853528593710531
ACC: 0.8919365079365079
---- C = 0.4229242874389499
ACC: 0.8871365079365079
---- C = 0.4641588833612782
ACC: 0.8887492063492063
---- C = 0.5094138014816381
ACC: 0.8839873015873015
---- C = 0.5590810182512228
ACC: 0.8791492063492063
---- C = 0.6135907273413176
ACC: 0.8950603174603176
---- C = 0.6734150657750828
ACC: 0.8791492063492063
---- C = 0.7390722033525783
ACC: 0.885473015873016
---- C = 0.8111308307896873
ACC: 0.8823238095238095
---- C = 0.8902150854450392
ACC: 0.8902857142857142
---- C = 0.9770099572992257
ACC: 0.8982222222222223
---- C = 1.0722672220103242
ACC: 0.885511111111111
---- C = 1.1768119524349991
ACC: 0.882311111111111
---- C = 1.291549665014884
ACC: 0.8966349206349207
---- C = 1.4174741629268062
ACC: 0.8854476190476191
---- C = 1.5556761439304723
ACC: 0.8855492063492063
---- C = 1.7073526474706922
ACC: 0.8951365079365079
---- C = 1.873817422860385
ACC: 0.9045714285714286
---- C = 2.0565123083486534
ACC: 0.8839492063492063
---- C = 2.2570197196339215
ACC: 0.8950603174603173
---- C = 2.4770763559917115
ACC: 0.8966857142857142
---- C = 2.718588242732943
ACC: 0.8966857142857144
---- C = 2.9836472402833403
ACC: 0.8775365079365078
---- C = 3.2745491628777317
ACC: 0.8918730158730158
---- C = 3.5938136638046294
ACC: 0.8966603174603174
---- C = 3.94420605943766
ACC: 0.8934222222222223
---- C = 4.328761281083062
```

```
---- C = 4.328761281083062
ACC: 0.8775492063492063
---- C = 4.750810162102798
ACC: 0.8998984126984126
---- C = 5.21400828799969
ACC: 0.9078095238095238
---- C = 5.72236765935022
ACC: 0.8744507936507937
---- C = 6.280291441834259
ACC: 0.9061841269841271
---- C = 6.892612104349702
ACC: 0.8871365079365079
---- C = 7.56463327554629
ACC: 0.8966476190476191
---- C = 8.302175681319753
ACC: 0.9030222222222222
---- C = 9.111627561154895
ACC: 0.8839873015873018
---- C = 10.0
ACC: 0.8775619047619048

Optimal C:  0.007742636826811269
```



SVC

SVM-RBF

```
gg = np.logspace(-5, -1, 100)

acc = []

for g in gg:
    print('---- gamma =', g)
```

```
        acc_cv = []

        kf = StratifiedKFold(n_splits=5, shuffle = True)

        for train_index, test_index in kf.split(x, y):

            # Training phase
            x_train = x[train_index, :]
            y_train = y[train_index]

            clf_cv = SVC(kernel ='rbf', gamma = g)

            clf_cv.fit(x_train, y_train)

            # Test phase
            x_test = x[test_index, :]
            y_test = y[test_index]
            y_pred = clf_cv.predict(x_test)

            acc_i = accuracy_score(y_test, y_pred)
            acc_cv.append(acc_i)

        acc_hyp = np.average(acc_cv)
        acc.append(acc_hyp)

        print('ACC:', acc_hyp)

    opt_index = np.argmax(acc)
    opt_hyperparameter = gg[opt_index]
    print("\nOptimal gamma: ", opt_hyperparameter)

    plt.plot(gg, acc)
    plt.xscale('log')
    plt.xlabel("gamma")
    plt.ylabel("Accuracy")

    plt.show()

    # Fit model with optimal number of features
    clf = SVC(kernel ='rbf', gamma = opt_hyperparameter)
    clf.fit(x, y)
```

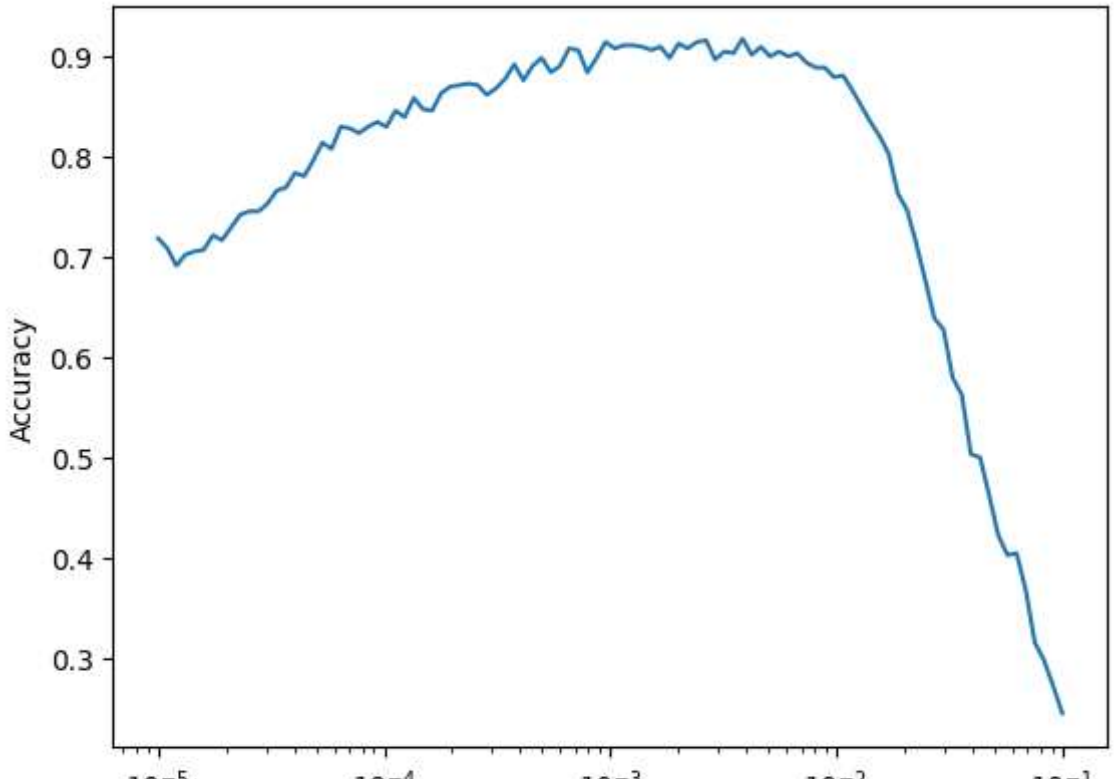
```
---- gamma = 1.3219411484660286e-05
ACC: 0.70264126984127
---- gamma = 1.4508287784959402e-05
ACC: 0.7057777777777778
---- gamma = 1.5922827933410938e-05
ACC: 0.7074412698412699
---- gamma = 1.747528400007683e-05
ACC: 0.7217523809523809
---- gamma = 1.917910261672489e-05
ACC: 0.7170539682539683
---- gamma = 2.104904144512022e-05
ACC: 0.7297015873015873
---- gamma = 2.310129700083158e-05
ACC: 0.7424126984126984
---- gamma = 2.5353644939701114e-05
ACC: 0.7456888888888888
---- gamma = 2.782559402207126e-05
ACC: 0.7456761904761905
---- gamma = 3.053855508833412e-05
ACC: 0.753511111111111
---- gamma = 3.351602650938841e-05
ACC: 0.7662857142857142
---- gamma = 3.678379771828634e-05
ACC: 0.7694730158730158
---- gamma = 4.037017258596558e-05
ACC: 0.7838730158730158
---- gamma = 4.430621457583878e-05
ACC: 0.7806984126984127
---- gamma = 4.8626015800653536e-05
ACC: 0.7965587301587302
---- gamma = 5.336699231206312e-05
ACC: 0.8140063492063492
---- gamma = 5.857020818056673e-05
ACC: 0.8077333333333334
---- gamma = 6.428073117284319e-05
ACC: 0.8299936507936507
---- gamma = 7.054802310718646e-05
ACC: 0.8282920634920634
---- gamma = 7.742636826811278e-05
ACC: 0.8234666666666668
---- gamma = 8.497534359086438e-05
ACC: 0.8298539682539683
---- gamma = 9.326033468832199e-05
ACC: 0.834679365079365
---- gamma = 0.00010235310218990269
ACC: 0.8298412698412697
---- gamma = 0.00011233240329780277
ACC: 0.8457396825396826
---- gamma = 0.00012328467394420660
ACC: 0.8394539682539683
---- gamma = 0.00013530477745798074
ACC: 0.8584507936507937
---- gamma = 0.00014849682622544650
ACC: 0.8473904761904762
---- gamma = 0.00016297508346206434
ACC: 0.8458285714285715
---- gamma = 0.00017886495290574350
ACC: 0.8632761904761905
---- gamma = 0.00019630406500402724
ACC: 0.8697015873015873
---- gamma = 0.00021544346900318845
```

```
---- gamma = 0.0002154434690031884
ACC: 0.8712634920634921
---- gamma = 0.00023644894126454073
ACC: 0.8727873015873016
---- gamma = 0.00025950242113997375
ACC: 0.8712380952380953
---- gamma = 0.0002848035868435802
ACC: 0.861663492063492
---- gamma = 0.00031257158496882353
ACC: 0.8680253968253968
---- gamma = 0.0003430469286314919
ACC: 0.8775492063492063
---- gamma = 0.00037649358067924713
ACC: 0.8918984126984129
---- gamma = 0.00041320124001153384
ACC: 0.876
---- gamma = 0.00045348785081285824
ACC: 0.8903238095238095
---- gamma = 0.0004977023564332114
ACC: 0.8982603174603174
---- gamma = 0.0005462277217684342
ACC: 0.8840380952380953
---- gamma = 0.0005994842503189409
ACC: 0.8903238095238095
---- gamma = 0.0006579332246575682
ACC: 0.907784126984127
---- gamma = 0.0007220809018385471
ACC: 0.9061587301587302
---- gamma = 0.0007924828983539178
ACC: 0.8838603174603176
---- gamma = 0.0008697490026177834
ACC: 0.8982603174603175
---- gamma = 0.0009545484566618347
ACC: 0.9141587301587302
---- gamma = 0.001047615752789665
ACC: 0.9077968253968255
---- gamma = 0.0011497569953977367
ACC: 0.9109333333333334
---- gamma = 0.0012618568830660211
ACC: 0.9109587301587302
---- gamma = 0.0013848863713938732
ACC: 0.9093714285714286
---- gamma = 0.0015199110829529348
ACC: 0.9061841269841269
---- gamma = 0.0016681005372000592
ACC: 0.9094222222222224
---- gamma = 0.0018307382802953698
ACC: 0.8982349206349207
---- gamma = 0.002009233002565048
ACC: 0.9125587301587302
---- gamma = 0.0022051307399030455
ACC: 0.9077968253968255
---- gamma = 0.0024201282647943836
ACC: 0.9141460317460318
---- gamma = 0.0026560877829466868
ACC: 0.9157587301587302
---- gamma = 0.0029150530628251786
ACC: 0.8966476190476189
---- gamma = 0.0031992671377973846
ACC: 0.9046984126984127
---- gamma = 0.0035111917342151347
```

```
ACC: 0.903047619047619
---- gamma = 0.0038535285937105314
ACC: 0.917320634920635
---- gamma = 0.0042292428743894986
ACC: 0.9014349206349207
---- gamma = 0.004641588833612782
ACC: 0.9093587301587303
---- gamma = 0.0050941380148163806
ACC: 0.8998476190476191
---- gamma = 0.005590810182512228
ACC: 0.9046603174603174
---- gamma = 0.006135907273413176
ACC: 0.8998095238095238
---- gamma = 0.006734150657750828
ACC: 0.9029460317460318
---- gamma = 0.007390722033525783
ACC: 0.8934984126984127
---- gamma = 0.008111308307896872
ACC: 0.8887619047619048
---- gamma = 0.008902150854450393
ACC: 0.888711111111111
---- gamma = 0.009770099572992257
ACC: 0.8791873015873015
---- gamma = 0.010722672220103242
ACC: 0.8807619047619047
---- gamma = 0.011768119524349991
ACC: 0.8664507936507937
---- gamma = 0.01291549665014884
ACC: 0.850463492063492
---- gamma = 0.014174741629268062
ACC: 0.83464126984127
---- gamma = 0.015556761439304723
ACC: 0.8203809523809523
---- gamma = 0.01707352647470692
ACC: 0.8028444444444445
---- gamma = 0.018738174228603847
ACC: 0.7631619047619047
---- gamma = 0.020565123083486535
ACC: 0.7472634920634921
---- gamma = 0.022570197196339216
ACC: 0.7138539682539683
---- gamma = 0.024770763559917114
ACC: 0.6772063492063493
---- gamma = 0.02718588242732943
ACC: 0.6390984126984127
---- gamma = 0.029836472402833405
ACC: 0.6279873015873016
---- gamma = 0.032745491628777317
ACC: 0.5803174603174603
---- gamma = 0.03593813663804629
ACC: 0.5629460317460317
---- gamma = 0.0394420605943766
ACC: 0.5040126984126984
---- gamma = 0.043287612810830614
ACC: 0.5008507936507937
---- gamma = 0.04750810162102798
ACC: 0.46270476190476184
---- gamma = 0.0521400828799969
ACC: 0.42302222222222224
---- gamma = 0.0572236765935022
```

```
ACC: 0.403695238095238
---- gamma = 0.0628029144183426
ACC: 0.405536507936508
---- gamma = 0.06892612104349702
ACC: 0.36883809523809524
---- gamma = 0.07564633275546291
ACC: 0.3163555555555556
---- gamma = 0.08302175681319753
ACC: 0.29894603174603174
---- gamma = 0.09111627561154896
ACC: 0.2734984126984127
---- gamma = 0.1
ACC: 0.2464253968253968

Optimal gamma:  0.0038535285937105314
```



4. Prepara tus modelos para producción haciendo lo siguiente:

- Opten los hiperparámetros óptimos utilizando todo el conjunto de datos con validación cruzada.
- Con los hiperparámetros óptimos, ajusta el modelo con todos los datos.

Linear SVM

```
cc = np.logspace(-3, 1, 100)

acc = []

for c in cc:
    print('---- C =', c)

    clf_cv = SVC(C=c, kernel = 'linear')
```

```
        clf_cv.fit(x, y)

        y_pred = clf_cv.predict(x)

        acc_i = accuracy_score(y, y_pred)
        acc.append(acc_i)
        print('ACC:', acc_i)

    opt_index = np.argmax(acc)
    opt_hyperparameter = cc[opt_index]
    print("\nOptimal C: ", opt_hyperparameter)

    plt.plot(cc, acc)
    plt.xscale('log')
    plt.xlabel("c")
    plt.ylabel("Accuracy")

    plt.show()

    # Fit model with optimal number of features
    clf = SVC(C=opt_hyperparameter, kernel = 'linear')
    clf.fit(x, y)
```

```
---- C = 0.001
ACC: 0.9427662957074722
---- C = 0.0010974987654930556
ACC: 0.9475357710651828
---- C = 0.0012045035402587824
ACC: 0.9491255961844197
---- C = 0.0013219411484660286
ACC: 0.9491255961844197
---- C = 0.0014508287784959402
ACC: 0.9523052464228935
---- C = 0.0015922827933410922
ACC: 0.9523052464228935
---- C = 0.0017477528400007683
ACC: 0.9538950715421304
---- C = 0.0019179102616724887
ACC: 0.9554848966613673
---- C = 0.00210490414451202
ACC: 0.9570747217806042
---- C = 0.0023101297000831605
ACC: 0.9570747217806042
---- C = 0.0025353644939701114
ACC: 0.9570747217806042
---- C = 0.0027825594022071257
ACC: 0.9570747217806042
---- C = 0.0030538555088334154
ACC: 0.9570747217806042
---- C = 0.003351602650938841
ACC: 0.958664546899841
---- C = 0.0036783797718286343
ACC: 0.9618441971383148
---- C = 0.004037017258596553
ACC: 0.9634340222575517
---- C = 0.004430621457583882
ACC: 0.9650238473767886
---- C = 0.004862601580065354
ACC: 0.9666136724960255
---- C = 0.005336699231206312
ACC: 0.9666136724960255
---- C = 0.005857020818056668
ACC: 0.9682034976152624
---- C = 0.006428073117284319
ACC: 0.9697933227344993
---- C = 0.007054802310718645
ACC: 0.9713831478537361
---- C = 0.007742636826811269
ACC: 0.9713831478537361
---- C = 0.008497534359086447
ACC: 0.9713831478537361
---- C = 0.009326033468832 2
ACC: 0.9745627980922098
---- C = 0.010235310218990263
ACC: 0.9793322734499205
---- C = 0.011233240329780276
ACC: 0.9793322734499205
---- C = 0.012328467394420665
ACC: 0.9793322734499205
---- C = 0.013530477745798075
ACC: 0.9793322734499205
---- C = 0.01484968262254465
ACC: 0.9809220985691574
```

```
---- C = 0.016297508346206444
ACC: 0.9793322734499205
---- C = 0.01788649529057435
ACC: 0.9793322734499205
---- C = 0.019630406500402715
ACC: 0.9793322734499205
---- C = 0.021544346900318846
ACC: 0.9825119236883942
---- C = 0.023644894126454083
ACC: 0.9825119236883942
---- C = 0.025950242113997372
ACC: 0.9825119236883942
---- C = 0.02848035868435802
ACC: 0.9841017488076311
---- C = 0.03125715849688237
ACC: 0.9872813990461049
---- C = 0.03430469286314919
ACC: 0.9872813990461049
---- C = 0.037649358067924694
ACC: 0.9872813990461049
---- C = 0.04132012400115339
ACC: 0.9888712241653418
---- C = 0.04534878508128584
ACC: 0.9904610492845787
---- C = 0.049770235643321115
ACC: 0.9920508744038156
---- C = 0.05462277217684343
ACC: 0.9920508744038156
---- C = 0.05994842503189412
ACC: 0.9936406995230525
---- C = 0.06579332246575682
ACC: 0.9936406995230525
---- C = 0.07220809018385467
ACC: 0.9936406995230525
---- C = 0.07924828983539177
ACC: 0.9952305246422893
---- C = 0.08697490026177834
ACC: 0.9952305246422893
---- C = 0.09545484566618342
ACC: 1.0
---- C = 0.10476157527896651
ACC: 1.0
---- C = 0.11497569953977368
ACC: 1.0
---- C = 0.1261856883066021
ACC: 1.0
---- C = 0.1384886371393873
ACC: 1.0
---- C = 0.15199110829529347
ACC: 1.0
---- C = 0.1668100537200059
ACC: 1.0
---- C = 0.18307382802953698
ACC: 1.0
---- C = 0.2009233002565048
ACC: 1.0
---- C = 0.22051307399030456
ACC: 1.0
---- C = 0.24201282647943834
ACC: 1.0
```

```
---- C = 0.26560877829466867
ACC: 1.0
---- C = 0.29150530628251786
ACC: 1.0
---- C = 0.31992671377973847
ACC: 1.0
---- C = 0.35111917342151344
ACC: 1.0
---- C = 0.3853528593710531
ACC: 1.0
---- C = 0.4229242874389499
ACC: 1.0
---- C = 0.4641588833612782
ACC: 1.0
---- C = 0.5094138014816381
ACC: 1.0
---- C = 0.5590810182512228
ACC: 1.0
---- C = 0.6135907273413176
ACC: 1.0
---- C = 0.6734150657750828
ACC: 1.0
---- C = 0.7390722033525783
ACC: 1.0
---- C = 0.8111308307896873
ACC: 1.0
---- C = 0.8902150854450392
ACC: 1.0
---- C = 0.9770099572992257
ACC: 1.0
---- C = 1.0722672220103242
ACC: 1.0
---- C = 1.1768119524349991
ACC: 1.0
---- C = 1.291549665014884
ACC: 1.0
---- C = 1.4174741629268062
ACC: 1.0
---- C = 1.5556761439304723
ACC: 1.0
---- C = 1.7073526474706922
ACC: 1.0
---- C = 1.873817422860385
ACC: 1.0
---- C = 2.0565123083486534
ACC: 1.0
---- C = 2.2570197196339215
ACC: 1.0
---- C = 2.4770763559917115
ACC: 1.0
---- C = 2.718588242732943
ACC: 1.0
---- C = 2.9836472402833403
ACC: 1.0
---- C = 3.2745491628777317
ACC: 1.0
---- C = 3.5938136638046294
ACC: 1.0
---- C = 3.94420605943766
ACC: 1.0
```

```
---- C = 4.328761281083062
ACC: 1.0
---- C = 4.750810162102798
ACC: 1.0
---- C = 5.21400828799969
ACC: 1.0
---- C = 5.72236765935022
ACC: 1.0
---- C = 6.280291441834259
ACC: 1.0
---- C = 6.892612104349702
ACC: 1.0
---- C = 7.56463327554629
ACC: 1.0
---- C = 8.302175681319753
ACC: 1.0
```

SVM-RBF

```
ACC: 1.0
```

```
gg = np.logspace(-3, 1, 100)

acc = []

for g in gg:
    print('---- gamma =', g)

    clf_cv = SVC(kernel ='rbf', gamma = g)

    clf_cv.fit(x, y)

    y_pred = clf_cv.predict(x)

    acc_i = accuracy_score(y, y_pred)
    acc.append(acc_i)
    print('ACC:', acc_i)

opt_index = np.argmax(acc)
opt_hyperparameter = gg[opt_index]
print("\nOptimal Gamma: ", opt_hyperparameter)

plt.plot(gg, acc)
plt.xscale('log')
plt.xlabel("gamma")
plt.ylabel("Accuracy")

plt.show()

# Fit model with optimal number of features
clf = SVC(kernel ='rbf', gamma = opt_hyperparameter)
clf.fit(x, y)
```

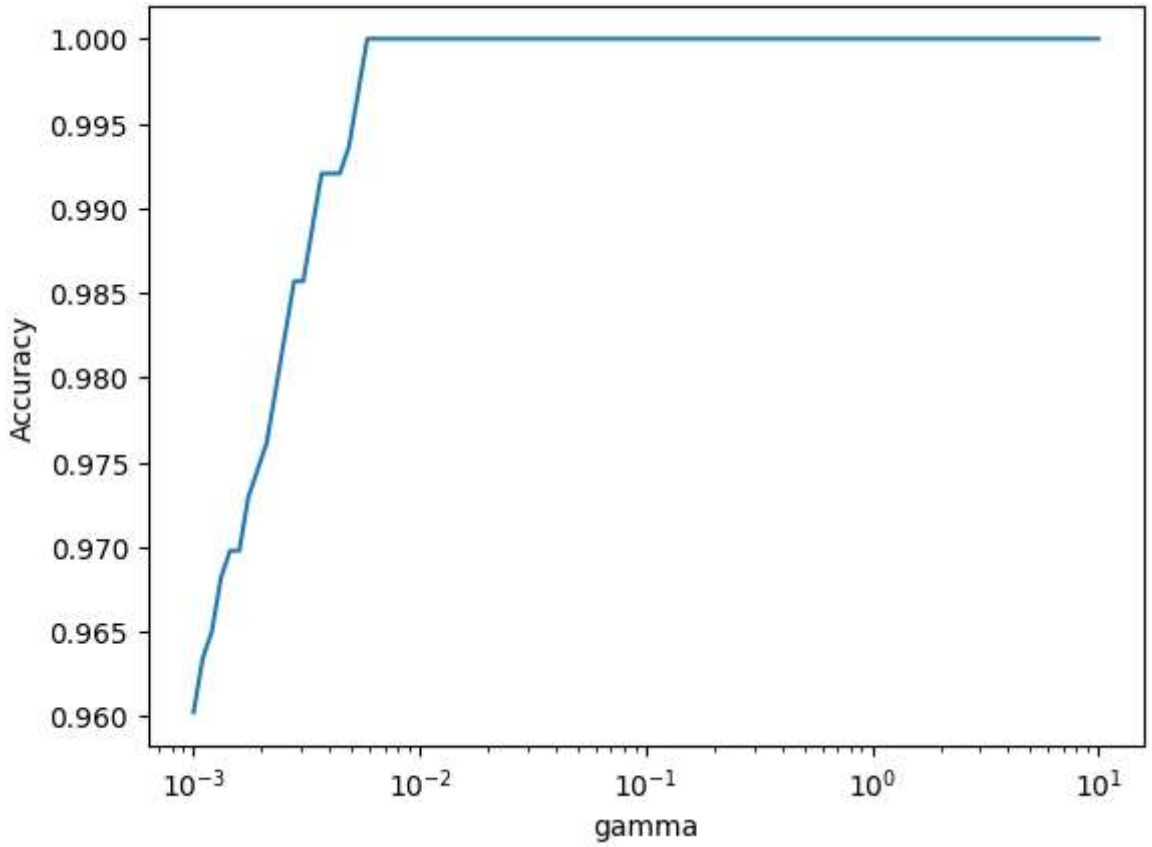
```
ACC: 0.9682034976152624
---- gamma = 0.0014508287784959402
ACC: 0.9697933227344993
---- gamma = 0.0015922827933410922
ACC: 0.9697933227344993
---- gamma = 0.001747528400007683
ACC: 0.972972972972973
---- gamma = 0.0019179102616724887
ACC: 0.9745627980922098
---- gamma = 0.00210490414451202
ACC: 0.9761526232114467
---- gamma = 0.0023101297000831605
ACC: 0.9793322734499205
---- gamma = 0.0025353644939701114
ACC: 0.9825119236883942
---- gamma = 0.0027825594022071257
ACC: 0.985691573926868
---- gamma = 0.0030538555088334154
ACC: 0.985691573926868
---- gamma = 0.003351602650938841
ACC: 0.9888712241653418
---- gamma = 0.0036783797718286343
ACC: 0.9920508744038156
---- gamma = 0.004037017258596553
ACC: 0.9920508744038156
---- gamma = 0.004430621457583882
ACC: 0.9920508744038156
---- gamma = 0.004862601580065354
ACC: 0.9936406995230525
---- gamma = 0.005336699231206312
ACC: 0.9968203497615262
---- gamma = 0.005857020818056668
ACC: 1.0
---- gamma = 0.006428073117284319
ACC: 1.0
---- gamma = 0.007054802310718645
ACC: 1.0
---- gamma = 0.007742636826811269
ACC: 1.0
---- gamma = 0.008497534359086447
ACC: 1.0
---- gamma = 0.0093260334688322
ACC: 1.0
---- gamma = 0.010235310218990263
ACC: 1.0
---- gamma = 0.011233240329780276
ACC: 1.0
---- gamma = 0.012328467394420665
ACC: 1.0
---- gamma = 0.013530477745798075
ACC: 1.0
---- gamma = 0.01484968262254465
ACC: 1.0
---- gamma = 0.016297508346206444
ACC: 1.0
---- gamma = 0.01788649529057435
ACC: 1.0
---- gamma = 0.019630406500402715
ACC: 1.0
---- gamma = 0.021544346900318846
```

```
ACC: 1.0
---- gamma = 0.023644894126454083
ACC: 1.0
---- gamma = 0.025950242113997372
ACC: 1.0
---- gamma = 0.02848035868435802
ACC: 1.0
---- gamma = 0.03125715849688237
ACC: 1.0
---- gamma = 0.03430469286314919
ACC: 1.0
---- gamma = 0.037649358067924694
ACC: 1.0
---- gamma = 0.04132012400115339
ACC: 1.0
---- gamma = 0.04534878508128584
ACC: 1.0
---- gamma = 0.049770235643321115
ACC: 1.0
---- gamma = 0.05462277217684343
ACC: 1.0
---- gamma = 0.05994842503189412
ACC: 1.0
---- gamma = 0.06579332246575682
ACC: 1.0
---- gamma = 0.07220809018385467
ACC: 1.0
---- gamma = 0.07924828983539177
ACC: 1.0
---- gamma = 0.08697490026177834
ACC: 1.0
---- gamma = 0.09545484566618342
ACC: 1.0
---- gamma = 0.10476157527896651
ACC: 1.0
---- gamma = 0.11497569953977368
ACC: 1.0
---- gamma = 0.1261856883066021
ACC: 1.0
---- gamma = 0.1384886371393873
ACC: 1.0
---- gamma = 0.15199110829529347
ACC: 1.0
---- gamma = 0.1668100537200059
ACC: 1.0
---- gamma = 0.18307382802953698
ACC: 1.0
---- gamma = 0.2009233002565048
ACC: 1.0
---- gamma = 0.22051307399030456
ACC: 1.0
---- gamma = 0.24201282647943834
ACC: 1.0
---- gamma = 0.26560877829466867
ACC: 1.0
---- gamma = 0.29150530628251786
ACC: 1.0
---- gamma = 0.31992671377973847
ACC: 1.0
---- gamma = 0.35111917342151344
```

```
ACC: 1.0
---- gamma = 0.3853528593710531
ACC: 1.0
---- gamma = 0.4229242874389499
ACC: 1.0
---- gamma = 0.4641588833612782
ACC: 1.0
---- gamma = 0.5094138014816381
ACC: 1.0
---- gamma = 0.5590810182512228
ACC: 1.0
---- gamma = 0.6135907273413176
ACC: 1.0
---- gamma = 0.6734150657750828
ACC: 1.0
---- gamma = 0.7390722033525783
ACC: 1.0
---- gamma = 0.8111308307896873
ACC: 1.0
---- gamma = 0.8902150854450392
ACC: 1.0
---- gamma = 0.9770099572992257
ACC: 1.0
---- gamma = 1.0722672220103242
ACC: 1.0
---- gamma = 1.1768119524349991
ACC: 1.0
---- gamma = 1.291549665014884
ACC: 1.0
---- gamma = 1.4174741629268062
ACC: 1.0
---- gamma = 1.5556761439304723
ACC: 1.0
---- gamma = 1.7073526474706922
ACC: 1.0
---- gamma = 1.873817422860385
ACC: 1.0
---- gamma = 2.0565123083486534
ACC: 1.0
---- gamma = 2.2570197196339215
ACC: 1.0
---- gamma = 2.4770763559917115
ACC: 1.0
---- gamma = 2.718588242732943
ACC: 1.0
---- gamma = 2.9836472402833403
ACC: 1.0
---- gamma = 3.2745491628777317
ACC: 1.0
---- gamma = 3.5938136638046294
ACC: 1.0
---- gamma = 3.94420605943766
ACC: 1.0
---- gamma = 4.328761281083062
ACC: 1.0
---- gamma = 4.750810162102798
ACC: 1.0
---- gamma = 5.21400828799969
ACC: 1.0
---- gamma = 5.72236765935022
ACC: 1.0
```

```
---- gamma = 6.280291441834259
ACC: 1.0
---- gamma = 6.892612104349702
ACC: 1.0
---- gamma = 7.56463327554629
ACC: 1.0
---- gamma = 8.302175681319753
ACC: 1.0
---- gamma = 9.111627561154895
ACC: 1.0
---- gamma = 10.0
ACC: 1.0

Optimal Gamma:  0.005857020818056668
```



SVC

```
SVC(gamma=0.005857020818056668)
```

Hay que aclarar que estas graficas no son de evaluación del modelo, son solo una demostración de los resultados del modelo, ya que estas predicciones se estan haciendo con los mismos datos con los que se entrenó el modelo, o sea, todos los datos.

5. Contesta lo siguientes:

- ¿Observas un problema en cuanto al balanceo de las clases? ¿Por qué?

  No particularmente, todas las clases mantienen un buen balance. Todas las muestras de cada clase mantienen practicamente el mismo numero de muestras. Solo en el recall no

están completamente iguales los recall de cada clase, pero no se alejan mucho entre sí, solo una o dos clases.

- ¿Qué modelo o modelos fueron efectivos para clasificar tus datos? ¿Observas algo especial sobre los modelos? Argumenta tu respuesta.

  Hubo 3 modelos muy buenos, ordenados del mejor al peor:

  - Linear SVM
  - SVM-RBF
  - KNN

  Como expliqué anteriormente, estos 3 modelos mantienen una muy buena precisión promedio y un muy buen recall promedio tambien.

  Una observación interesante es que los dos mejores modelos son clasificadores basados en Maquinas Soporte Vectorial (SVM).

- ¿Observas alguna mejora importante al optimizar hiperparámetros? ¿Es el resultado que esperabas? Argumenta tu respuesta.

  Pues la accuracy de los modelos aumentó, no mucho, pero hizo más exacto al modelo. Los 3 mejores modelos aumentaron su accuracy como entre un 0.05 y un 0.08.

- ¿Qué inconvenientes hay al encontrar hiperparámetros? ¿Por qué?

  Encontrar los hiperparametros perfectos puede llevar mucho procesamiento al momento de probar tantas opciones, y esto es un gran costo computacional. Tambien podría generarse sobreajuste al hondar mucho en los parametros porque podría ser que se encuentre el hiperparametro perfecto, pero para este conjunto de datos, exclusivamente.