Memoria de sesiones de Prácticas de Laboratorio (1-4)

Juan Francisco Mier Montoto
Inteligencia de Negocio, EPI Gijón 23-24

Índice

Práctica 1

Durante esta práctica, tan solo se realizan pequeñas modificaciones al código base que se tiene.

Sesión 1

Durante la primera sesión, se juega con tareas de la librería `scipy`: distribuciones de probabilidad, test estadísticos, distancias entre instancias...

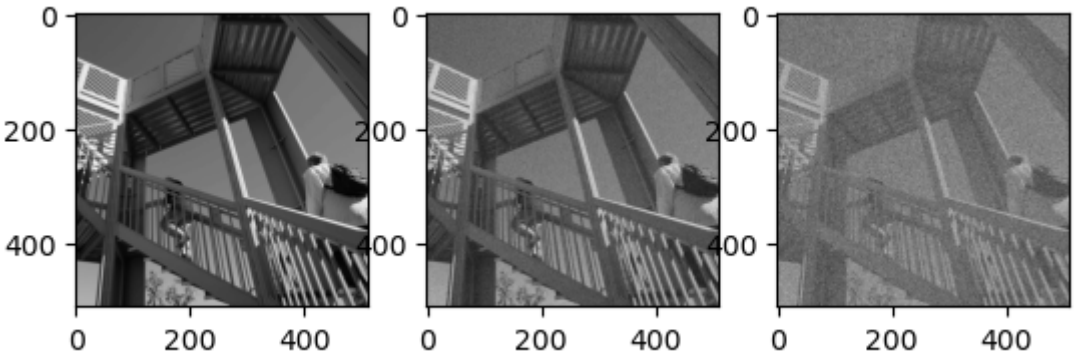
```
# Imports generales
import scipy
import matplotlib.pyplot as plt
import numpy as np
```

Ejercicio 1

Código

```
from scipy.stats import norm

def signaltonoise(a, axis=0, ddof=0):
    a = np.asanyarray(a)
    m = a.mean(axis)
    sd = a.std(axis=axis, ddof=ddof)
    return np.where(sd == 0, 0, m / sd)


face = scipy.datasets.ascent().astype(float)  # INSTALAR POOCH!!! → se utiliza datasets para eliminar warning de ejecución
faceruido16 = face + norm.rvs(loc=0, scale=16, size=face.shape)
faceruido64 = face + norm.rvs(loc=0, scale=64, size=face.shape)
print(signaltonoise(face, axis=None))

plt.gray()  # se muestran las imágenes en escala de grises

# foto original, sin ruido
plt.subplot(131, aspect='equal')
plt.imshow(face)

# foto con ruido (varianza 16)
plt.subplot(132, aspect='equal')
plt.imshow(faceruido16)

# foto con ruido (varianza 64)
plt.subplot(133, aspect='equal')
plt.imshow(faceruido64)

plt.show()
```

Resultado



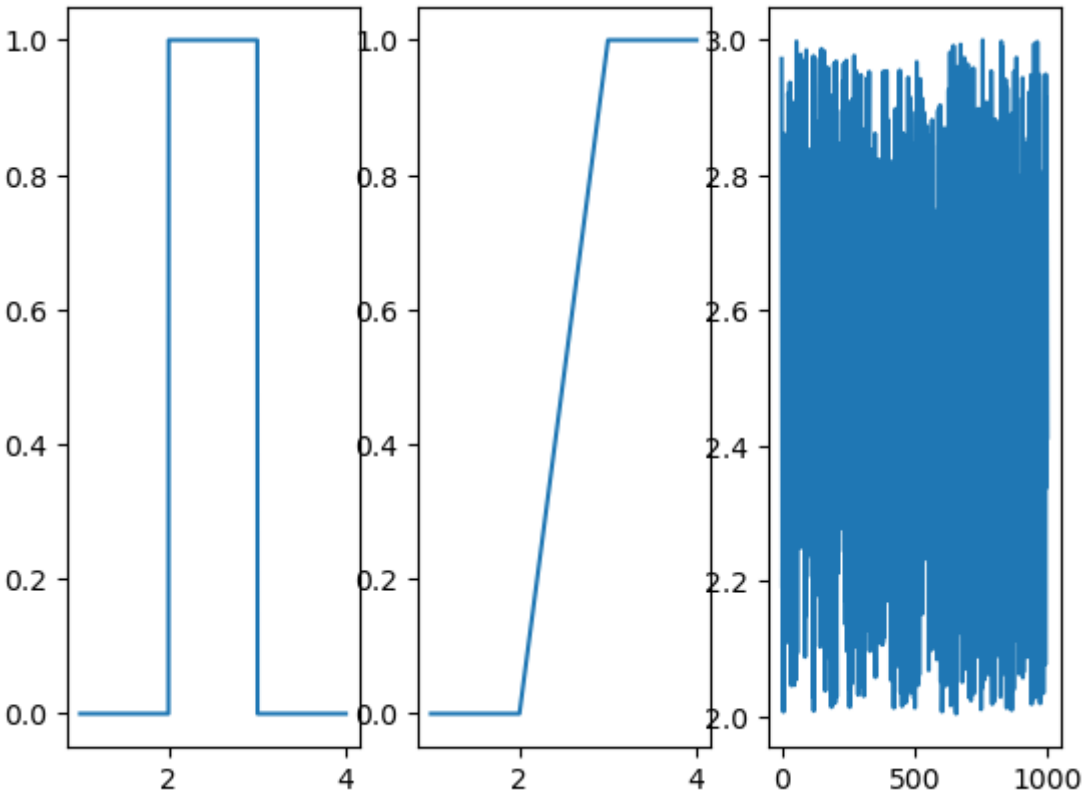
Ejercicio 2

Código

```
from scipy.stats import uniform

x = np.linspace(1, 4, 1000)

# Función de densidad
plt.subplot(131)
plt.plot(x, uniform.pdf(x, 2, 1))

# Función de distribución
plt.subplot(132)
plt.plot(x, uniform.cdf(x, 2, 1))

# Generador aleatorio
plt.subplot(133)
plt.plot(uniform.rvs(2, 1, size=1000))

plt.show()
```

Copy

Resultado

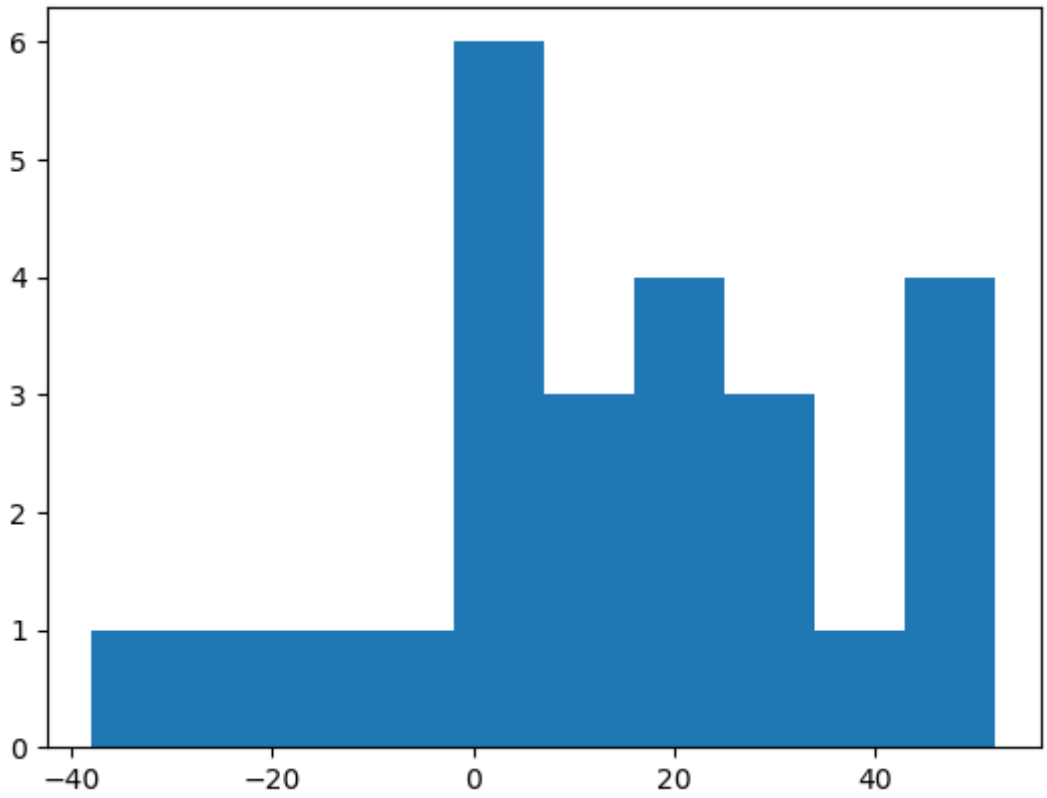


Ejercicio 3

Código

```
from scipy.stats import wilcoxon                                                    Copy

data = np.array([[113, 105, 130, 101, 138, 118, 87, 116, 75, 96,
                  122, 103, 116, 107, 118, 103, 111, 104, 111, 89, 78, 100,
89, 85, 88],
                  [137, 105, 133, 108, 115, 170, 103, 145, 78, 107,
                  84, 148, 147, 87, 166, 146, 123, 135, 112, 93, 76, 116, 78,
101, 123]])
dataDiff = data[1, :] - data[0, :]
print(dataDiff.mean(), dataDiff.std())
# Histograma
plt.hist(dataDiff)
plt.show()
# Test t
wilcoxon_stat, p_value = wilcoxon(dataDiff)
print("El p-valor es: %02f" % p_value)
if p_value < 0.05:
    print("Los tiempos son significativamente diferentes")
else:
    print("No hay evidencia para rechazar que sean iguales: los dos actuadores
son indistintos")
```

Resultado




```
El p-valor es: 0.007531
Los tiempos son significativamente diferentes
```

Copy

Ejercicio 4

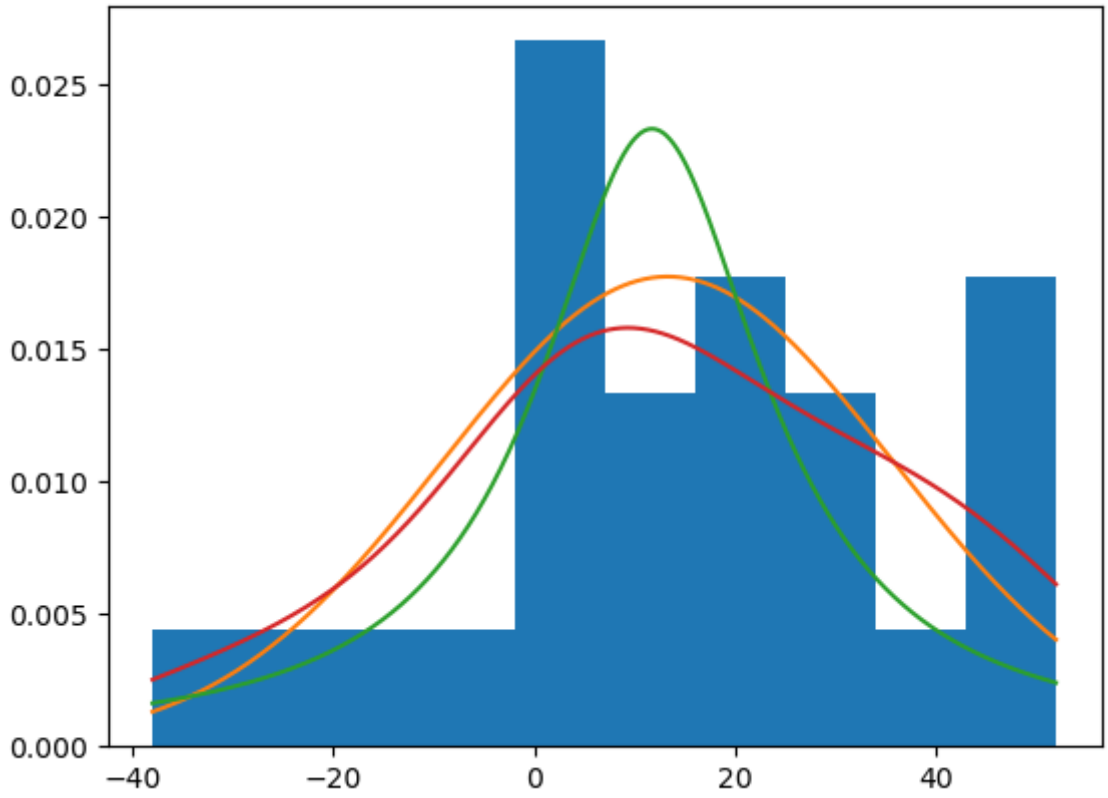
Código

```
import scipy.stats as ss

data = np.array([[113, 105, 130, 101, 138, 118, 87, 116, 75, 96,
                  122, 103, 116, 107, 118, 103, 111, 104, 111, 89, 78, 100,
89, 85, 88],
                  [137, 105, 133, 108, 115, 170, 103, 145, 78, 107,
                  84, 148, 147, 87, 166, 146, 123, 135, 112, 93, 76, 116, 78,
101, 123]])
dataDiff = data[1, :] - data[0, :]

# Ajuste de una normal a los datos
mean, std = ss.norm.fit(dataDiff)
cauchy_mean, cauchy_std = ss.cauchy.fit(dataDiff)

plt.hist(dataDiff, density=1)
x = np.linspace(dataDiff.min(), dataDiff.max(), 1000)

pdf = ss.norm.pdf(x, mean, std)
kde = ss.gaussian_kde(dataDiff)
cauchy = ss.cauchy.pdf(x, cauchy_mean, cauchy_std)
```

Copy

```
plt.plot(x, pdf)
plt.plot(x, cauchy)
plt.plot(x, kde(x))
plt.show()
```

Resultado



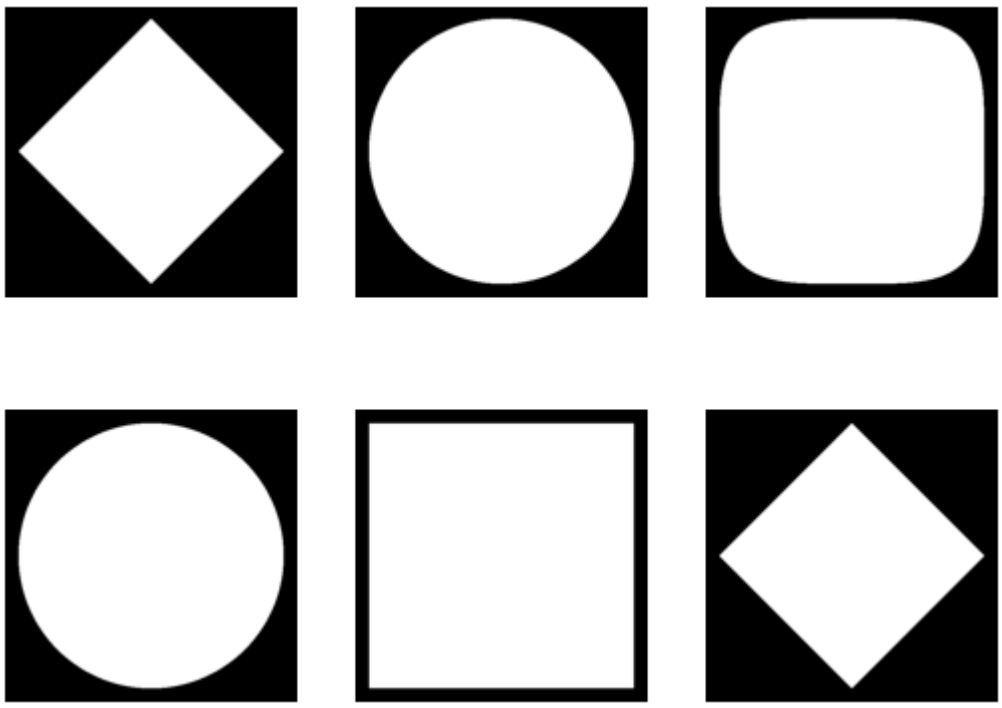
Ejercicio 5

Código

```
from scipy.spatial.distance import minkowski, euclidean, chebyshev, cityblock

Square = np.meshgrid(np.linspace(-1.1, 1.1, 512), np.linspace(-1.1, 1.1, 512),
indexing='ij')
X = Square[0]
Y = Square[1]
f = lambda x, y, p: minkowski([x, y], [0.0, 0.0], p) <= 1.0
Ball = lambda p: np.vectorize(f)(X, Y, p)

plt.subplot(231, aspect='equal')
plt.imshow(Ball(1))
plt.axis('off')

plt.subplot(232, aspect='equal')
plt.imshow(Ball(2))
plt.axis('off')

plt.subplot(233, aspect='equal')
```

```
plt.imshow(Ball(4))
plt.axis('off')

# distancia euclídea
f = lambda x, y: euclidean([x, y], [0.0, 0.0]) <= 1.0
Ball = lambda: np.vectorize(f)(X, Y)

plt.subplot(234, aspect='equal')
plt.imshow(Ball())
plt.axis('off')

f = lambda x, y: chebyshev([x, y], [0.0, 0.0]) <= 1.0
Ball = lambda: np.vectorize(f)(X, Y)
plt.subplot(235, aspect='equal')
plt.imshow(Ball())
plt.axis('off')

f = lambda x, y: cityblock([x, y], [0.0, 0.0]) <= 1.0
Ball = lambda: np.vectorize(f)(X, Y)
plt.subplot(236, aspect='equal')
plt.imshow(Ball())
plt.axis('off')

plt.show()
```

Resultado



Sesión 2

Durante la segunda sesión, se hace uso de la librería `sklearn` principalmente.

Ejercicio 6

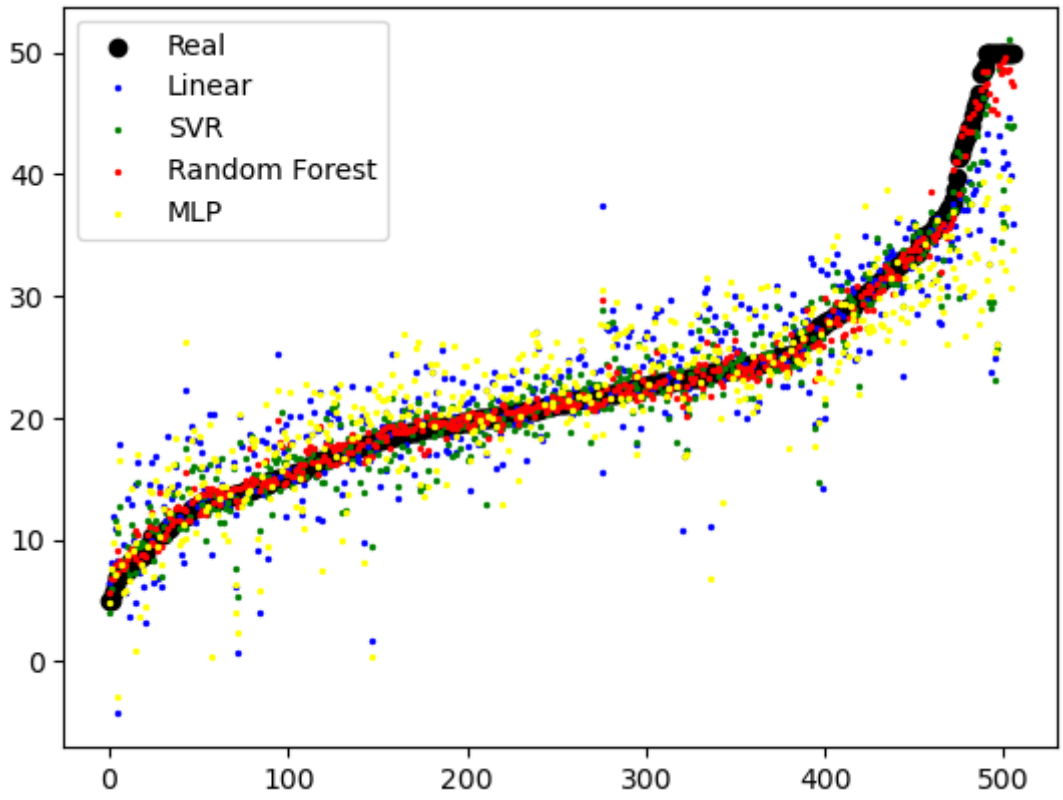
Código

Como base para el ejercicio 6, se utiliza el archivo `sklearn-1-multivariate.py`.

Copy

```
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.feature_selection import SelectKBest, f_regression
from sklearn.linear_model import LinearRegression
from sklearn.svm import SVR
from sklearn.ensemble import RandomForestRegressor
from sklearn.neural_network import MLPRegressor
import numpy as np
import pandas as pd

data_url = "http://lib.stat.cmu.edu/datasets/boston"
raw_df = pd.read_csv(data_url, sep="\s+", skiprows=22, header=None)
X_full = np.hstack([raw_df.values[::2, :], raw_df.values[1::2, :2]])
Y = raw_df.values[1::2, 2]
print(raw_df.columns)

print(X_full.shape)
print(Y.shape)
X = X_full[:, :]
orden = np.argsort(Y)
horizontal = np.arange(Y.shape[0])
plt.scatter(horizontal, Y[orden], color='black')

regressor = LinearRegression()
regressor.fit(X, Y)
plt.scatter(horizontal, regressor.predict(X)[orden], 2, color='blue')

regressor = SVR(kernel='rbf', C=1e5, epsilon=1)
regressor.fit(X, Y)
plt.scatter(horizontal, regressor.predict(X)[orden], 2, color='green')

regressor = RandomForestRegressor()
regressor.fit(X, Y)
plt.scatter(horizontal, regressor.predict(X)[orden], 2, color='red')

regressor = MLPRegressor()
regressor.fit(X, Y)
plt.scatter(horizontal, regressor.predict(X)[orden], 2, color='yellow')

plt.legend(['Real', 'Linear', 'SVR', 'Random Forest', 'MLP'])
plt.show()
```

Resultado y análisis



- A simple vista, se puede observar que el modelo `RandomForestRegressor` es el más preciso.
- Para `SVR`, los diferentes tipos de kernel no afectan prácticamente nada a la predicción final.
- Con valores superiores de k, el resto de modelos se ajustan considerablemente mejor a la "realidad".

Ejercicio 7

Código

Copy

```
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.feature_selection import SelectKBest, f_regression
from sklearn.linear_model import LinearRegression
from sklearn.svm import SVR
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import cross_val_score, cross_val_predict
from sklearn.metrics import mean_squared_error
import numpy as np
import pandas as pd

data_url = "http://lib.stat.cmu.edu/datasets/boston"
raw_df = pd.read_csv(data_url, sep="\s+", skiprows=22, header=None)
X_full = np.hstack([raw_df.values[::2, :], raw_df.values[1::2, :2]])
Y = raw_df.values[1::2, 2]
print(raw_df.columns)

print(X_full.shape)
print(Y.shape)
# Se elige la variable mas dependiente de la salida
```

```
selector = SelectKBest(f_regression, k=1)
selector.fit(X_full, Y)
X = X_full[:, :]

regressor = LinearRegression()
regressor.fit(X, Y)
# Error cuadratico medio de cada fold, seguido de media de folds
score = cross_val_score(regressor, X, Y, scoring='neg_mean_squared_error',
cv=10).mean()
# La prediccion es la respuesta del modelo aprendido en el
# el fold para el que la instancia fue parte del conjunto de test
predicted = cross_val_predict(regressor, X, Y, cv=10)
mse = mean_squared_error(Y, predicted)
# Resultados parecidos pero no iguales
print("LIN MSE =", mse)
print("LIN score =", -score)

regressor = SVR(kernel='rbf', C=1e1, epsilon=1)
regressor.fit(X, Y)
score = cross_val_score(regressor, X, Y, scoring='neg_mean_squared_error',
cv=10).mean()
predicted = cross_val_predict(regressor, X, Y, cv=10)
mse = mean_squared_error(Y, predicted)
print("SVR MSE =", mse)
print("SVR score =", -score)

regressor = RandomForestRegressor()
regressor.fit(X, Y)
score = cross_val_score(regressor, X, Y, scoring='neg_mean_squared_error',
cv=10).mean()
predicted = cross_val_predict(regressor, X, Y, cv=10)
mse = mean_squared_error(Y, predicted)
print("RNF MSE =", mse)
print("RNF score =", -score)

# Resulta mejor usar SVR, luego LIN y por último RNF
# Esto es porque el error cuadratico medio es menor

# Si se utilizan todas las variables, el resultado es el inverso.
```

Resultado y análisis

```
LIN MSE = 34.5396595399932
LIN score = 34.705255944524865
SVR MSE = 65.9780875187515
SVR score = 65.85931872419 85
RNF MSE = 21.881361942687747
RNF score = 21.706483113568613
```

En cuanto a error se refiere, el mejor es, de nuevo, `RandomForestGenerator`, y el peor es el regresor lineal. El resultado es similar se empleen todas las variables o no.

Sesión 3

Durante la tercera sesión, se trabaja con `pandas` (y un poco la "visión general").
Los cinco archivos aportados son ejercicios ya completados, que se leen y se ejecutan para comprender su funcionamiento.

Ejercicio 8

```
import pandas as pd
import numpy as np
from sklearn.model_selection import cross_val_score, cross_val_predict
from sklearn.linear_model import LinearRegression
from sklearn.svm import SVR
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error

df = pd.read_excel("Churn_Modelling_NANs.xlsx", na_values='NA')
```

```
# 1. Elimina las filas con valores perdidos
df = df.dropna()

# 2. Selecciona un dataframe 'X' con las columnas
'CreditScore','Age','Tenure','Balance','NumOfProducts','HasCrCard','IsActiveMem
ber'
X = df[['CreditScore', 'Age', 'Tenure', 'Balance', 'NumOfProducts',
'HasCrCard', 'IsActiveMember']]

# 3. Selecciona un dataframe 'y' con la columna 'EstimatedSalary'
y = df[['EstimatedSalary']]
```

```
# 4. Haz tres modelos (LinearRegression, SVR, RandomForestRegressor) de Y
frente a X
#        y compara el error cuadrático medio de los tres con validación cruzada
(10 fold)
cv = 10

# LinearRegression
lr = LinearRegression()
lr.fit(X, y)
lr_scores = cross_val_score(lr, X, y, cv=cv,
scoring="neg_mean_squared_error").mean()
lr_predicted = cross_val_predict(lr, X, y, cv=cv)
print("LinearRegression mse mean: " + str(mean_squared_error(y, lr_predicted)))

# SVR
svr = SVR(kernel="linear")
```

```
svr.fit(X, y)
svr_scores = cross_val_score(svr, X, y, cv=cv,
scoring="neg_mean_squared_error")
svr_predicted = cross_val_predict(svr, X, y, cv=cv)
print("SVR mse mean: " + str(mean_squared_error(y, svr_predicted)))

# RandomForestRegressor
rfr = RandomForestRegressor(n_estimators=10)
rfr.fit(X, y)
rfr_scores = cross_val_score(rfr, X, y, cv=cv,
scoring="neg_mean_squared_error")
rfr_predicted = cross_val_predict(rfr, X, y, cv=cv)
print("RandomForestRegressor mse mean: " + str(mean_squared_error(y,
rfr_predicted)))
```

```
# 5. Selecciona un dataframe 'XC' con las columnas CreditScore, Age, Tenure,Copy
Balance,
#        NumOfProducts, HasCrCard, IsActiveMember, EstimatedSalary y un
dataframe 'C' con
#        la columna Exited. Haz tres clasificaciones diferentes de C frente a
XC y compara
#        sus porcentajes de aciertos con validación cruzada (10 fold)
XC = df[['CreditScore', 'Age', 'Tenure', 'Balance', 'NumOfProducts',
'HasCrCard', 'IsActiveMember', 'EstimatedSalary']]
C = df[['Exited']]

# LinearRegression
lr = LinearRegression()
lr_scores = cross_val_score(lr, XC, C, cv=cv, scoring="neg_mean_squared_error")
print("LinearRegression mse mean: " + str(lr_scores.mean()))

# SVR
svr = SVR(kernel="linear")
svr_scores = cross_val_score(svr, XC, C, cv=cv,
scoring="neg_mean_squared_error")
print("SVR mse mean: " + str(svr_scores.mean()))

# RandomForestRegressor
rfr = RandomForestRegressor(n_estimators=10)
rfr_scores = cross_val_score(rfr, XC, C, cv=cv,
scoring="neg_mean_squared_error")
print("RandomForestRegressor mse mean: " + str(rfr_scores.mean()))
```

Práctica 2

*Nota inicial: `sklearn` se debe instalar mediante el comando `pip install scikit-learn`.

Módulo adicional

IMPORTANTE: para la realización de esta práctica se ha generado un nuevo módulo de Python que evite tener que ejecutar los mismos comandos una y otra vez, automatizando un poco el transcurso de la práctica.

```
def slice(dataset) -> [pd.DataFrame, pd.DataFrame]:
        """
        :param dataset: dataset
        :return: dataset without the target column and the target column
        """
        return dataset.iloc[:, 0:-1], dataset.iloc[:, -1]


def mdav(dataset) -> float:
        """
        :param dataset: dataset
        :return: the variance of the most dispersed attribute
        """
        return np.var(dataset, axis=0).max()


def reduce_by_var(dataset, base) -> pd.DataFrame:
        """
        :param dataset: dataset
        :param base: base mutiplier of mdav
        :return: the dataset without attributes higher than `base *
mdav(dataset)`
        """
        return VarianceThreshold(base * mdav(dataset)).fit_transform(dataset)


def reduce_with_univariate(dataset_x, dataset_y, k) -> pd.DataFrame:
        """
        :param dataset: dataset
        :param k: number of attributes to keep
        :return: the dataset with the `k` most relevant attributes
        """
        return SelectKBest(chi2, k=k).fit_transform(dataset_x, dataset_y)


def reduce_with_rfe(dataset_x, dataset_y):
        """
        :param dataset: dataset
        :return: the dataset without attributes that are not relevant
        """
```

```
        return RFECV(SVC(kernel="linear"), step=1,
cv=5).fit_transform(dataset_x, dataset_y)

def importances(dataset_x, dataset_y, name):
        """
        :param dataset: dataset
        """

        forest = ExtraTreesClassifier(n_estimators=250, random_state=0)

        forest.fit(dataset_x, dataset_y)
        importances = forest.feature_importances_
        std = np.std([tree.feature_importances_ for tree in
forest.estimators_], axis=0)
        indices = np.argsort(importances)[::-1]

        plt.figure()
        plt.title(f"{name} Feature importances")
        plt.bar(range(dataset_x.shape[1]), importances[indices], color="r",
yerr=std[indices], align="center")
        plt.xticks(range(dataset_x.shape[1]), indices)
        plt.xlim([-1, dataset_x.shape[1]])
        plt.show()
```

Este módulo se utiliza en la siguiente práctica para automatizar también algunas acciones.

Práctica

El objetivo de la práctica es la selección de las características más relevantes mediante el uso de diferentes técnicas:

- Eliminación de variables con poca varianza
- Eliminación de variables basada en estadísticos univariantes
- Eliminación recursiva de variables
- Eliminación de variables usando `SelectFromModel`

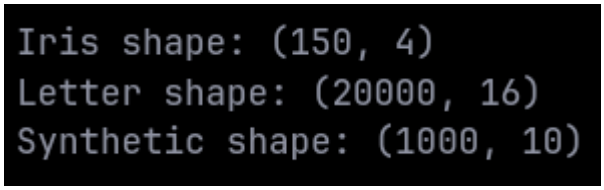
0. Lectura de archivos

Para esta práctica, se utilizan tres conjuntos de datos diferentes que se han de cargar de manera distinta. Teniendo en cuenta mi estructura de archivos local, donde los archivos `csv` están en una subcarpeta `datasets` con respecto a la carpeta de la práctica, los conjuntos se cargan de la siguiente manera:

```
import datasets.datasets_synthetic
import pandas as pd
import utils

# Carga inicial de datasets
iris = pd.read_csv('datasets/datasets-uci-iris.csv', sep=',', decimal='.',
header=None, names=['sepal_length', 'sepal_width', 'petal_length',
```

```
'petal_width', 'target'])
letter = pd.read_csv('datasets/datasets-uci-letter.csv', sep=',', decimal='.')
synthetic = datasets.datasets_synthetic.gen()

iris_x, iris_y = utils.slice(iris)
letter_x, letter_y = utils.slice(letter)
synthetic_x = synthetic[0]
synthetic_y = synthetic[1]

print(f"Iris shape: {iris_x.shape}")
print(f"Letter shape: {letter_x.shape}")
print(f"Synthetic shape: {synthetic_x.shape}")
```

Como se puede comprobar, se dividen los conjuntos para extraer la clasificación a una variable distinta, por necesidades de algunos métodos de eliminación. Además, la célula imprime la "forma" de los conjuntos importados como referencia para las reducciones posteriores:



Para la obtención del conjunto "sintético", se crea un pequeño módulo dentro de la anteriormente mencionada carpeta `datasets`:

```
from sklearn.datasets import make_classification
import pandas as pd


def gen():
        return make_classification(n_samples=1000, n_features=10,
n_informative=3, n_redundant=0, n_repeated=0, n_classes=2, random_state=0,
shuffle=False)
```

1. Eliminación de variables con poca varianza

```
base = 0.1

iris_var = utils.reduce_by_var(iris_x, base)
letter_var = utils.reduce_by_var(letter_x, base)
synthetic_var = utils.reduce_by_var(synthetic_x, base)

print(f"Iris reduced_by_var shape: {iris_var.shape}")
print(f"Letter reduced_by_var shape: {letter_var.shape}")
print(f"Synthetic reduced_by_var shape: {syntehtic_var.shape}")
```

El resultado de la célula anterior es una reducción de una sola variable en el conjunto de IRIS:

```
Iris reduced_by_var shape: (150, 3)
Letter reduced_by_var shape: (20000, 16)
Synthetic reduced_by_var shape: (1000, 10)
```

2. Eliminación de variables basada en estadísticos univariantes

Se hace uso del estadístico `SelectKBest`, como se indica en el enunciado de la práctica, con $k = 2$.
Debido a un problema desconocido, no se puede aplicar esta técnica al dataset sintético, por lo que se dejan las correspondientes instrucciones comentadas.

```
k = 2

iris_uni = utils.reduce_with_univariate(iris_x, iris_y, k)
letter_uni = utils.reduce_with_univariate(letter_x, letter_y, k)
# synthetic_uni = utils.reduce_with_univariate(synthetic_x, synthetic_y, k)

print(f"Iris reduced_with_univariate shape: {iris_uni.shape}")
print(f"Letter reduced_with_univariate shape: {letter_uni.shape}")
# print(f"Synthetic reduced_with_univariate shape: {synthetic_uni.shape}")
```

El resultado de la ejecución es lo siguiente:

```
Iris reduced_with_univariate shape: (150, 2)
Letter reduced_with_univariate shape: (20000, 2)
```

3. Eliminación recursiva de variables

Siguiendo las indicaciones y explicaciones del enunciado de la práctica, se utiliza el siguiente estimador y selector:

```
estimator = SVC(kernel="linear")
selector = RFECV(estimator, step=1, cv=5)
```

El código de la práctica es el siguiente:

```
iris_rfe = utils.reduce_with_rfe(iris_x, iris_y)
# letter_rfe = utils.reduce_with_rfe(letter_x, letter_y)
synthetic_rfe = utils.reduce_with_rfe(synthetic_x, synthetic_y)

print(f"Iris reduced_with_rfe shape: {iris_rfe.shape}")
# print(f"Letter reduced_with_rfe shape: {letter_rfe.shape}")
print(f"Synthetic reduced_with_rfe shape: {synthetic_rfe.shape}")
```

En este caso, es el dataset "letter" el que no se puede hacer funcionar, por lo que se comentan las instrucciones pertinentes.
El resultado de la ejecución es el siguiente:

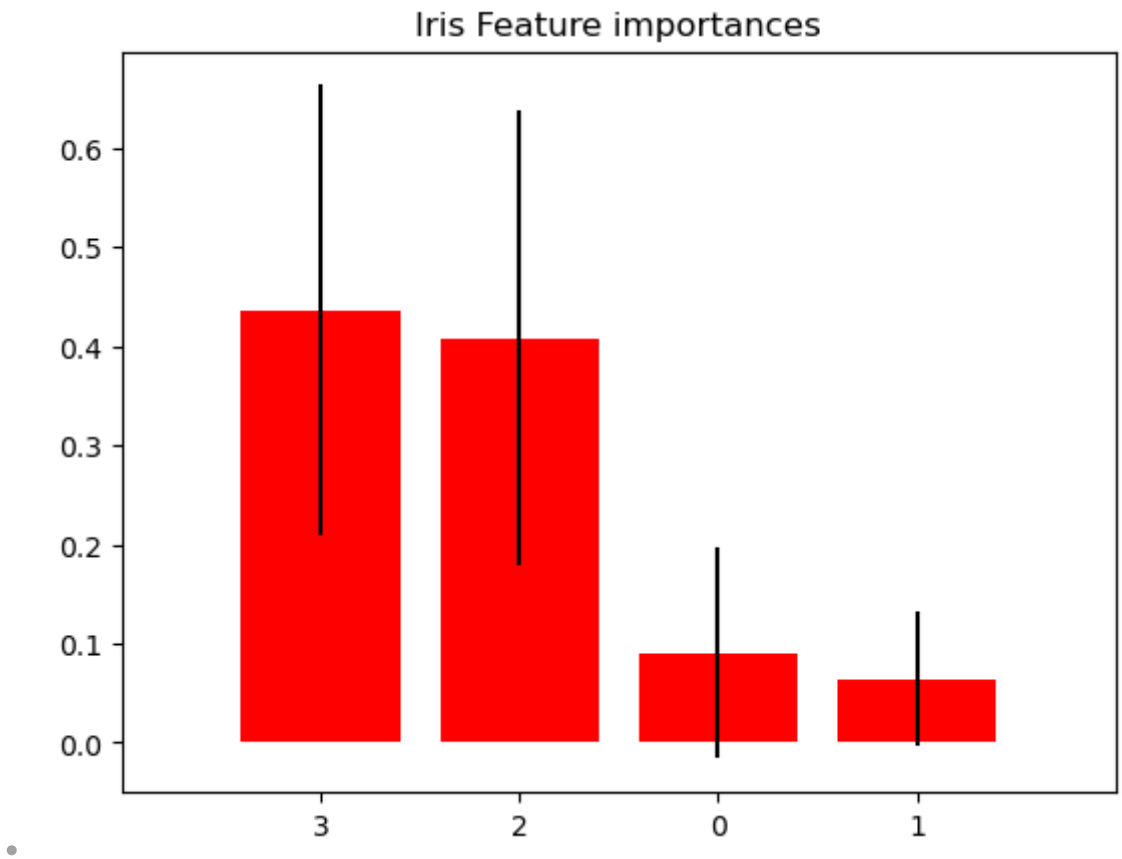
```
Iris reduced_with_rfe shape: (150, 4)
Synthetic reduced_with_rfe shape: (1000, 1)
```

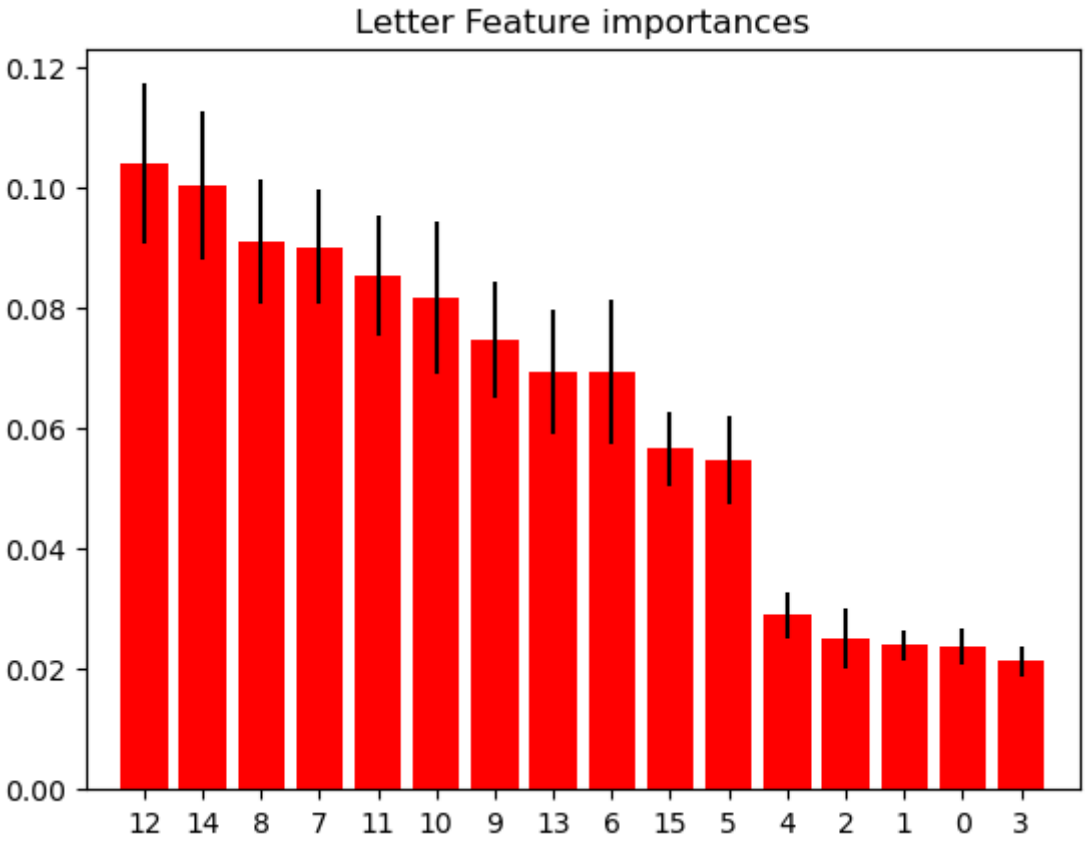
4. Eliminación de variables usando `SelectFromModel`

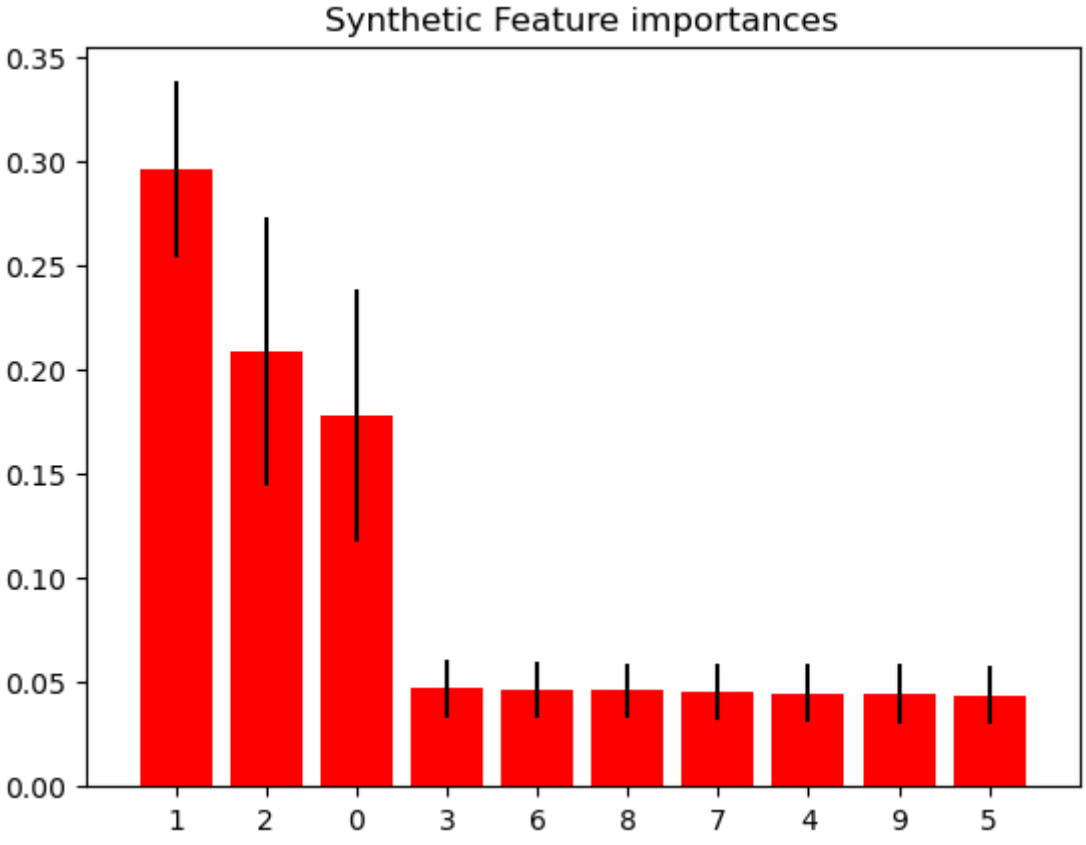
```
utils.importances(iris_x, iris_y, "Iris")
utils.importances(letter_x, letter_y, "Letter")
utils.importances(synthetic_x, synthetic_y, "Synthetic")
```

Copy

El código anterior devuelve los siguientes gráficos:







Según estos resultados, los atributos más relevantes de cada dataset son, respectivamente:

- `petal_width` y `petal_length` para Iris.
- `x-ege` y `y-ege` para Letter.
- Para el conjunto sintético, el segundo y tercer atributo.

Práctica 3

Nota: para esta práctica se utiliza el módulo de la práctica anterior

Como menciona el enunciado, hay una serie de objetivos en esta práctica:

- Limpiar el dataset, quitando los valores perdidos
- Escalar o normalizar las variables
- Detectar las variables irrelevantes o redundantes
- Construir un modelo lineal y otro con random forest
- Realizar la validación cruzada de ambos modelos y decidir cuál es la precisión del modelo conseguido

El dataset que se utiliza tiene los siguientes atributos:

- N : número de habitantes
- R : radio de la población
- S : función desconocida del número de habitantes
- T : función desconocida del radio de la población
- U : datos aleatorios con el mismo rango que la solución
- G : datos aleatorios con la misma media y desviación típica que la solución
- L : longitud de cable en la población

Se utiliza una celda de Jupyter por cada objetivo de la práctica. El resultado en código es el siguiente:

```
# Leer los datos del dataset
import pandas as pd
cables_filename = 'cables.csv'
cables = pd.read_csv(cables_filename, sep=',', decimal='.')
cables = pd.DataFrame(cables).iloc[:, 1:] # Eliminar la primera columna
```

Copy

```
# Limpiar el dataset, quitando los valores perdidos
print(f"Before cleaning: {cables.shape}")
cables = cables.dropna(axis=0, how='any')
print(f"After cleaning: {cables.shape}")
```

Copy

```
# Escalar o normalizar las variables
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()
cables = scaler.fit_transform(cables)
cables = pd.DataFrame(cables)
```

Copy

```
# Detectar las variables irrelevantes o redundantes
import utils

cables_x, cables_y = utils.slice(cables)
print(f"Shape before reducing: {cables_x.shape}")
cables_x = utils.reduce_by_var(cables_x, 0.1)
print(f"Shape after reducing: {cables_x.shape}")
```

Copy

```
# Construir un modelo lineal y otro con RandomForest
from sklearn.linear_model import LinearRegression
from sklearn.ensemble import RandomForestRegressor

linear = LinearRegression()
forest = RandomForestRegressor()

linear.fit(cables_x, cables_y)
forest.fit(cables_x, cables_y)
```

Copy

```
# Realizar la validación cruzada de ambos modelos y
# decidir cuál es la precisión del modelo conseguido.
from sklearn.model_selection import cross_val_score
import numpy as np

scores_linear = cross_val_score(linear, cables_x, cables_y, cv=10)
scores_forest = cross_val_score(forest, cables_x, cables_y, cv=10)

print('Linear score: ', np.mean(scores_linear))
print('Forest score: ', np.mean(scores_forest))

# Calcular el MSE de las predicciones de ambos modelos
from sklearn.metrics import mean_squared_error

linear_pred = linear.predict(cables_x)
forest_pred = forest.predict(cables_x)

print('Linear MSE: ', mean_squared_error(cables_y, linear_pred))
print('Forest MSE: ', mean_squared_error(cables_y, forest_pred))
```

Copy

De la ejecución de la última celda se obtienen los siguientes valores:

```
Linear score:  0.6946090180893877
Forest score:  0.623781387753034
Linear MSE:  0.006244325252000521
Forest MSE:  0.0011039784436457961
```

RandomForest tiene un score inferior pero también un error medio inferior, por lo que ambos modelos son similares pero mediocres, ya que ninguno obtiene buenos resultados.

Práctica 4

Nota inicial: `umap` se debe instalar mediante el comando `pip install ulmap-learn`.

Para la resolución de esta práctica, se utiliza el Jupyter Notebook mencionado en el enunciado de la misma `manifold-learning.ipynb` como punto de partida para ejecutar los diferentes métodos.

Después de copiar la sección de librerías y métodos del notebook anterior, se cargan los datos aportados mediante los siguientes comandos:

```
import pandas as pd

data = pd.read_excel('4000datosSimuladosEnergia.xlsx')
points = data.iloc[:, 0:-1].values
colors = data.iloc[:, -1].values
```

Para que `pandas` sea capaz de importar archivos con formato Excel, se necesita la librería `openpyxl` (`pip install openpyxl`).

Se utilizan los siguientes parámetros durante toda la práctica:

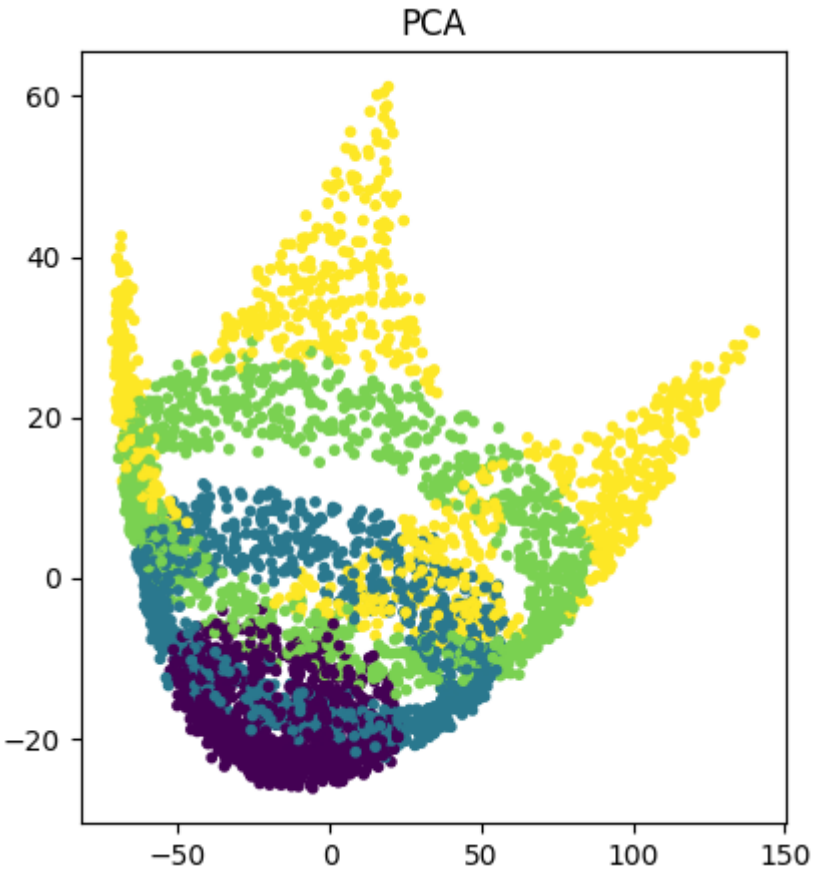
```
n_neighbors = 12
n_components = 2
params = {
        "n_neighbors": n_neighbors,
        "n_components": n_components,
        "eigen_solver": "auto",
        "random_state": 0
}
```

El análisis de los resultados obtenidos se realiza al final de este apartado.

Parte obligatoria

A continuación, se recopilan todas las gráficas obtenidas de cada módulo y el código utilizado para ejecutar cada una.

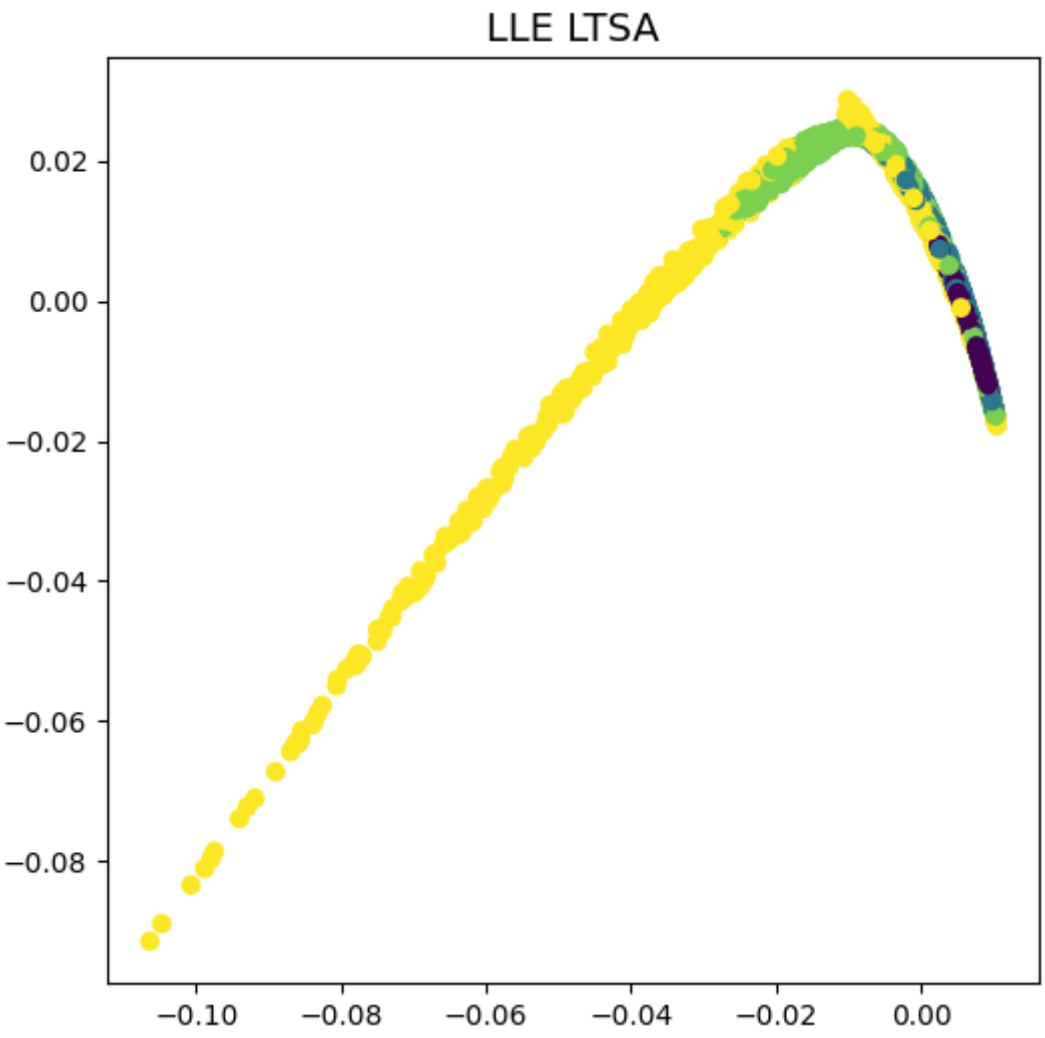
PCA




```
# PCA
fig = plt.figure(figsize=(10, 5))

pca = PCA(n_components=2)
pca.fit(data)
data_pca = pca.transform(data)

ax = fig.add_subplot(1, 2, 1)
ax.scatter(data_pca[:, 0], data_pca[:, 1], c=colors, s=10)
ax.set_title('PCA')
```

Copy

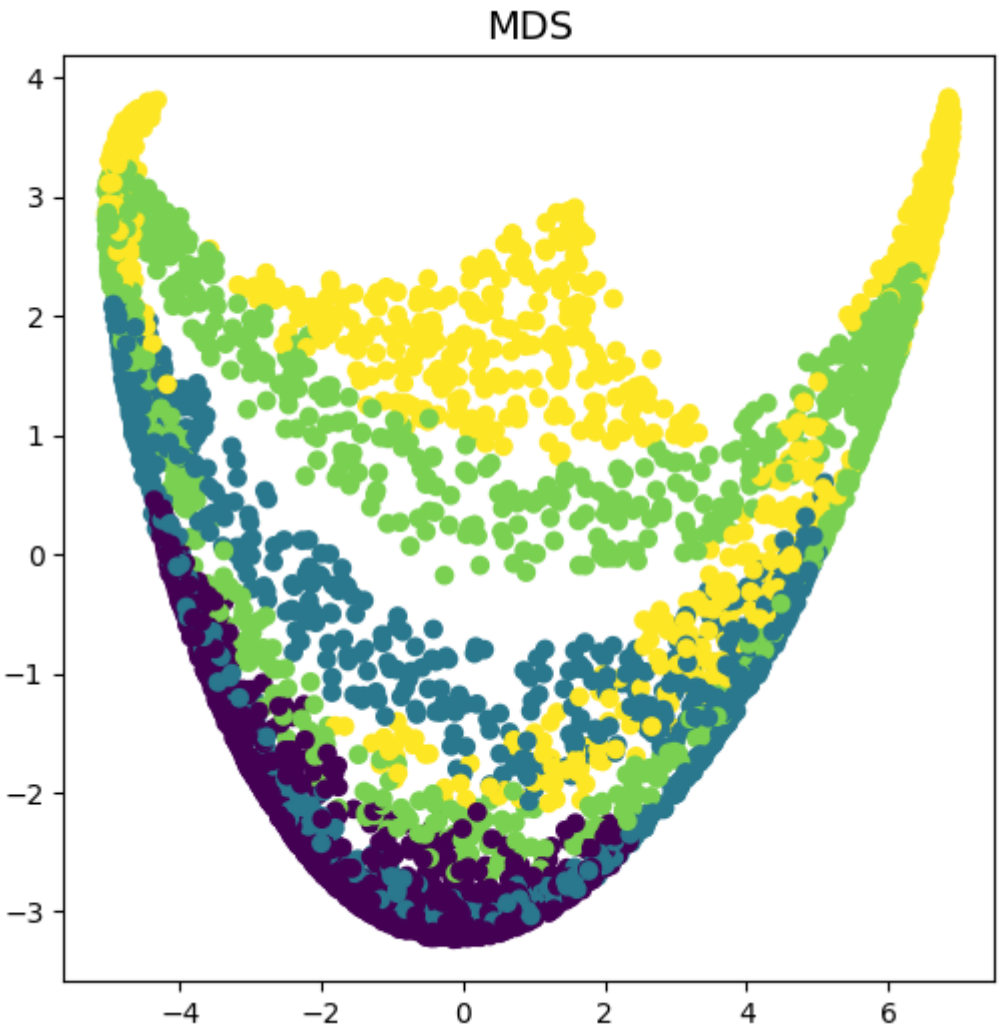
LLE LTSA




```
# LLE LTSA
fig = plt.figure(figsize=(6,6))

model = manifold.LocallyLinearEmbedding(method="ltsa", **params)
X = model.fit_transform(points)
ax = fig.add_subplot(1, 1, 1)
ax.scatter(X[:, 0], X[:, 1], c=colors)
ax.set_title('LLE LTSA', size=14)
```

Copy

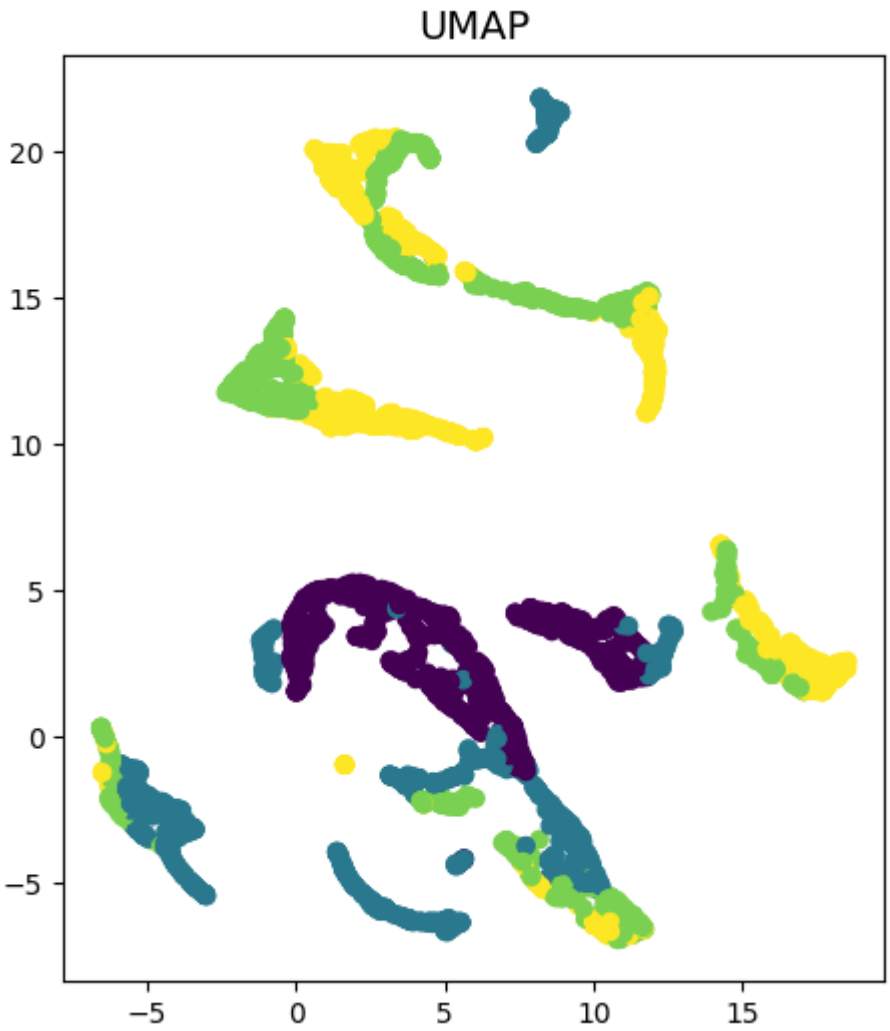
MDS




```
# MDS
fig = plt.figure(figsize=(6,6))

model = MDS(n_components=n_components)
X = model.fit_transform(points)
ax = fig.add_subplot(1, 1, 1)
ax.scatter(X[:, 0], X[:, 1], c=colors)
ax.set_title('MDS', size=14)
```

Copy

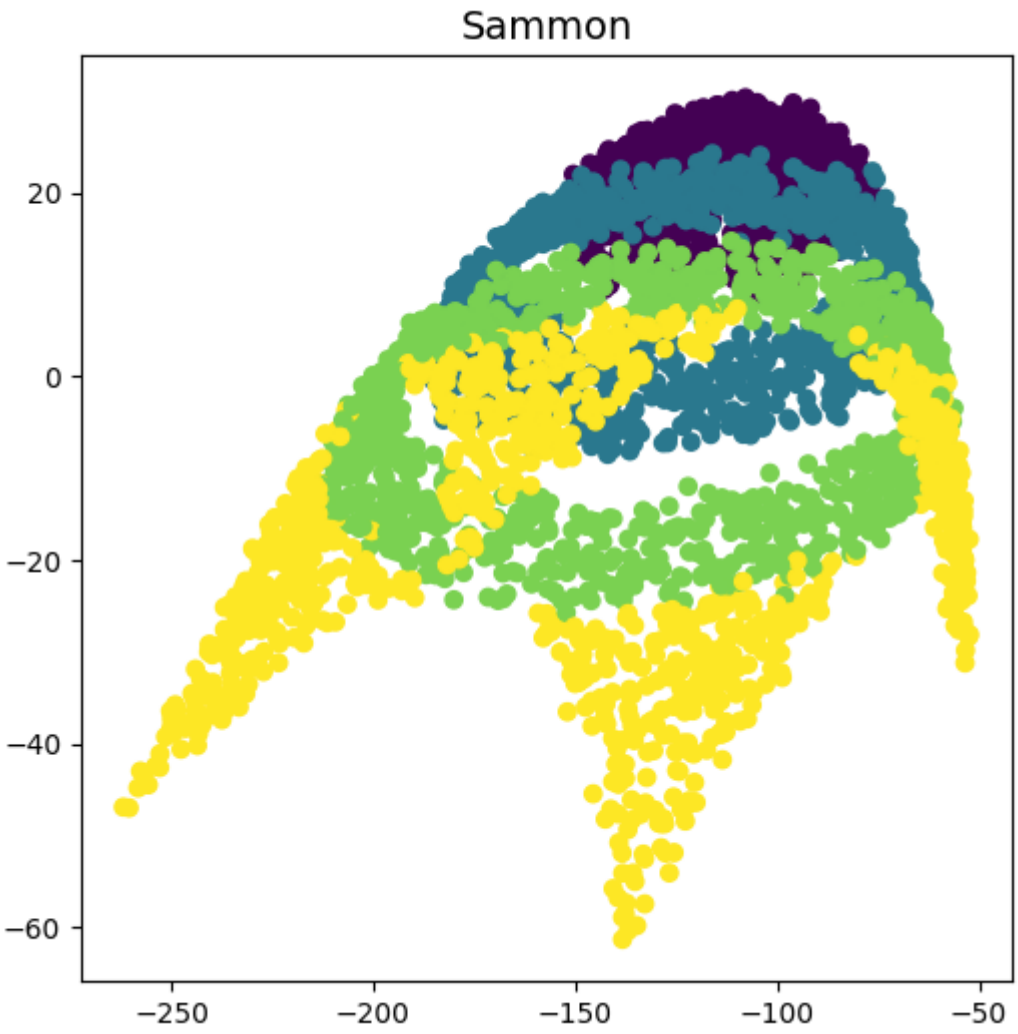
UMAP




```
# UMAP                                                                    Copy
pumap = UMAP(n_components=n_components, init="random", random_state=0)
sr_umap = pumap.fit_transform(points)

fig = plt.figure(figsize=(18, 6))
ax = fig.add_subplot(1, 3, 1)
ax.scatter(sr_umap[:, 0], sr_umap[:, 1], c=colors)
ax.set_title('UMAP', size=14)
```

Parte opcional 1

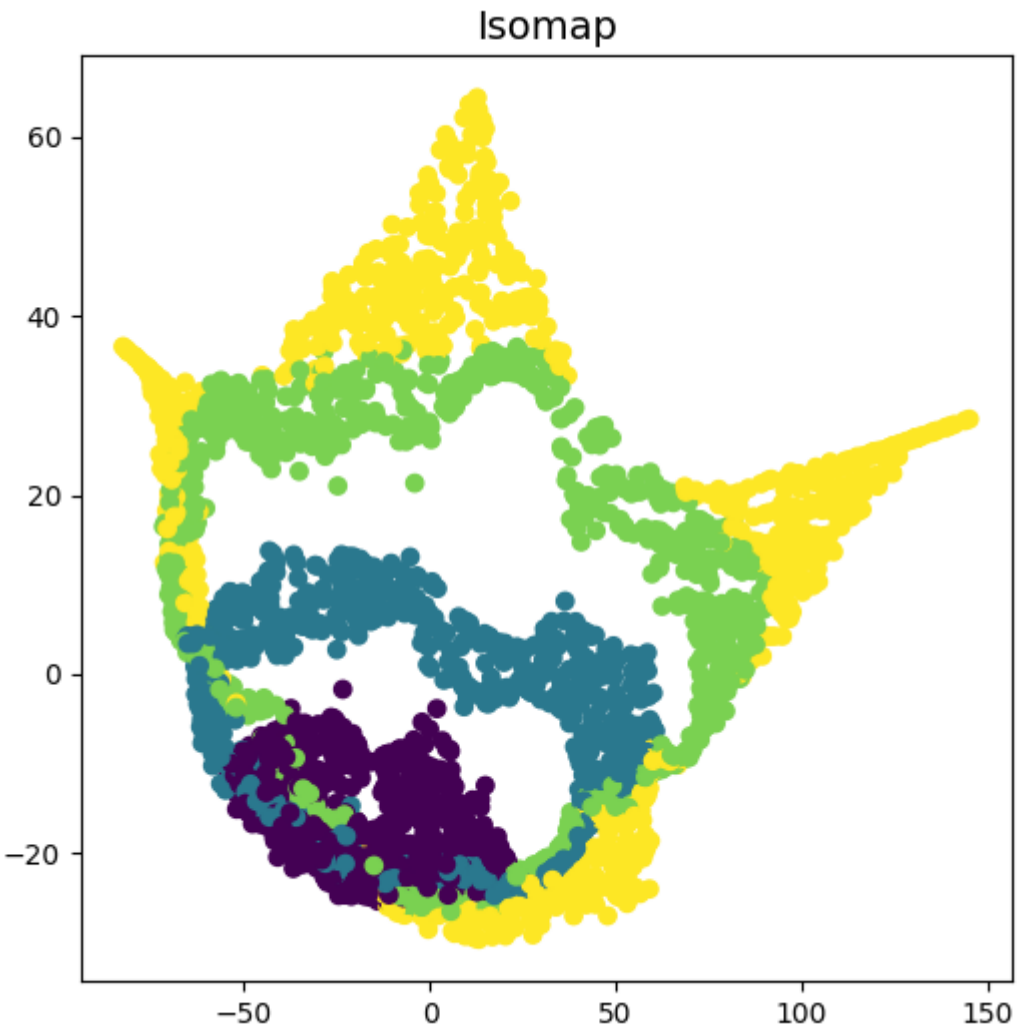
Sammon




```
# Sammon
fig = plt.figure(figsize=(6, 6))

(sp, index) = np.unique(points, axis=0, return_index=True)
(y, E) = sammon(sp, 2)
ax = fig.add_subplot(1, 1, 1)
ax.scatter(y[:, 0], y[:, 1], c=color[index])
ax.set_title('Sammon', size=14)
```

Copy

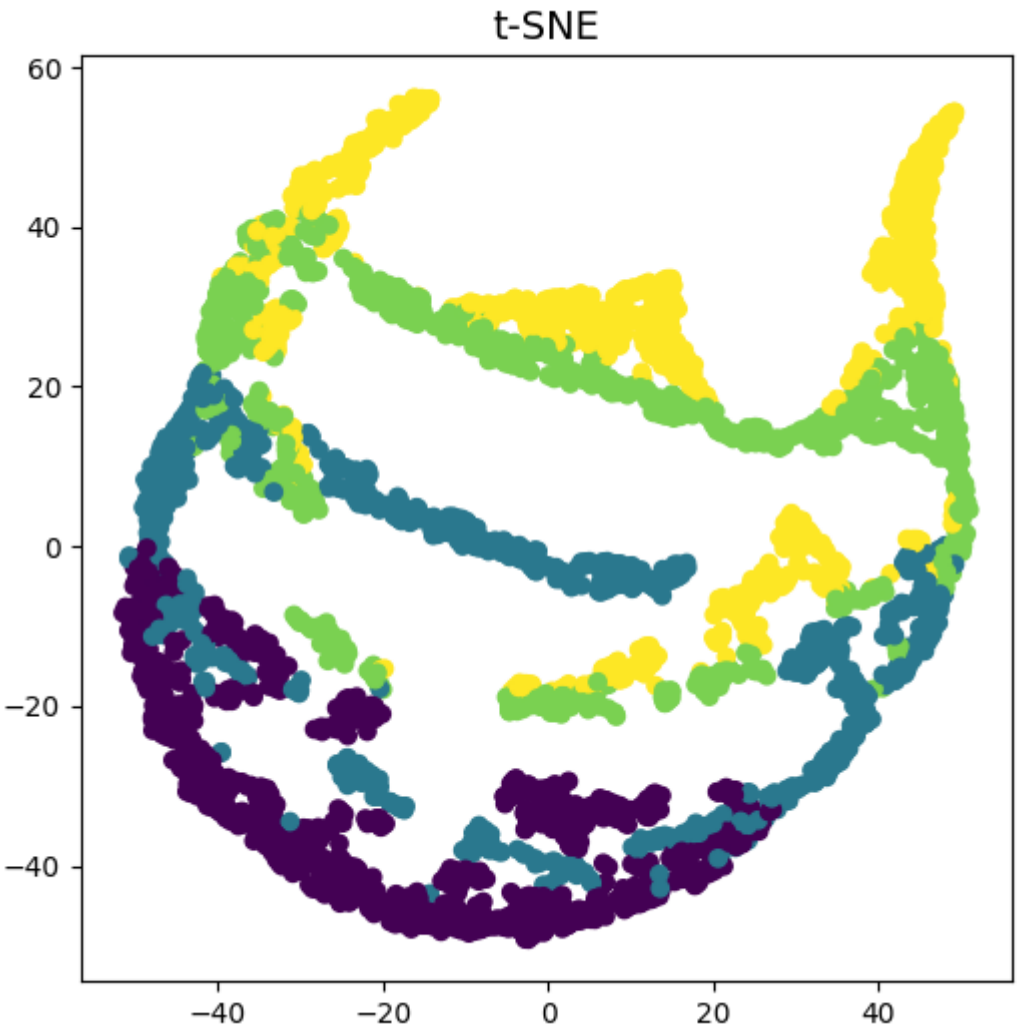
Isomap




```
# Isomap
fig = plt.figure(figsize=(6, 6))

model = Isomap(n_neighbors=n_neighbors, n_components=n_components)
X = model.fit_transform(points)
ax = fig.add_subplot(1, 1, 1)
ax.scatter(X[:, 0], X[:, 1], c=colors)
ax.set_title('Isomap', size=14)
```

Copy

t-SNE




```
# t-SNE
fig = plt.figure(figsize=(6, 6))

sr_tsne = manifold.TSNE(n_components=n_components, perplexity=40,
random_state=0).fit_transform(points)
ax = fig.add_subplot(1, 1, 1)
ax.scatter(sr_tsne[:, 0], sr_tsne[:, 1], c=colors)
ax.set_title('t-SNE', size=14)
```

Copy

Parte opcional 2

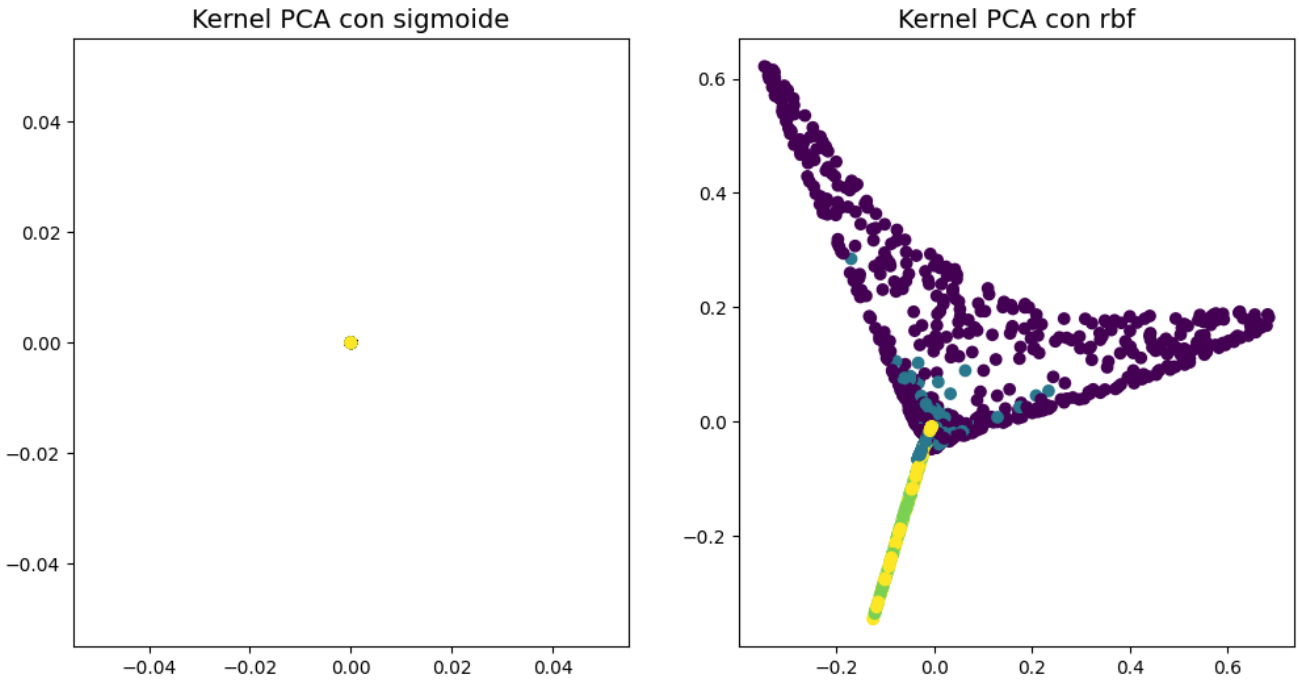
Aprovechando lo que se ve en el notebook de teoría, se muestran las dos opciones que se piden en columnas separadas.

```
# Kernel PCA con sigmoide y rbf
fig, axes = plt.subplots(figsize=(12, 6), ncols=2)

model = KernelPCA(n_components=n_components, kernel="sigmoid", gamma=0.05)
X = model.fit_transform(points)
ax = axes[0]
ax.scatter(X[:, 0], X[:, 1], c=colors)
ax.set_title('Kernel PCA con sigmoide', size=14)
```

Copy

```
model = KernelPCA(n_components=n_components, kernel="rbf")
X = model.fit_transform(points)
ax = axes[1]
ax.scatter(X[:, 0], X[:, 1], c=colors)
ax.set_title('Kernel PCA con rbf', size=14)
```

Resultados



Reflexión

Como se puede observar, existen algunos problemas de hiperparámetros que hacen que los modelos seleccionados no cumplan su función correctamente.

Para tratar de arreglar este problema, se han probado los siguientes hiperparámetros:

- **Cambiar las gammas**. Las IAs generativas sugerían 10 de gamma, con resultados nefastos. El notebook de teoría utiliza 0.05 para PCA con sigmoide y nada para PCA con rbf. También se ha probado intercambiando todos estos valores entre sí, y valores intermedios, sin ningún resultado positivo.
- **Cambiar el número de componentes.** Sin ningún resultado, lo cual es lógico porque durante toda la práctica se utiliza el mismo valor.
- **Introducir parámetros nuevos.** Por ejemplo, `coef0`, sin ningún resultado aparente.

Puesto que el primer modelo agrupa todos los puntos en $[0, 0]$, no se tiene en cuenta a la hora de analizar los resultados.

Análisis

Tras observar los resultados obtenidos por todos los métodos, se llega a las siguientes conclusiones:

1. La configuración es muy relevante en esta práctica, ya que un cambio en un hiperparámetro puede provocar que un modelo produzca resultados drásticamente diferentes o directamente

no funcionen.

2. La reducción dimensional provoca que la mayoría de resultados no sean fáciles de predecir.
3. Los mejores modelos son aquellos que consigan separar más los puntos dependiendo de su clasificación (como *t-SNE* o *UMAP*)
4. Otros modelos, como *Sammon* o *Isomap*, consiguen buenos resultados pese a existir intersecciones entre las clases.
5. El resto de modelos no cumplen las expectativas.