Librerías

```
# Librerias:

import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
import json
from sklearn.experimental import enable_iterative_imputer
from sklearn.impute import IterativeImputer
import pickle
from sklearn.preprocessing import PolynomialFeatures
from sklearn.preprocessing import MinMaxScaler, MaxAbsScaler,
StandardScaler
from sklearn.feature_selection import VarianceThreshold
from sklearn.decomposition import PCA
from sklearn import linear_model

import matplotlib.pyplot as plt
```

Tratamiento inicial de datos:

```
# Guardar datos en una variable:

data = pd.read_csv('/content/sample_data/mnist_test.csv', header=None)
#                                                    sep=';',
decimal='.', index_col=0, na_values='?')

# Separacion en train y test:

train, test = train_test_split(data, test_size= 0.20, random_state= 8)

# Info de los datos:

print(train.info())                                          # Info de los
datos
print(train.describe())                                      # Info
estadística básica
print('\nModas:\n', train.mode(axis=0, dropna=False))        # Tabla de
modas
N_x, D_x = train.shape                                       # Guardar el
tamño en variables
```

Modificación de la tabla de datos:

```
# Sacar etiquetas de los datos:

```

```
y_train = train[0]
train.drop(0, axis=1, implace=True)
```

Modificación de los datos:

```
# Codificación de los datos tipo 'object':

def encode_object_columns(df):
  df_code = df.copy()
  code_to_categ = {}
  for col in df_code.columns:
    if df_code[col].dtype == 'object':
      df_code[col] = df_code[col].astype('category')
      codes = df_code[col].cat.codes.replace(-1, np.nan) # Reemplaza -
1 por NaN
      code_to_categ[col] = dict(zip(codes, df_code[col]))
      df_code[col] = codes
  return df_code, code_to_categ

def encode_etiquetas(df):
  df_code = df.copy()
  for i in df_code.index:
    if df_code[i] != 1:
      df_code[i] = 0
  return df_code

x_train_df, code_to_categ_X = encode_object_columns(train)
y_train_df = encode_etiquetas(y_train)
```

Valores NAN

```
# Visualización de valores NaN:

missing_data = x_train_df.isna()
missing_values_per_column = missing_data.sum(axis=0) # 'NA' por cada
columna
missing_values_per_row = missing_data.sum(axis=1) # 'NA' por cada fila
mask_mayorq0 = missing_values_per_column > 0 # Crea una máscara
de␣Pandas para indicar si hay columnas con NA
mask_mayorq1 = missing_values_per_row > 0 # Crea una máscara de␣Pandas
para indicar si hay filas con NA

missing_count_row = missing_values_per_row.value_counts().sort_index()
print(f'Valores NaN en cada fila:\n{missing_count_row}')
missing_count_col =
missing_values_per_column.value_counts().sort_index()
print(f'Valores NaN en cada columna:\n{missing_count_col}')
```

```
# Crear columna dummies valores NaN

for col in x_train_df.columns:
    x_train_df[col + '_isna'] = x_train_df[col].isna().astype(int)

# Sustitución de valores NaN por valor 'mean' columnas 'rbc':

mean_rbc = x_train_df['rbc'].mean()
x_train_df['rbc'].fillna(mean_rbc, inplace=True)
```

Columnas Dummies

```
data_dummies = pd.get_dummies(x_train_df['columna_categorica'],
prefix='columna_categorica')    # Crea el DataFrame con las columnas
dummies de la columna indicada
x_train_df = pd.concat([x_train_df, data_dummies], axis=1)
# Concatena los DataFrames
```

Imputación multivariante

```
#-- IMPUTACIÓN MULTIVARIANTE --#

# Creas una copia del DataFrame original:
x_train_df_copy = x_train_df.copy()

# Imputación multivariante de los datos NaN:
imputer = IterativeImputer()
train_imputed = imputer.fit_transform(x_train_df_copy)
x_train_df = pd.DataFrame(train_imputed,
columns=x_train_df_copy.columns)   # Convertir de nuevo a DataFrame
print(x_train_df.isna().sum())
print('\nTamaño x_train_df: ', x_train_df.shape)
```

Aumento de dimensionalidad

```
# Aumento de dimensionalidad:

degree = 2                  # número de columnas convinadas a demás de
las iniciales
interaction_only = True

polyf = PolynomialFeatures(degree=degree,
interaction_only=interaction_only)
polyf.set_output(transform="pandas")

x_train_dim = polyf.fit_transform(x_train_df)
print('Tamaño x_train_dim: ',x_train_dim.shape)
```

Escalados a diferentes intervalos

```
# Escalado a intervalo unidad [0,1]:
scalerUnit = MinMaxScaler()
x_train_dim_unit = scalerUnit.fit_transform(x_train_dim)

# Escalado al máx. de los valores abs.:
scalerMaxAbs = MaxAbsScaler()
x_train_dim_MaxAbs = scalerMaxAbs.fit_transform(x_train_dim)

# Estandarizamos:
scalerStd = StandardScaler()
x_train_dim_std = scalerStd.fit_transform(x_train_dim)
```

Filtrado

```
# Filtrado por varianza:
selector_var = VarianceThreshold()
selector_var.set_output(transform="pandas")

x_train_unit_var = selector_var.fit_transform(x_train_dim_unit)
x_train_MaxAbs_var = selector_var.fit_transform(x_train_dim_MaxAbs)
x_train_sdt_var = selector_var.fit_transform(x_train_dim_std)

# Filtrado por correlación:

def drop_highly_correlated_columns(df):
    df = pd.DataFrame(df)   # Crea un DataFrame de pandas
    df_corr = df.corr().abs()  # Calculas la matriz de correlación
    upper = df_corr.where(np.triu(np.ones(df_corr.shape),
k=1).astype(bool))    # Seleccionas el triángulo superior de la matriz
de correlación
    to_drop = [column for column in upper.columns if any(upper[column]
> 0.90)]  # Encuentras las columnas con correlación mayor a 0.9
(puedes ajustar este valor a tus necesidades)    to_drop = [column for
column in upper.columns if any(upper[column] > 0.9)]
    col_dropped = list(to_drop) # Guarda la lista de columnas
eliminadas
    df_ret = df.drop(col_dropped, axis=1)    # Eliminas las columnas
altamente correlacionadas
    return df_ret, col_dropped

x_train_corr, col_drop_corr =
drop_highly_correlated_columns(x_train_dim)
x_train_unit_corr, col_drop_unit =
drop_highly_correlated_columns(x_train_dim_unit)
x_train_MaxAbs_corr, col_drop_MaxAbs =
drop_highly_correlated_columns(x_train_dim_MaxAbs)
x_train_std_corr, col_drop_std =
drop_highly_correlated_columns(x_train_dim_std)
```

Entrenamiento de los modelos lineales

```
reg_model_unit = linear_model.LinearRegression()
reg_model_unit.fit(x_train_unit_var, y_train_df['class'])

reg_model_MaxAbs = linear_model.LinearRegression()
reg_model_MaxAbs.fit(x_train_MaxAbs_var, y_train_df['class'])

reg_model_std = linear_model.LinearRegression()
reg_model_std.fit(x_train_sdt_var, y_train_df['class'])

reg_model_corr = linear_model.LinearRegression()
reg_model_corr.fit(x_train_corr, y_train_df['class'])

reg_model_unit_corr = linear_model.LinearRegression()
reg_model_unit_corr.fit(x_train_unit_corr, y_train_df['class'])

reg_model_MaxAbs_corr = linear_model.LinearRegression()
reg_model_MaxAbs_corr.fit(x_train_MaxAbs_corr, y_train_df['class'])

reg_model_std_corr = linear_model.LinearRegression()
reg_model_std_corr.fit(x_train_std_corr, y_train_df['class'])
```

PCA

```
# Estandarizamos primero:

print('Tamaño del DataFrame original: ', x_train_df.shape)
scaler = StandardScaler().set_output(transform="pandas")
scaler.fit(x_train_df)
x_train_df_std = scaler.transform(x_train_df)


# PCA seleccionando diréctamente el número de componentes o un
porcentaje de información que queremos mantener:

n_components = 0.90 # si se pone un núm. entero (3) sería el núm. de
columnas que mantendríamos

pca = PCA(n_components= n_components).set_output(transform='pandas')
pca.fit(x_train_df_std)
x_train_pca = pca.transform(x_train_df_std)

print('Tamaño del nuevo DataFrame: ', x_train_pca.shape, f'\n\nTabla
con los componentes principales hasta explicar el {n_components*100}%
de la varianza')

# Entrenamos el modelo PCA:

reg_model_pca = linear_model.LinearRegression()
reg_model_pca.fit(x_train_pca, y_train_df['class'])
```

```
# Visualización del peso de cada columna calculada:

def plot_PCA(pca):
  plt.stem(pca.explained_variance_ratio_.cumsum(),'b')
  plt.stem(pca.explained_variance_ratio_,'r')

  titleStr = 'Varianza explicada por cada componente principal (rojo)'
  titleStr = titleStr+'\n'
  titleStr = titleStr+'Varianza explicada acumulada con cada
componente principal (azul)'
  plt.title(titleStr, fontsize=10)
  ax = plt.gca()
  ax.axis([-0.1,1.1,0,1])
  ax.set_xticks([i for i in range(pca.n_components_)])
  ax.set_xticklabels(["pca"+str(i) for i in range(pca.n_components_)])

  fig = plt.gcf()
  fig.set_size_inches(6,2)

  plt.show()

plot_PCA(pca)
```

TEST

Codificación

```
# Codificación de los datos de test:

def apply_encoding(df_test, encoding_dict):
    for col, mapping in encoding_dict.items():
        df_test[col] = df_test[col].map(lambda x: mapping.get(x, x))
    return df_test

x_test_df = apply_encoding(x_test, encoding_dict_x) # DataFrame test,
diccionario_de_codificacion de datos
y_test_df = apply_encoding(y_test, encoding_dict_y) # DataFrame
etiquetas test, diccionario_de_codificacion de etiquetas
```

Imputación multivariante

```
# Imputación multivariante de los datos NaN:

test_imputed = loaded_imputer.transform(x_test_df_copy)
x_test_df = pd.DataFrame(test_imputed, columns=x_test_df_copy.columns)
# Convertir de nuevo a DataFrame
print(x_test_df.isna().sum())
print('\nTamaño x_test_df: ', x_test_df.shape)
```

Aumento de dimensionalidad

```
# Aumento de dimensionalidad:
x_test_dim = loaded_polyf.transform(x_test_df)
print('Tamaño x_test_dim: ',x_test_dim.shape)
```

Escalados a diferentes intervalos

```
# Escalado a intervalo unidad [0,1]:
x_test_dim_unit = loaded_scalerUnit.transform(x_test_dim)

# Escalado al máx. de los valores abs.:
x_test_dim_MaxAbs = loaded_scalerMaxAbs.transform(x_test_dim)

# Estandarizamos:
x_test_dim_std = loaded_scalerStd.transform(x_test_dim)
```

Filtrado

```
# Filtrado por varianza:

# x_test_var = selector_var.transform(x_test_dim)
x_test_unit_var = loaded_selector_var.transform(x_test_dim_unit)
x_test_MaxAbs_var = loaded_selector_var.transform(x_test_dim_MaxAbs)
x_test_sdt_var = loaded_selector_var.transform(x_test_dim_std)

# Filtrado por correlación:
def drop_col_test(df, l_col):
    df = pd.DataFrame(df)
    return df.drop(columns=l_col)

x_test_corr = drop_col_test(x_test_dim, loaded_col_drop_corr)
x_test_unit_corr = drop_col_test(x_test_dim_unit,
loaded_col_drop_unit)
x_test_MaxAbs_corr = drop_col_test(x_test_dim_MaxAbs,
loaded_col_drop_MaxAbs)
x_test_std_corr = drop_col_test(x_test_dim_std, loaded_col_drop_std)
```

PCA - test

```
# Estandarizamos primero para PCA:
X_test_df_std = scaler.transform(X_test_df)

# PCA seleccionando diréctamente el número de componentes o un
porcentaje de␣información que queremos mantener:
X_test_pca = pca.transform(X_test_df_std)
```

COMPARACIÓN DE MODELOS

```
# Cálculo de las probabilidades ScalerUnit:
y_score_scalerUnit = loaded_reg_model_unit.predict(x_test_unit_var)
# Cálculo de la curva ROC y AUC
fpr_scalerUnit, tpr_scalerUnit, thresholds =
roc_curve(y_test_df['class'], y_score_scalerUnit)
roc_auc_scalerUnit = auc(fpr_scalerUnit, tpr_scalerUnit)

# Cálculo de las probabilidades MaxAbs:
y_score_MaxAbs = loaded_reg_model_MaxAbs.predict(x_test_MaxAbs_var)
# Cálculo de la curva ROC y AUC
fpr_MaxAbs, tpr_MaxAbs, thresholds_MaxAbs =
roc_curve(y_test_df['class'], y_score_MaxAbs)
roc_auc_MaxAbs = auc(fpr_MaxAbs, tpr_MaxAbs)

# Cálculo de las probabilidades Std:
y_score_Std = loaded_reg_model_std.predict(x_test_sdt_var)
# Cálculo de la curva ROC y AUC
fpr_Std, tpr_Std, thresholds_Std = roc_curve(y_test_df['class'],
y_score_Std)
roc_auc_Std = auc(fpr_Std, tpr_Std)


# Cálculo de las probabilidades Corr:
y_score_corr = loaded_reg_model_corr.predict(x_test_corr)
# Cálculo de la curva ROC y AUC
fpr_corr, tpr_corr, thresholds_corr = roc_curve(y_test_df['class'],
y_score_corr)
roc_auc_corr = auc(fpr_corr, tpr_corr)

# Cálculo de las probabilidades ScalerUnit_corr:
y_score_scalerUnit_corr =
loaded_reg_model_unit_corr.predict(x_test_unit_corr)
# Cálculo de la curva ROC y AUC
fpr_ScalerUnit_corr, tpr_ScalerUnit_corr, thresholds_ScalerUnit_corr =
roc_curve(y_test_df['class'], y_score_scalerUnit_corr)
roc_auc_ScalerUnit_corr = auc(fpr_ScalerUnit_corr,
tpr_ScalerUnit_corr)

# Cálculo de las probabilidades MaxAbs_corr:
y_score_MaxAbs_corr =
loaded_reg_model_MaxAbs.predict(x_test_MaxAbs_var)
# Cálculo de la curva ROC y AUC
fpr_MaxAbs_corr, tpr_MaxAbs_corr, thresholds =
roc_curve(y_test_df['class'], y_score_MaxAbs_corr)
roc_auc_MaxAbs_corr = auc(fpr_MaxAbs_corr, tpr_MaxAbs_corr)

# Cálculo de las probabilidades Std_corr:
y_score_Std_corr = loaded_reg_model_std_corr.predict(x_test_std_corr)
```

```
# Cálculo de la curva ROC y AUC
fpr_Std_corr, tpr_Std_corr, thresholds_Std_corr =
roc_curve(y_test_df['class'], y_score_Std_corr)
roc_auc_Std_corr = auc(fpr_Std_corr, tpr_Std_corr)


# Cálculo de las probabilidades PCA:
y_score_pca = loaded_reg_model_pca.predict(x_test_pca)
# Cálculo de la curva ROC y AUC
fpr_pca, tpr_pca, thresholds_pca = roc_curve(y_test_df['class'],
y_score_pca)
roc_auc_pca = auc(fpr_pca, tpr_pca)
```

La variable `thresholds_...` es el umbral óptimo del modelo

Visualización de modelos

```
# Visualización de la curva ROC Var:
# plt.plot(fpr_var, tpr_var, label='Modelo Var (AUROC = %0.2f)' %
roc_auc_var)

# Visualización de la curva ROC ScalerUnit:
plt.plot(fpr_scalerUnit, tpr_scalerUnit, label='Modelo ScalerUnit
(AUROC = %0.2f)' % roc_auc_scalerUnit)

# Visualización de la curva ROC MaxAbs:
plt.plot(fpr_MaxAbs, tpr_MaxAbs, label='Modelo MaxAbs (AUROC = %0.2f)'
% roc_auc_MaxAbs)

# Visualización de la curva ROC Std:
plt.plot(fpr_Std, tpr_Std, label='Modelo Std (AUROC = %0.2f)' %
roc_auc_Std)

# Visualización de la curva ROC Corr:
plt.plot(fpr_corr, tpr_corr, label='Modelo Corr (AUROC = %0.2f)' %
roc_auc_corr)

# Visualización de la curva ROC ScalerUnit_corr:
plt.plot(fpr_ScalerUnit_corr, tpr_ScalerUnit_corr, label='Modelo
ScalerUnit_corr (AUROC = %0.2f)' % roc_auc_ScalerUnit_corr)

# Visualización de la curva ROC MaxAbs_corr:
plt.plot(fpr_MaxAbs_corr, tpr_MaxAbs_corr, label='Modelo MaxAbs_corr
(AUROC = %0.2f)' % roc_auc_MaxAbs_corr)

# Visualización de la curva ROC Std:
plt.plot(fpr_Std_corr, tpr_Std_corr, label='Modelo Std_corr (AUROC =
%0.2f)' % roc_auc_Std_corr)

# Visualización de la curva ROC PCA:
```

```
plt.plot(fpr_pca, tpr_pca, label='Modelo PCA (AUROC = %0.2f)' %
roc_auc_pca)



# Curva de no discriminación
plt.plot([0, 1], [0, 1], color='navy', linestyle='--')

# Configuración de los límites del gráfico
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])

# Etiquetas y título
plt.xlabel('Tasa de Falsos Positivos')
plt.ylabel('Tasa de Verdaderos Positivos')
plt.title('Comparación de las Curvas ROC')
plt.legend(loc="lower right")

# Mostrar la gráfica
plt.show()
```

Matriz de confusión

```
 # Matriz de confusión

cm = confusion_matrix(y_test_df['class'], y_pred)

# Guarda el objeto imputer en un archivo pickle: (guardar modelos en
archivo externo)

with open('imputer.pkl', 'wb') as file:
    pickle.dump(imputer, file)
```

Librerías

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
from sklearn.feature_selection import VarianceThreshold
from sklearn.decomposition import PCA
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import confusion_matrix
```

Descarga el conjunto de datos

```
DF = pd.read_csv("./sample_data/mnist_train_small.csv", header = None)
```

Divide el df en train-test

```
test_size = 0.2
df_x, df_val = train_test_split(DF, test_size=test_size, random_state=3)
```

Extrae la primera columna (etiquetas) y elimina la primera columna del df original

```
print(df_x.shape)
df_y = df_x[0]

print(df_y.shape)
df_x = df_x.drop(columns=0)

print(df_x.shape)
print(df_x.info())
```

Mapea las etiquetas

```
# Mapear df_y
d = {3: 1, 1: 0, 2: 0, 4: 0, 5: 0, 6: 0, 7: 0, 8: 0, 9: 0}
df_y = pd.DataFrame(map(df_y, d))
print(df_y.head)
```

Inicializa las transformaciones del modelo

```
minmax_scaler = MinMaxScaler().set_output(transform='pandas')
var_filter = VarianceThreshold(threshold=0.1).set_output(transform="pandas")
pca = PCA(n_components=.9)
reg_logi = LogisticRegression(max_iter=1000)
```

Aplica las transformaciones

```
df_x = minmax_scaler.fit_transform(df_x)
df_x = var_filter.fit_transform(df_x)
df_x = pca.fit_transform(df_x)
```

Entrena el modelo

```
reg_logi.fit(df_x, df_y)
```

Repite el proceso con el test

```
df_valy = df_val[0]
df_val = df_val.drop(columns=0)
df_val = minmax_scaler.transform(df_val)
df_val = var_filter.transform(df_val)
df_val = pca.transform(df_val)
```

Aplica el modelo

```
y_hat = reg_logi.predict(df_val)
```

Calcula e imprime la matriz de confusión comparando las etiquetas verdaderas df_valy con las predicciones y_hat.

```
conf_matrix = confusion_matrix(df_valy, y_hat)
print(conf_matrix)
```