Machine Learning

U2 - Implementing a Predictor from scratch

Author:

Christian Adriel Rodriguez

29/10/2023

INTRODUCTION

Creating new models independent of external dependencies or using pre-existing libraries is a crucial option in the field of predictive modeling. This study offers a comparison of these two methods of implementation. Our goal is to clarify the benefits, drawbacks, and differences in performance that each approach has.

This analysis is helpful to people who are looking for a useful reference for their decision-making process as well as those who are trying to understand the ramifications of the decisions they are making for implementation. Readers will be better prepared to make decisions based on their unique needs, limits, and goals after reading this study. Come along as we investigate the science and art of prediction.

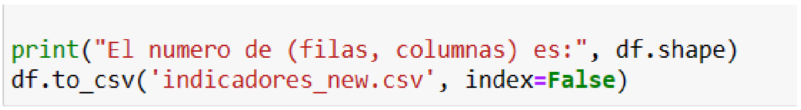
DEVELOPMENT

During the development section I will explain the process of: Cleaning correctly the dataset, implementing of a predictor by scratch and finally the implementation of predictor by using a library.

Data preparation:

1. First step: Data processing is the process before the implementation of a predictive model. During this process we prepare a data set, eliminating all NaN values, analysing of features, observations, and the understanding of does our data is. However, the process was stocked by *pandas'* problem. The letter **"g"**, the letter g problem I supposed that is originated by the panda's system of writing. The position oh the letter is different than the system of my jupyter notebook. So, when I start to elaborate the data pre-processing procedure the library did not find the columns in question.

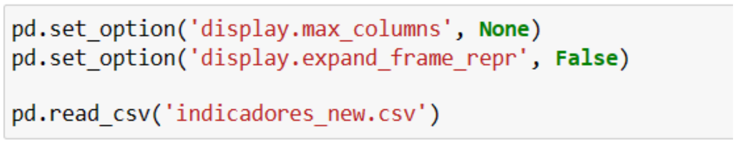
2. Part of the data-preprocessing, we must eliminate whole values Null and NA. Afterward, the procedure starts identifying all these error values, then we ought to change those to a general variable. In this case NaN values.



```python
print("El numero de (filas, columnas) es:", df.shape)
df.to_csv('indicadores_new.csv', index=False)
```

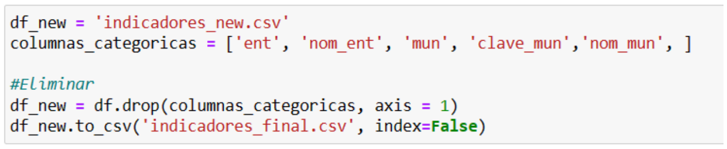
To verify the correct operation, we used the following part of the code to check the if there are columns with NaN values.

3. My problem when I tried to verify the elimination of the columns that It won't be useful for the problem. I used the code, to display the whole columns.

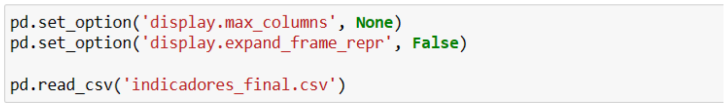


```python
pd.set_option('display.max_columns', None)
pd.set_option('display.expand_frame_repr', False)

pd.read_csv('indicadores_new.csv')
```

4. In order of the data-preprocessing, it is important to acknowledge the objective of our data case, then identify the columns that won't be used. To facilitate the calculus, all values must be numeric. Therefore, columns as Entity, Number, Name of the entity won't be used during the coding process.



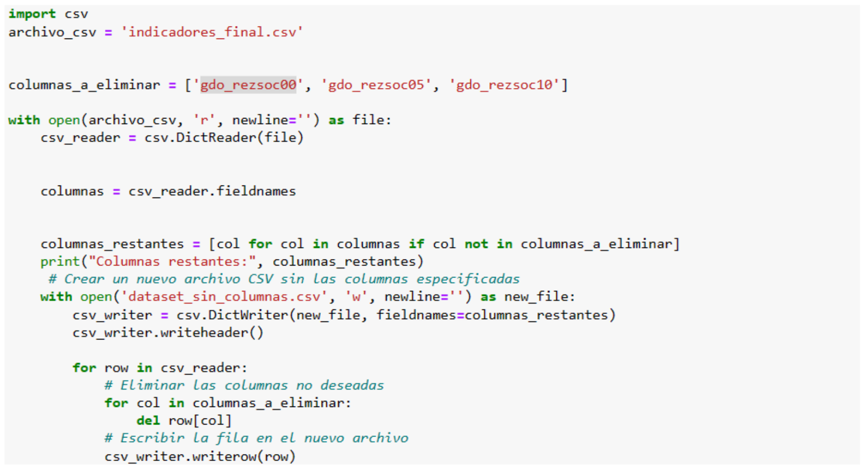
```python
df_new = 'indicadores_new.csv'
columnas_categoricas = ['ent', 'nom_ent', 'mun', 'clave_mun','nom_mun', ]

#Eliminar
df_new = df.drop(columnas_categoricas, axis = 1)
df_new.to_csv('indicadores_final.csv', index=False)
```



```python
pd.set_option('display.max_columns', None)
pd.set_option('display.expand_frame_repr', False)

pd.read_csv('indicadores_final.csv')
```

5. ***Problem with the elimination of the categoric columns.*** As I mentioned in the first point. I had been having problems with the letter g during my operations. So, every time I tried to start doing an operation with the categoric columns, the library ***pandas*** did not find the columns. The message was "The gdo_rezsoc00 does not exist". This was due of the panda's format of the letter **g.** Consequently, I did not have the opportunity to use those columns.

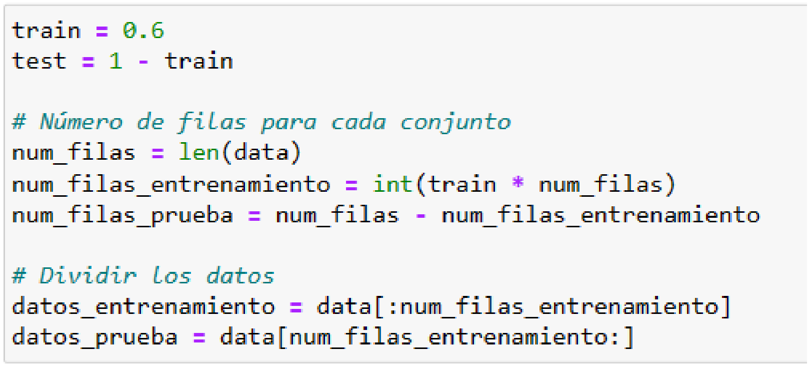
   *To rectify this issue. I opted to use the **CSV** library to erase the columns. As the following code:*



```python
import csv
archivo_csv = 'indicadores_final.csv'


columnas_a_eliminar = ['gdo_rezsoc00', 'gdo_rezsoc05', 'gdo_rezsoc10']

with open(archivo_csv, 'r', newline='') as file:
    csv_reader = csv.DictReader(file)


    columnas = csv_reader.fieldnames


    columnas_restantes = [col for col in columnas if col not in columnas_a_eliminar]
    print("Columnas restantes:", columnas_restantes)
     # Crear un nuevo archivo CSV sin las columnas especificadas
    with open('dataset_sin_columnas.csv', 'w', newline='') as new_file:
        csv_writer = csv.DictWriter(new_file, fieldnames=columnas_restantes)
        csv_writer.writeheader()

        for row in csv_reader:
            # Eliminar las columnas no deseadas
            for col in columnas_a_eliminar:
                del row[col]
            # Escribir la fila en el nuevo archivo
            csv_writer.writerow(row)
```

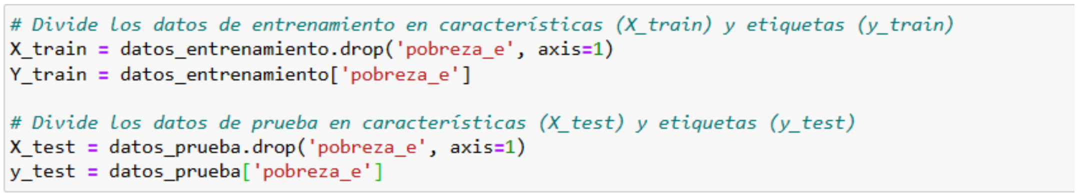
DATA DIVISON

1. In order of the data preparation, the next step is to division of the data. In the machine learning problems. The operator must divide the dataset into ***Train and Test*** data. The normativity is to divide 80% of train and 20% for the data test.



```
train = 0.6
test = 1 - train

# Número de filas para cada conjunto
num_filas = len(data)
num_filas_entrenamiento = int(train * num_filas)
num_filas_prueba = num_filas - num_filas_entrenamiento

# Dividir los datos
datos_entrenamiento = data[:num_filas_entrenamiento]
datos_prueba = data[num_filas_entrenamiento:]
```

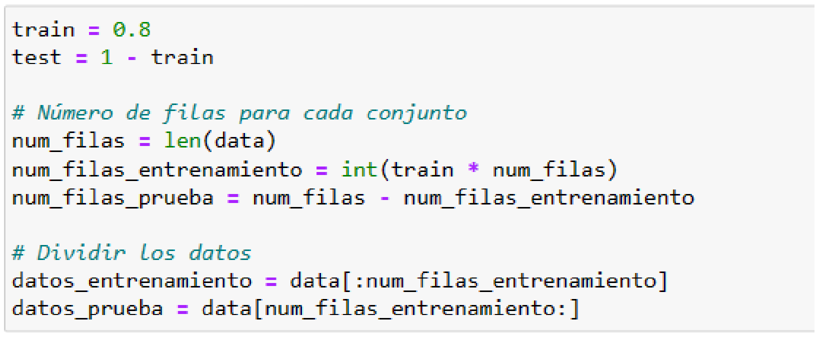
2. Now, it is necessary to divide the train and test dataset in X and Y each of them. For a machine learning task, this algorithm splits a dataset into training and testing subsets. To prepare the training and testing data for model training and evaluation, it first isolates the features (independent variables) from the target variable (dependent variable).



```
# Divide los datos de entrenamiento en características (X_train) y etiquetas (y_train)
X_train = datos_entrenamiento.drop('pobreza_e', axis=1)
Y_train = datos_entrenamiento['pobreza_e']

# Divide los datos de prueba en características (X_test) y etiquetas (y_test)
X_test = datos_prueba.drop('pobreza_e', axis=1)
y_test = datos_prueba['pobreza_e']
```

DATA DIVISON

1. Now, it is time to declare the divisions we will use during the training process. For the machine learning process, it is necessary to divide the dataset into train and test parts. It crucial when you are looking for a good prediction.
   In this case we will use a division of:

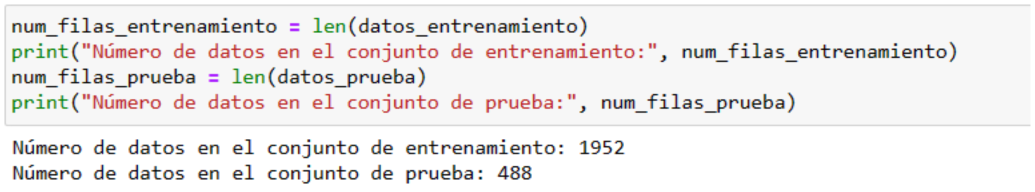


```
train = 0.8
test = 1 - train

# Número de filas para cada conjunto
num_filas = len(data)
num_filas_entrenamiento = int(train * num_filas)
num_filas_prueba = num_filas - num_filas_entrenamiento

# Dividir los datos
datos_entrenamiento = data[:num_filas_entrenamiento]
datos_prueba = data[num_filas_entrenamiento:]
```

80 % of training

20 % of testing

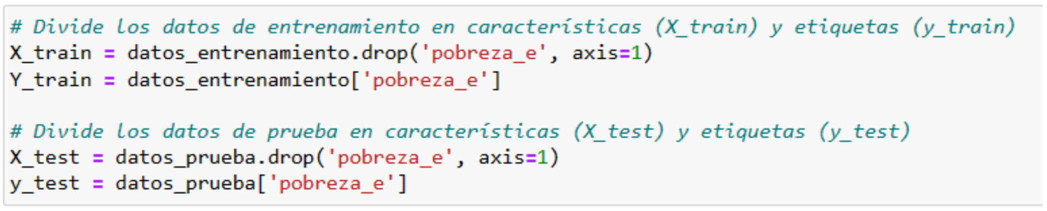
2. To visualize the division of data was done. We used the following code. This show, the amount of data train and data test.



```
num_filas_entrenamiento = len(datos_entrenamiento)
print("Número de datos en el conjunto de entrenamiento:", num_filas_entrenamiento)
num_filas_prueba = len(datos_prueba)
print("Número de datos en el conjunto de prueba:", num_filas_prueba)
```

```
Número de datos en el conjunto de entrenamiento: 1952
Número de datos en el conjunto de prueba: 488
```

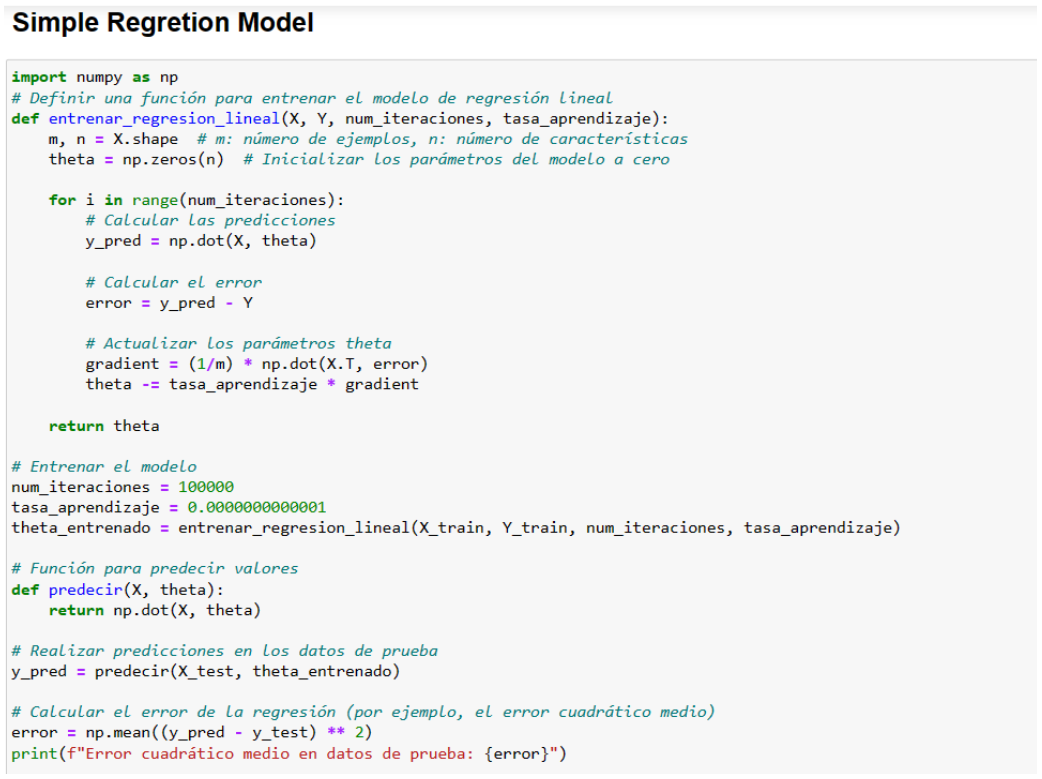
3. To finalize the process of data division. We must divide the train and test into X_TEST and Y_TRAIN.



```
# Divide los datos de entrenamiento en características (X_train) y etiquetas (y_train)
X_train = datos_entrenamiento.drop('pobreza_e', axis=1)
Y_train = datos_entrenamiento['pobreza_e']

# Divide los datos de prueba en características (X_test) y etiquetas (y_test)
X_test = datos_prueba.drop('pobreza_e', axis=1)
y_test = datos_prueba['pobreza_e']
```

SIMPLE REGRESION MODEL

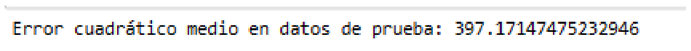
Finally, as part of the assignment of the implementation of a predictor. We will start with the simple regression model. This model, consist of programming a predictor without libraries as Sklearn that facilitates the implementation.



Simple Regretion Model

```
import numpy as np
# Definir una función para entrenar el modelo de regresión lineal
def entrenar_regresion_lineal(X, Y, num_iteraciones, tasa_aprendizaje):
    m, n = X.shape  # m: número de ejemplos, n: número de características
    theta = np.zeros(n)  # Inicializar los parámetros del modelo a cero

    for i in range(num_iteraciones):
        # Calcular las predicciones
        y_pred = np.dot(X, theta)

        # Calcular el error
        error = y_pred - Y

        # Actualizar los parámetros theta
        gradient = (1/m) * np.dot(X.T, error)
        theta -= tasa_aprendizaje * gradient

    return theta

# Entrenar el modelo
num_iteraciones = 100000
tasa_aprendizaje = 0.0000000000001
theta_entrenado = entrenar_regresion_lineal(X_train, Y_train, num_iteraciones, tasa_aprendizaje)

# Función para predecir valores
def predecir(X, theta):
    return np.dot(X, theta)

# Realizar predicciones en los datos de prueba
y_pred = predecir(X_test, theta_entrenado)

# Calcular el error de la regresión (por ejemplo, el error cuadrático medio)
error = np.mean((y_pred - y_test) ** 2)
print(f"Error cuadrático medio en datos de prueba: {error}")
```

1. **Def entrenar_regresion_lineal(X, Y, num_iteraciones, tasa_aprendizaje):** This step is I declared the function "Entrenar_regresion_lineal", this function takes the following variables:
   - **X = independent variables.**
   - **Y = dependent variables.**
   - **Num_iteraciones = Number of epocs to train the model**
   - **Tasa_aprendizaje = The learning rate control the velocity of updates of the model.**
2. Inside the function:
   - **M, N = X.Shape:** Get the dimensions of the matrix "X", where **m** is the number of examples and **"N"** is the number of features.
   - **Theta = np.zeros(n):** Initialize the model parameters to zero.
3. Then, a loop is entered that runs **"num_interaciones"** times.
   - The feature matrix X is multiplied by the parameter vector theta to determine the predictions, or **y_pred**.
   - The difference between the actual labels Y and the predictions **y_pred** is used to calculate error.
   - Gradient descent is used to update the parameters theta, where gradient is the gradient of the error with respect to the parameters.
   - The gradient times the learning rate is subtracted to update the theta parameters.
4. Using the training parameters **theta_entrenado,** the predecir function is then used to make predictions on the test dataset **X_test.**
5. The test dataset's Mean Squared Error (MSE) is computed as the difference between the actual labels, **y_test**, and the predictions, **y_pred**. The outcome is kept in the error variable.
6. To gauge the quality of the model, the MSE is printed at the end.



```
Error cuadrático medio en datos de prueba: 397.17147475232946
```

Why I used the gradient model?

- **Complexity of the Problem:** Predicting poverty rates in Mexican states is a complex problem influenced by numerous socio-economic, demographic, and geographic factors. The relationships between these variables can be intricate and nonlinear. Gradient descent is a powerful optimization technique that allows your model to navigate this complex parameter space to find the optimal set of coefficients that minimize prediction errors.

- **Large Dataset:** Datasets related to socio-economic indicators for multiple Mexican states can be substantial. Gradient descent is efficient and scalable, making it suitable for handling large datasets. It allows your model to process a vast amount of data and learn from it effectively.

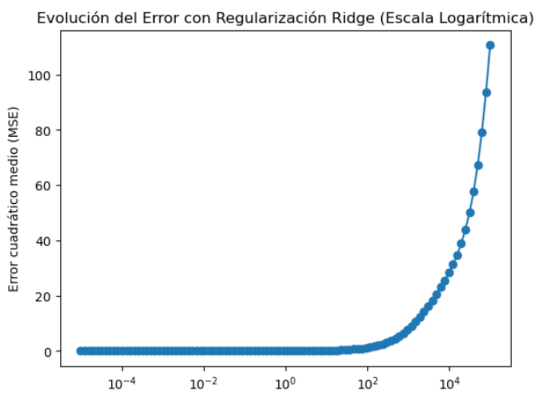
REGRETION MODEL USING LIBRARIES

1. The data set's labels (y) and characteristics (X) are defined. The objective variable, 'pobreza_e', is removed to obtain the characteristics.
2. To make sure that every feature has the same scale, standardization of features is done using the scikit-learn StandardScaler class. This is crucial for models like Ridge regression that rely on the feature scale.
3. The train_test_split function of sklearn is then used to split the data into training and test sets. A test set comprising 20% of the data will be employed, and to guarantee reproducibility, the random_state parameter has been set to 42.
4. Using np.logspace, a logarithmic scale alpha value list is produced. You may experiment with various regularization levels thanks to this.
5. initiates an iterative loop over the alpha values:
   - Using the current alpha value, a Ridge regression model is constructed.
   - The training set of data fits the model.
   - There are predictions in the test set.
   - By comparing predictions with actual tags, get the mean quadratic error (MSE).The outcomes are kept in the results list as dictionaries with the alpha and MSE values.
6. The results list is transformed into a pandas DataFrame called result_df.
7. The DataFrame's idxmin() method finds the optimal alpha value based on the lowest MSE.
8. The best alpha value discovered is presented along with the matching MSE.

9. On a logarithmic scale, the error's evolution is shown versus alpha values. This makes the variation in the MSE with various regularization levels easier to see.

```
# Definir las características y etiquetas
X = datos.drop('pobreza_e', axis=1)
y = datos['pobreza_e']

# Normalización de características
scaler = StandardScaler()
X = scaler.fit_transform(X)

# Dividir los datos en conjuntos de entrenamiento y prueba
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Crear una lista de valores de alpha en una escala logarítmica
alphas = np.logspace(-5, 5, num=100)

# Almacenar resultados en una lista de diccionarios
resultados = []

for alpha in alphas:
    # Crear un modelo de regresión Ridge
    modelo = Ridge(alpha=alpha)

    modelo.fit(X_train, y_train)

    y_pred = modelo.predict(X_test)

    mse = mean_squared_error(y_test, y_pred)

    resultados.append({'Alpha': alpha, 'MSE': mse})

# Convertir la lista de resultados en un DataFrame
resultados_df = pd.DataFrame(resultados)

# Encontrar el mejor valor de alpha basado en el menor MSE
mejor_fila = resultados_df.loc[resultados_df['MSE'].idxmin()]
mejor_alpha = mejor_fila['Alpha']
mejor_mse = mejor_fila['MSE']

# Imprimir el mejor valor de alpha y su MSE correspondiente
print(f"Mejor valor de alpha: {mejor_alpha}")
print(f"Error cuadrático medio (MSE) correspondiente: {mejor_mse}")

# Graficar la evolución del error en función de los valores de alpha en una escala logarítmica
plt.plot(resultados_df['Alpha'], resultados_df['MSE'], marker='o')
plt.xscale('log')
plt.xlabel('Valor de alpha (log scale)')
plt.ylabel('Error cuadrático medio (MSE)')
plt.title('Evolución del Error con Regularización Ridge (Escala Logarítmica)')
plt.show()
```

RESULTS



```
Mejor valor de alpha: 1e-05
Error cuadrático medio (MSE) correspondiente: 2.3680690367204795e-12
```

CONCLUSION

In this section we will finish with the comparison of both models. So, this is the conclusion:

Lacking Libraries:

Quadratic error medium in test data: 397.17147475232946

The MSE of the library-free implementation is comparatively high, suggesting a substantial degree of inaccuracy between the expected and actual values. This implies that the model is not operating at its best.

Regarding Libraries:

The corresponding mean square error (MSE) is 2.3680690367204795e-12.

The model performs incredibly well, as evidenced by the extremely low MSE of the library implementation. There appears to be a good degree of accuracy because the predictions and actual numbers are rather close.

LINKS

HTML: Classificator

GITHUB: https://github.com/2009117/Machine-Learning.git