U2 - Implementing a Predictor from scratch

Gabriela ELizabeth Avila Chan
Computational Robotics Engineering
Universidad Politécnica de Yucatán
Km. 4.5. Carretera Mérida — Tetiz
Tablaje Catastral 7193. CP 97357
Ucú, Yucatán. México
Email: 2009003@upy.edu.mx

Victor Alejandro Ortiz Santiago
Universidad Politécnica de Yucatán
Km. 4.5. Carretera Mérida — Tetiz
Tablaje Catastral 7193. CP 97357
Ucú, Yucatán. México
Email: victor.ortiz@upy.edu.mx

Abstract

This assignment involved the preparation, training, and evaluation of a predictive model using a dataset from INEGI. A target variable was selected,and then it was decided a Regression approach, and cleaned the data to ensure its suitability for model training. Then, custom algorithms were created, and them compared to see and understand their performance with scikit-learn's implementations. The documentation and code are available on GitHub. The assignment's main challenges included feature selection, algorithm choice, and performance evaluation. In the context of Robotics, this project can be applied for tasks like sensor data analysis, enabling robots to make decisions or predictions based on their sensor inputs, which is crucial for autonomous robotics applications.

U2 - Implementing a Predictor from scratch

I. Introduction

In In the field of machine learning, the correct handling of the dataset to obtain accurate data for training future intelligence is vital. In this assignment the preparation, training, and evaluation of a predictive model using data from INEGI will be presented. The assignment covers essential stages, including data acquisition, cleansing, custom model development, and performance evaluation.

The assignment began with the careful selection of a target variable, followed by the decision to frame it as a Regression problem. Data cleansing procedures addressed issues such as missing values and outliers to ensure the dataset's suitability for model training. Custom algorithms were then developed, tailored to the specific Regression problem, and compared with scikit-learn's implementations.

II. Dataset Pre-processing

Data comes in all types and forms. The process starts with taking the raw data and converting it eventually into a more understandable format for the machine, so that it can be easily interpret by the user to achieve an enterprise (output).

The very first step in this process is data preprocessing. It is a technique that is also used to convert the initial data into a standardized format. "Noisy" data needs to be cleaned and standardized for the next course of action. The aim is to make clean and formatted data available for building AI/ML models.

The words "Preprocessing" and "Processing" may sound interchangeable, but there's a fine line dividing them. Data preprocessing is nothing but a subset of the overall data processing technique. If the right data processing technique is not apply, the model will not be able to turn out meaningful or accurate information from the data analytics. [1]

In this section the process to preprocess the dataset "Indicadores_municipales_sabana_DA" provided by the INEGI, will be present.

```
import numpy as np
import pandas as pd

dataset = pd.read_csv("/content/
Indicadores_municipales_sabana_DA.csv",
encoding='ISO-8859-1')
dataset
```

The first step is to import necessary libraries, in this case, numpy was used as np to simplify function calls it is used for numerical operations; and imports pandas as pd, it provides easy-to-use data structures and functions for working with structured data, particularly tabular data. In this part, it will be used to Import the dataset. This last one, is then printed to allow the user to visualise it.

```
missing_values = dataset.isnull().sum()
for column, count in missing_values.items():
    if count > 0:
        print(f'Columns "{column}": {count}
        missing values')
```

The next part is to identify the columns within the dataset that contain missing values. It calculates the count of missing values for each column, then it iterates through the columns with missing values, and for each such column, it prints the column name and the count of missing values.

This process was done so that is easier to create a shorter variable to fill all that missing values.

```
columns_missing_values = [
    "cpic_cv", "pobtot_00", ...,
    "pobreza_patrim_00",
    "gini_90", "gini_00"
]

for column in columns_missing_values:
    if dataset[column].isnull().any():
        if dataset[column].dtype ==
        'float64' or dataset[column].dtype
        ==
        'int64':dataset[column].fillna
        (dataset[column].mean(), inplace=True)
```

As mentioned before, the "columns_missing_values" variable is used to store all the column names with missing values. The code iterates through the column names identified, and it checks whether a column contains missing values and, if so, whether it has a numeric data type ('float64' or 'int64'). If the column satisfies both conditions, it fills in the missing values with the mean of that specific column.

It was decided to use the mean to fill the missing values since this method is suitable when dealing with continuous, numeric data, and considering that float64 data is being handle, is the ideal one. It preserves data's distribution characteristics, particularly for normally distributed data, it is the best method for filling missing values while minimising the impact on data statistics and analysis.

```
columns_string_missingvalue =
dataset.columns[dataset.isnull().any()]
print(columns_string_missingvalue)
```

While filling the numerical values, the dataset showed that some of the columns with missing values were string type. Therefore, a set of column names with missing values and having string data types is created.

```
frequent_values = dataset[['gdo_rezsoc00'
, 'gdo_rezsoc05']].mode().iloc[0]
dataset['gdo_rezsoc00'].fillna
(frequent_values['gdo_rezsoc00'],
inplace=True)
dataset['gdo_rezsoc05'].fillna
(frequent_values['gdo_rezsoc05'],
inplace=True)
```

After that, the code calculates the mode (most frequent value) for the 'gdo_rezsoc00' and 'gdo_rezsoc05' columns. The missing values in these particular string columns are filled with their respective modes. It identifies columns with string data types using the select_dtypes function.

It was decide to use the mode since it is a suitable method for imputing string data because it represents the most frequent value, preserving data integrity. It was ideal since this part is dealing with categorical or non-numeric variables and the mean or median methods are not suitable for this case.

```
columns_string = dataset.select_dtypes
(include=['object']).columns
columns_string

columns_to_delete =
['ent', 'nom_ent', 'mun',
'clave_mun', 'nom_mun']
dataset.drop(columns=columns_to_delete,
inplace=True)
print(dataset)
```

The "columns_string" function it is just used to display the names of the columns of all the dataset. Since some of those columns will not be used, a set of column names that need to be removed, referred to as "columns_to_delete", is defined. The drop method is employed to eliminate these specified columns from the dataset, the columns that were drop are the one that indicate the number and name of the state, as well as the number anr name of the municipalities, since those are not rellevant to calculate the percentage What is the percentage of the illiterate population considering other socioeconomic indicators in 2010 in the capitals of the states of Mexico.

```
categorical_columns =
['gdo_rezsoc00', 'gdo_rezsoc05',
'gdo_rezsoc10']
dataset = pd.get_dummies
(dataset, columns=categorical_columns)
```

A set containing column names representing categorical string columns is established as "categorical_columns". The pd.get_dummies function is utilized to perform one-hot encoding on these categorical columns, converting them into numeric columns.

Then a function where the code calculates the sum of missing values in the dataset is created. It iterates through the columns with missing values to verify if any columns still contain missing values. If any such columns are found, their names and the count of missing values are printed. It was only used to make sure the dataset was correctly filled.

```
features = dataset.drop
(columns=['porc_pob_15_analfa10'])
target = dataset['porc_pob_15_analfa10']
```

Then the variables that will be used to test the models will be set. A variable named features is created by removing the "porc_pob_15_analfa10" column from the dataset. And, another variable, target, is established to contain the "porc_pob_15_analfa10" column.

```
columns_to_scale = features.columns

mean = features[columns_to_scale].mean()
std = features[columns_to_scale].std()

scaled_features = (features
[columns_to_scale] - mean) / std

scaled_dataset = pd.concat(
[scaled_features, target], axis=1)
```

Considering the importance of having a dataset with normalized value, a set of column names to be standardized is defined as "columns_to_scale". The mean and standard deviation of these specified columns are calculated, then the code standardizes the selected columns by subtracting the mean and dividing by the standard deviation. The scaled features are then combined with the target column to create a new dataset.

Then a function where the code calculates the sum of NaN values for each column in the "scaled_dataset". It checks if any columns within the scaled dataset have NaN values and, if so, prints the names of those columns. It was only used to make sure the dataset was correctly filled.

```
scaled_dataset.to_csv
('scaled_dataset.csv', index=False)
```

The final step involves saving the scaled_dataset to a CSV file named 'scaled_dataset.csv

III. Algorithm from Scratch

The algorithm that was selected to use is a linear regression algorithm. Linear regression is an algorithm that provides a linear relationship between an independent variable and a dependent variable to predict the outcome of future events. It is a statistical method used in data science and machine learning for predictive analysis.

The independent variable is also the predictor or explanatory variable that remains unchanged due to the change in other variables. However, the dependent variable changes with fluctuations in the independent variable. The regression model predicts the value of the dependent variable, which is the response or outcome variable being analyzed or studied.

Thus, linear regression is a supervised learning algorithm that simulates a mathematical relationship between variables and makes predictions for continuous or numeric variables such as sales, salary, age, product price, etc. [2]

Considering the research information and the dataset provide, linear regression is a smart choice because it helps predict real-world outcomes by understanding how one thing depends on another. It is like guessing future sales, salaries, or prices based on past data. This method is super useful in data science and machine learning for making these predictions with ease. This first code only uses the Numpy library, therefore the alThe process done is the following one:

```
split_ratio = 0.8
split_index = int(len
(scaled_features) * split_ratio)
```

The first thing to do is to ”split_ratio is set to 0.8, which means that 80 percent of the data will be used for training, and 20 percent for evaluation. Then, ”split_index” is calculated as an integer value representing the index at which the data will be split into training and evaluation sets.

```
scaled_features_train,
scaled_features_eval =
scaled_features[:split_index],
scaled_features[split_index:]
target_train, target_eval =
target[:split_index], target[split_index:]
```

Then, the data its split into two parts based on the previously calculated split_index. scaled_features_train contains the training data, which is the first 80 percent of the scaled_features array. scaled_features_eval contains the evaluation data, which is the remaining 20 percent of the scaled_features array. Similarly, target_train and target_eval are split from the target array accordingly.

```
def linear_regression_early_stopping
(scaled_features, target, learning_rate,
max_epochs, patience, eval_freq):
    weights = np.zeros
    (scaled_features.shape[1])
    bias = 0

    mse_values = []
    best_mse = float('inf')
    no_improvement_count = 0
```

The function linear_regression_early_stopping is defined to perform linear regression with early stopping. It takes several arguments: scaled_features (training data), target (training target), learning_rate, max_epochs (maximum number of training epochs), patience (early stopping patience), and eval_freq (frequency of evaluation).

Early stopping is implemented to prevent overfitting in machine learning models. It halts training when the model's performance on a validation set stops improving, avoiding excessive complexity and improving generalization.

Then, the weights and bias are initialized with zero values. Lists mse_values, best_mse, and no_improvement_count are initialized to keep track of mean squared errors, the best MSE seen so far, and the count of epochs with no improvement.

```
for epoch in range(max_epochs):
        target_pred = np.dot
        (scaled_features, weights) + bias
        error = target - target_pred

        gradient = np.dot
        (error, scaled_features)
        gradient = np.clip
        (gradient, -1, 1)
        weights += learning_rate
        * gradient
        bias += learning_rate
        * np.sum(error)

        mse = np.mean(error ** 2)
        mse_values.append(mse)
```

Inside the training loop, the data iterates over a range of epochs (up to max_epochs). For each epoch, it calculates predictions (target_pred) using the current weights and bias, it computes the error between the predicted values and the actual target values. Gradient descent is used to update the weights and bias to minimize the mean squared error (MSE). The calculated MSE for the current epoch is appended to the mse_values list.

```
if (epoch + 1) % eval_freq == 0:
 target_pred_eval = np.dot
 (scaled_features_eval, weights) + bias
 eval_mse = np.mean(
 (target_eval - target_pred_eval) ** 2)
 if eval_mse < best_mse:
 best_mse = eval_mse
 no_improvement_count = 0
else:
 no_improvement_count += 1
 if no_improvement_count >= patience:
 print(f"Early stopping at
 epoch {epoch+1} due to no improvement.")
```

In the early stopping evaluation, after a certain number of epochs specified by eval_freq, the code evaluates the model's performance on the evaluation dataset. It calculates the MSE between the predicted values (target_pred_eval) and the actual evaluation target values (target_eval), then checks if the evaluation MSE is better than the best seen MSE (best_mse). If the evaluation MSE is better, the best_mse is updated, and no_improvement_count is reset to 0. If the evaluation MSE does not improve for a certain number of epochs defined by patience, the training is stopped early, and a message is printed to indicate that early stopping has occurred.

```
return weights, bias, mse_values
```

The function returns the final values of weights, bias, and the list of mse_values.

```
learning_rate = 0.001
max_epochs = 2000
patience = 50
eval_freq = 50
```

In this step, the learning rate (learning_rate), maximum number of epochs (max_epochs), early stopping patience (patience), and evaluation frequency (eval_freq) are defined.

```
weights, bias, mse_values =
linear_regression_early_stopping
(scaled_features_train, target_train,
learning_rate, max_epochs,
patience, eval_freq)
```

This is the model training, where the linear_regression_early_stopping function is called with the training data and hyperparameters, and it returns the final weights, bias, and mse_values.

```
target_pred_eval =
np.dot(scaled_features_eval, weights)
+ bias
final_mse = np.mean(
(target_eval - target_pred_eval) ** 2)
```

The trained model is used to make predictions on the evaluation data and calculates the final MSE between the predicted and actual evaluation target values. To finish, using the matplotlib.pyplot , a scatter plot is created to visualize the model's performance. It shows the actual values (target_eval) and an ideal prediction line. Also, the final MSE is printed as ”Final Mean Squared Error.”

```
new_data = scaled_features_eval
predicted_values = np.dot
(new_data, weights) + bias
print("Predicted Values:",
predicted_values)

mse = np.mean((target_eval -
predicted_values) ** 2)
print("Mean Squared Error (MSE):", mse)
```

This part is used for making predictions using the trained linear regression model shown before; predicted_values: It calculates predictions by multiplying the new_data with the learned weights, then adding the bias. This simulates the linear relationship established during training. Then, once again the MSE for the predicted values is printed.

The Mean Squared Error (MSE) showed to be 3.2213023556259914, meaning that it does not predict all the values with a high accuracy.

IV. Algorithm Using Libraries

During this section, the discription will be focused on the second part of the implementation,in this one the library sklearn to compare the model's performance and results.

Lasso and Ridge Regression are the two most popular regularization techniques used in Regression analysis for better model performance and feature selection. Ridge Regression, also known as L2 regularization, is an extension to linear Regression that introduces a regularization term to reduce model complexity and help prevent overfitting.

Like Ridge Regression, Lasso Regression adds a regularization term to the linear Regression objective function. The difference lies in the loss function used – Lasso Regression uses L1 regularization, which aims to minimize the sum of the absolute values of coefficients multiplied by penalty factor λ. [3] Since both methods have pros and cons, both will be evaluated and compared using the linear regression model.

1) Lasso Regression:

```
from sklearn.model_selection
import train_test_split
from sklearn.linear_model
import Lasso, Ridge
```

train_test_split from sklearn.model_selection is used to split the dataset into training and evaluation sets. Lasso and Ridge from sklearn.linear_model are used to create Lasso and Ridge regression models, but only the Lasso model is used in this first code. matplotlib.pyplot is imported to create a scatter plot for visualizing the model's performance.

```
split_ratio = 0.8
scaled_features_train,
scaled_features_eval, target_train,
target_eval = train_test_split(
    scaled_features, target,
    test_size=1 - split_ratio
)
```

The split_ratio variable is set to 0.8, which means that 80% of the data will be used for training the model, and the remaining 20% will be used for evaluation.

Then the train_test_split function is called with scaled_features and target as inputs, along with the test size defined by $1 -$ `split_ratio`. This splits the data into training and evaluation sets.

```
lasso = Lasso(alpha=0.01, max_iter=10000)
lasso.fit(scaled_features_train,
target_train)
```

Then, the Lasso regression model is instantiated with an alpha (regularization strength) of 0.01 and a maximum number of iterations set to 10,000. The Lasso model is then fitted to the training data using lasso.fit.

```
target_pred_eval = lasso.predict
(scaled_features_eval)
mse = np.mean((target_eval -
target_pred_eval) ** 2)
```

The Lasso model is used to predict the target values for the evaluation set, and these predictions are stored in target_pred_eval. The MSE is calculated by taking the mean of the squared differences between the actual target values (target_eval) and the predicted values (target_pred_eval).

Then, a plot is created to show the model performance of the Lasso model and to compare the actual values (target_eval) and the predicted values (target_pred_eval). The calculated MSE is printed to the console, indicating how well the Lasso regression model performed.

```
lasso_predicted_values =
lasso.predict(scaled_features_eval)
print("Lasso Predicted Values:",
lasso_predicted_values)

mse = np.mean((target_eval -
lasso_predicted_values) ** 2)
```

```
print("Mean Squared Error (MSE):", mse)
```

The model is used to predict values for the evaluation set, and these predictions are stored in lasso_predicted_values, and there are printed. The MSE is calculated again and printed to the console.

2) Ridge Rgression: In this case, L2 regularization, also known as Ridge regularization, is used, this method adds a penalty term to the linear regression loss function, encouraging smaller coefficients. It does not set coefficients exactly to zero but reduces their magnitude.

The code for this one is similar to the one described above, of Lasso regression, but has some important differences, these will be explained.

```
ridge = Ridge(alpha=0.01)
ridge.fit(scaled_features_train,
target_train)
```

In the code snippet that employs L2 regularization, a Ridge model is initialized with a regularization strength (alpha) of 0.01.

The ridge.fit method is used to train the Ridge model on the training data, where scaled_features_train represents the input features, and target_train is the target variable.

```
target_pred_eval = ridge.predict
(scaled_features_eval)
mse = np.mean((target_eval -
target_pred_eval) ** 2)
```

Following training, the Ridge model is applied to make predictions on the evaluation data. The Mean Squared Error (MSE) is then computed to assess the model's performance.

To visualize the model's performance, a scatter plot is generated, akin to the L1 (Lasso) regularization snippet. This plot illustrates the relationship between actual and predicted values, offering insights into the model's effectiveness.

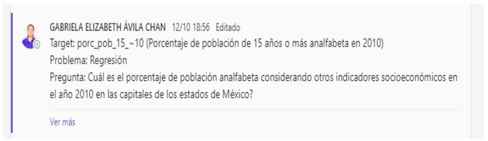
These annotations help distinguish this model from the L1 (Lasso) regularization model.

```
ridge_predicted_values =
ridge.predict(scaled_features_eval)
print("Ridge Predicted Values:",
ridge_predicted_values)

mse = np.mean((target_eval -
lasso_predicted_values) ** 2)
print("Mean Squared Error (MSE):", mse)
```

The model is used to predict values for the evaluation set, and these predictions are stored in ridge_predicted_values, and there are printed. The MSE is calculated again and printed to the console.

V. Screenshot



GABRIELA ELIZABETH ÁVILA CHAN 12/10 18:56 Editado
Target: porc_pob_15_~10 (Porcentaje de población de 15 años o más analfabeta en 2010)
Problema: Regresión
Pregunta: Cuál es el porcentaje de población analfabeta considerando otros indicadores socioeconómicos en el año 2010 en las capitales de los estados de México?

Figure 1. Screenshot of the post in the general channel.

VI. Conclusion

To sum up, developing a linear regression algorithm using only NumPy posed challenges in terms of efficiency and scalability, while scikit-learn offered a more comprehensive and user-friendly solution. The introduction of Lasso and Ridge regularization methods further improved model performance. Specifically, Ridge regression, with its ability to encourage smaller coefficients and mitigate multicollinearity, is particularly valuable in robotics.

In this field, where multiple sensors and inputs are often highly correlated, Ridge regression helps create more stable and generalized robotic control systems. By reducing overfitting and enhancing feature selection, these techniques contribute to safer and more reliable robotic operations, underlining the significance of advanced machine learning methods in the realm of robotics.

References

[1] "The importance of data preprocessing in machine learning". Express Analytics. Accedido el 25 de octubre de 2023. [En línea]. Disponible: https://www.expressanalytics.com/blog/data-preprocessing-machine-learning/

[2] "What is linear regression?- spiceworks - spiceworks". Spiceworks. Accedido el 25 de octubre de 2023. [En línea]. Disponible: https://www.spiceworks.com/tech/artificial-intelligence/articles/what-is-linear-regression/

[3] "Guide to lasso and ridge regression techniques with use cases". Blogs Updates on Data Science, Business Analytics, AI Machine Learning. Accedido el 25 de octubre de 2023. [En línea]. Disponible: https://www.analytixlabs.co.in/blog/lasso-and-ridge-regression/