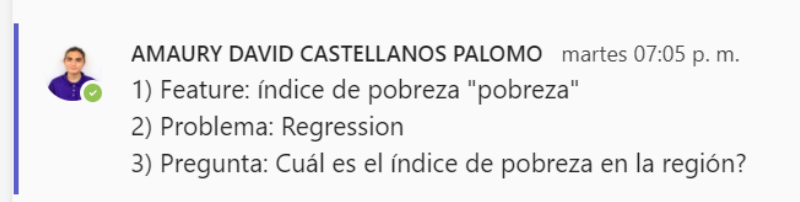
U2 - Implementing a Predictor from scratch

Amaury David Castellanos Palomo

A- Screenshot of the post in the general channel, where you state your Selected target, your selected problem (regression or classification), and the question that the predictor will answer.



B- Chronological document or journal

Step 1 – Clean the dataset.

The original dataset was a 2456x 139 matrix containing 26 columns of integers, 108 of floats and 5 of strings. Since the problem is a regression type, I decided to erase all columns that weren't numbers, that is to say any column that contained a string will be gone.

The first step was to eliminate all nan values, this is done so that there isn't any "hole" in my data, had there be any nan, this code would replace it with a 0

```
#Replace NAN with 0
OriginalDs = OriginalDs.fillna(0)
```

Next we select which column will be our target, I choose "pobreza" as the column that my model will predict, and we create the dataset that will be used to train the model. The pobrezaDs DataFrame is created by removing several columns (features) from OriginalDs. These columns are either unrelated to the target variable or contain non-numeric data.

```
target_pobreza = OriginalDs["pobreza"]
pobrezaDs = OriginalDs.drop(columns=[
    "mun",
    "clave_mun",
    "ent",
    "nom_ent",
    "nom_mun",
    "gdo_rezsoc00",
    "gdo_rezsoc05",
    "gdo_rezsoc10",
    "rankin_p",
    "rankin_pe",
    "N_pobreza",
    "N_pobreza_e",
    "N_pobreza_m",
```

Step 2 – Split the dataset

After that we split our data into 4 datasets

```
test_size = 0.2 #20 per cent

# Split the dataset and target into training and testing sets
trainPobreza_ds, testPobreza_ds, trainPobreza_target, testPobreza_target = train_test_split(
    pobrezaDs, target_pobreza, test_size=test_size)

#make pandas' datasets into numpy arrays
trainPobreza_target_Numpy = trainPobreza_target.to_numpy()
trainPobreza_ds_Numpy = trainPobreza_ds.to_numpy()

testPobreza_target_Numpy = testPobreza_target.to_numpy()
testPobreza_ds_Numpy = testPobreza_ds.to_numpy()
```

As we see in the code, we used "train_test_split" to separate the data into training and testing sets. "test_size" determines the proportion of the data that will be used for testing (in this case, 20%). The resulting datasets, "trainPobreza_ds" and "testPobreza_ds," represent the feature matrices for training and testing, while "trainPobreza_target" and "testPobreza_target" represent the corresponding target vectors. To ensure compatibility with machine learning models, the datasets are then converted from pandas DataFrames into NumPy arrays. This preprocessing is essential for training and evaluating machine learning models efficiently.

Step 3 – Define the model

The code a simplified linear regression model from scratch. It is designed and coded as follows:

Initialization Method

The constructor for the linear regression class is defined in the __init__() method. Two essential parameters are provided:

- learning_rate: The learning rate for gradient descent.
- epochs: The number of training iterations (epochs).

The weights and bias are initialized as None and will be set during training.

```
class myLinearRegression:
    def __init__(self, learning_rate, epochs):
        self.learning_rate = learning_rate
        self.epochs = epochs
        self.weights = None
        self.bias = None
```

Training Method (Gradient Descent)

The training process for the linear regression model is implemented in the train() method using gradient descent. The following steps are taken:

1. Obtain the number of samples and features from the input data X.
2. Initialize the weights and bias with zeros.

3. Run a loop for the specified number of epochs.
4. In each epoch, compute the predicted values y_predicted using the current weights and bias.
5. Calculate the gradients for the weights (dw) and bias (db) based on the mean squared error loss function.
6. Update the weights and bias using the computed gradients and learning rate.

This process is iteratively repeated to refine the model's parameters and minimize the error.

```
def train(self, X, y):
    #num_ints = X.shape
    num_samples, num_features = X.shape
    self.weights = np.zeros(num_features)  # Correct shape for weights
    self.bias = 0

    for _ in range(self.epochs):
        y_predicted = np.dot(X, self.weights) + self.bias

        dw = (1 / num_samples) * np.dot(X.T, (y_predicted - y))
        db = (1 / num_samples) * np.sum(y_predicted - y)

        self.weights -= self.learning_rate * dw
        self.bias -= self.learning_rate * db
```

Prediction Method

The predict() method allows users to make predictions using the trained model. It calculates the predicted values for the input data X based on the learned weights and bias.

```
def predict(self, X):
    y_predicted = np.dot(X, self.weights) + self.bias
    return y_predicted
```

Mean Squared Error Method (MSE)

The MSE() method calculates the mean squared error between the true target values y_true and the predicted values y_pred. It computes the squared differences, sums them, and divides by the number of samples to get the mean squared error.

```
def MSE(y_true, y_pred):
    n = len(y_true)
    e= np.sum((y_true-y_pred)**2)
    mean= e/n

    return mean
```

Step 2 - Train the model.

Setting Up Data

The training features or input data for the linear regression model are assigned to the variable trainPobreza_ds_Numpy. The target values or ground truth data you want to predict are assigned to the variable trainPobreza_target_Numpy.

Instantiating the Model

An instance of the custom linear regression model (myLinearRegression) is created with the following parameters:

- learning_rate: This is set to an extremely small value, 0.00000000001, to stabilize the learning process when using gradient descent. In this case, an exceptionally small value has been chosen because otherwise all the weight would turn to NAN.
- epochs: This is set to 2000, which defines the number of training iterations.

Training the Model

The model is then trained using the training data (X and y) with the specified learning rate and number of epochs. During training, the model learns to minimize the mean squared error between its predictions and the actual target values.

```
[24] X = trainPobreza_ds_Numpy
     y = trainPobreza_target_Numpy
                                          #0.00000000001
     model4 = myLinearRegression(learning_rate=0.00000000001, epochs=2000)  #learning_rate=0.001, epochs=1000
     model4.train(X, y)
```

Making Predictions

After training, the trained model is used to make predictions on the test data (testPobreza_ds_Numpy). The predictions are stored in the variable predictions.

Calculating Accuracy (Mean Squared Error)

The accuracy is calculated using the Mean Squared Error (MSE) method, which measures the average squared difference between the real target values (testPobreza_target_Numpy) and the predicted values (predictions). Lower MSE values indicate a better fit of the model to the data. The calculated MSE is stored in the variable myAccuracy.

Printing the Accuracy

Finally, the calculated accuracy using MSE is printed, labeled as "My Accuracy." This value represents how well the model performs on the test data, with lower values indicating better predictive accuracy.

```
predictions = model4.predict(testPobreza_ds_Numpy)
myAccuracy = myLinearRegression.MSE(testPobreza_target_Numpy,predictions) # (real, prediction)
print("My Accuracy: ", myAccuracy)
```

Step 3 – compare results with predictor from libraries.

When comparing my scores vs the s scikit learn, we can see there is a huge gap between them. I believe this is due to the fact the regression model from scikit-learn applies feature scaling before processing the datasets.

In linear regression, the coefficients (weights) associated with each feature determine the impact of that feature on the predicted outcome. If features have different scales, the magnitude of the weights can vary widely.

```
"
LR_Sklearn = LinearRegression().fit(X, y)
test_score = LR_Sklearn.score(testPobreza_ds_Numpy, testPobreza_target)

#MSE = LR_Sklearn.mean_squared_error(testPobreza_target_Numpy,predictions)
print("SKL_accuracy: ", test_score) #<--- :(
```

```
My Accuracy:  4699.372514409704
SKL_accuracy:  1.0
```

Step 4 – link to github. :)

https://github.com/Amaury137/MachineLEarning9A/tree/master

C – Conclusion (A reflexion over what were your main challenges during this project, how do you see this could be applied to the field of Robotics.)

Throughout this project, one of the primary challenges was implementing the weight optimization algorithm from scratch. This required a deep understanding of linear regression and gradient descent. Developing a custom linear regression model involved writing code to calculate the gradients and iteratively update the model's weights and bias.